QuickDraw 3D Technical Reference

3D Graphics Programming With QuickDraw 3D 1.5.4

Including *3D Metafile Reference* and *Renderer Acceleration Virtual Engine*

Apple Technical Publications

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings

Chapter 1 Introduction to QuickDraw 3D 41

Chapter 2 3D Viewer 91

Chapter 3 QuickDraw 3D Objects 163

Chapter 4 Geometric Objects 237

About This Document

This Document, *3D Graphics Programming With QuickDraw 3D 1.5*, describes QuickDraw 3D 1.5, a graphics library that you can use to define three-dimensional (3D) models, apply colors and other attributes to parts of the models, and create images of those models. You can use these capabilities to develop a wide range of applications, including interactive three-dimensional modeling, simulation and animation, data visualization, computer-aided drafting and design, games, and many other uses.

QuickDraw 3D 1.5 provides these basic services:

- A large number of predefined geometric object types. You can create multiple instances of any type of object and assign them individual characteristics.
- Support for standard lighting types and illumination algorithms.
- Support for standard methods of projecting a model onto a viewing plane.
- Ability to perform both immediate and retained mode rendering, and support for multiple rendering styles.
- Built-in support for reading and writing data stored in a standard 3D data file format (the QuickDraw 3D 1.5 Object Metafile).
- Support for any available 3D pointing devices, including devices that provide multiple degrees of freedom.
- Support for multiple operating and window systems. QuickDraw 3D 1.5 is extremely portable and operates independently of the native window system. It provides consistent capabilities and performance across all supported platforms.
- Fast interactive rendering.

This document describes the application programming interfaces that you can use to develop applications and other software using QuickDraw 3D 1.5. Although QuickDraw 3D 1.5 provides a large set of basic 3D objects and operations, it is also designed for easy extensibility, so that you can add custom capabilities (for instance, custom object types, attributes, renderers, and shading algorithms) to those provided by QuickDraw 3D 1.5.

To use this document, you should be generally familiar with computer graphics and with 3D modeling and rendering techniques. This document explains some of the fundamental 3D concepts, but it is not intended to be either an introduction to or a technical reference for 3D graphics in general. Rather, it explains how QuickDraw 3D 1.5 implements the standard techniques for 3D modeling, rendering, and interaction. You can consult the Bibliography near the end of this document for a list of some books that might help you acquire a basic knowledge of those techniques.

Note
The book *3D Computer Graphics,* second edition, by Alan Watt is particularly helpful for beginners. ◆

You should also be familiar with the techniques that underlie object-oriented programming. QuickDraw 3D 1.5 is object oriented in the sense that many of its capabilities are accessed by creating and manipulating QuickDraw 3D 1.5 objects. In addition, QuickDraw 3D 1.5 classes (of which QuickDraw 3D 1.5 objects are instances) are arranged in a hierarchy, which provides for method inheritance and method overriding.

Note
Currently, only C language programming interfaces are available. ◆

You should begin this Document by reading the chapter "Introduction to QuickDraw 3D." That chapter describes the basic capabilities provided by QuickDraw 3D 1.5 and the QuickDraw 3D 1.5 application programming interfaces that you use to create and manipulate objects in that hierarchy. It also provides source code samples illustrating how to use QuickDraw 3D 1.5 to define, configure, and render simple 3D models.

If you just want to be able to display an existing 3D model in a window and don't need to use the powerful capabilities of QuickDraw 3D 1.5, you can use the 3D Viewer supplied with QuickDraw 3D 1.5. The 3D Viewer allows you to display 3D data with minimal programming effort. It is therefore analogous to the movie controller provided with QuickTime. Read the chapter "3D Viewer" for complete information.

Once you are familiar with the basic uses of QuickDraw 3D 1.5, you can read the remaining chapters in this document for more information on any particular topic. For example, for complete information on the types of lights provided by QuickDraw 3D 1.5, see the chapter "Light Objects."

Companion Documents

Two other Apple documents contain information that extends and amplifies the content of this document:

- *3D Metafile 1.5 Reference* describes the 3D Metafile, a file format designed to permit the storage and interchange of 3D data.
- *QuickDraw 3D 1.5 1.5 Renderer Acceleration Virtual Engine* describes RAVE, the part of the QuickDraw 3D 1.5 Macintosh system software that controls 3D drawing engines, also known as 3D drivers.

These documents are available in online form on the QuickDraw 3D 1.5 SDK.

Format of a Typical Chapter

Almost all chapters in this document follow a standard structure. For example, the chapter "Attribute Objects" contains these sections:

- "About Attribute Objects." This section provides an overview of the features QuickDraw 3D 1.5 provides for managing attribute objects.
- "Using Attribute Objects." This section describes the tasks you can accomplish using attribute objects.
- "Attribute Objects Reference." This section provides a complete reference for QuickDraw 3D 1.5 attribute objects by describing the constants, data structures, and routines you can use to manage attribute objects. Each routine description also follows a standard format, which presents the routine declaration followed by a description of every parameter of the routine.
- "Attribute Errors." This section lists error messages (as well as warnings and notices) that attribute routines may return.

Note
At the end of this document are a bibliography, a glossary, an index of API elements, and a general index. ◆

Conventions Used in This Document

This document uses special conventions to present certain types of information. Words that require special treatment appear in specific fonts or font styles. Certain information, such as parameter blocks, appears in special formats so that you can scan it quickly.

Special Fonts

All code listings, reserved words, and the names of actual data structures, constants, fields, parameters, and routines are shown in Letter Gothic (`this is Letter Gothic`).

Words that appear in **boldface** are key terms or concepts and are defined in the glossary.

Types of Notes

There are several types of notes used in this document.

Note
A note like this contains information that is interesting but possibly not essential to an understanding of the main text. (An example appears on page 42.) ◆

IMPORTANT
A note like this contains information that is essential for an understanding of the main text. (An example appears on page 53.) ▲

▲ **WARNING**
Warnings like this indicate potential problems that you should be aware of as you design your application. Failure to heed these warnings could result in system crashes or loss of data. (An example appears on page 991.) ▲

Development Environment

The system software routines described in this document are available using C interfaces. How you access these routines depends on the development environment you are using. When showing QuickDraw 3D 1.5 routines, this document uses the C interfaces available with the Macintosh Programmer's Workshop (MPW).

All code listings in this document are shown in C. They show ways of using various routines and illustrate techniques for accomplishing particular tasks. All code listings have been compiled and, in most cases, tested. However, Apple Computer, Inc., does not intend for you to use these code samples in your application.

For More Information

For information about Apple technology and developer support programs, connect to one of the following online sites:

`devworld.apple.com` for information about Apple Developer Programs

`quicktime.apple.com` for information about Apple QuickTime technologies

`www.apple.com`. for general information about Apple Computer, Inc.

`devworld.apple.com/techinfo/techdocs/index.html` for Apple Technical Documentation

P R E F A C E

CHAPTER 1

Introduction to QuickDraw 3D

This chapter provides an introduction to QuickDraw 3D, a graphics library that you can use to manage virtually all aspects of 3D graphics, including modeling, rendering, and data storage. For example, you can use QuickDraw 3D to define three-dimensional models, apply colors or other attributes to parts of the models, and create images of those models. QuickDraw 3D provides a large set of capabilities for creating and interacting with models of 3D objects. In addition, QuickDraw 3D is easily extensible in many ways, so you can, if necessary, add capabilities that are not provided by QuickDraw 3D.

This chapter begins by describing the basic capabilities provided by QuickDraw 3D. Then it describes the application programming interfaces that you use to create and manipulate QuickDraw 3D objects. The section "Using QuickDraw 3D," beginning on page 52 provides source code examples illustrating how to use QuickDraw 3D to define, configure, and render simple three-dimensional objects. The section "QuickDraw 3D Reference," beginning on page 71, describes the QuickDraw 3D routines you need to use to initialize and terminate QuickDraw 3D, as well as some basic routines for managing sets, shapes, and strings.

About QuickDraw 3D

QuickDraw 3D is a graphics library developed by Apple Computer that you can use to create, configure, and render three-dimensional objects. It is specifically designed to be useful to a wide range of software developers, from those with very little knowledge of 3D modeling concepts and rendering techniques to those with very extensive experience with those concepts and techniques.

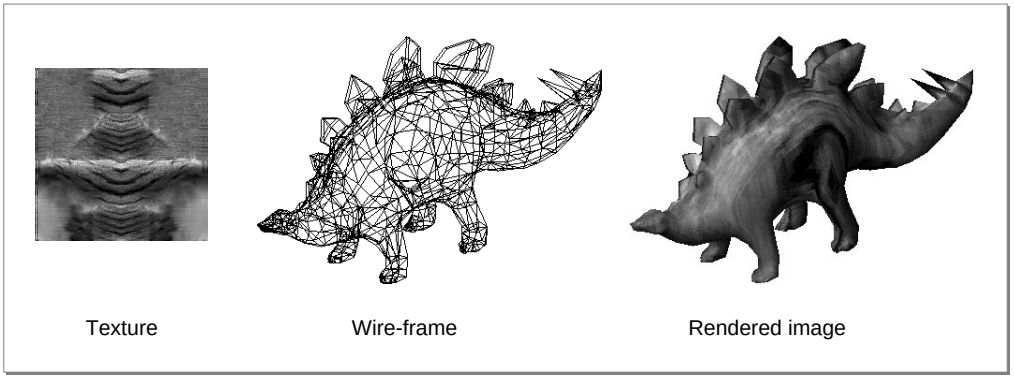
At the most basic level, you can use the file format and file-access routines provided by QuickDraw 3D to read and display 3D graphics created by another

application. For example, a word-processing application might want to import a picture created by a 3D modeling or image-capturing application. QuickDraw 3D supports the **3D Viewer,** which you can use to display 3D data and objects in a window and allow users limited interaction with that data, without having to learn any of the core QuickDraw 3D application programming interfaces.

Note
See the chapter "3D Viewer" for complete information about the 3D viewer, as well as complete source code samples illustrating how to create and manage a viewer object. ◆

You can also use QuickDraw 3D for more sophisticated applications, such as interactive 3D modeling and rendering, animation, data visualization, or any of thousands of other ways of interpreting and displaying data in three (or more) dimensions. Figure 1-1 illustrates the kinds of images you can produce using QuickDraw 3D. It shows a texture, a wireframe model, and the result of applying the texture to that model.

Figure 1-1 A simple three-dimensional picture



Modeling and Rendering

To create images such as that shown in Figure 1-1, you typically engage in at least two distinguishable main tasks: modeling and rendering. **Modeling** is the

process of creating a representation of real or abstract objects, and **rendering** is the process of creating an image (on the screen or some other medium) of a model. QuickDraw 3D subdivides each of these tasks into a number of subtasks.

In QuickDraw 3D, *modeling* involves

- creating, configuring, and positioning basic *geometric objects* and *groups* of geometric objects. QuickDraw 3D defines many basic types of geometric objects and a large number of ways to *transform* such objects.
- assigning sets of *attributes* (such as diffuse and specular colors) to objects and *parts of* objects.
- applying *textures* to surfaces of objects.
- configuring a model's *lights* and *shading*. QuickDraw 3D supplies four types of lights (ambient light, directional lights, spot lights, and point lights) and several types of shaders.

In QuickDraw 3D, *rendering* involves

- specifying a *camera* position and type. A camera type is defined by a method of projecting the model onto a flat surface, called the view plane. QuickDraw 3D provides two types of cameras that use perspective projection (the aspect ratio and view plane cameras) and one type of camera that uses parallel projection (the orthographic camera).
- specifying a *renderer* or method of rendering. QuickDraw 3D provides a wireframe and an interactive renderer. Renderers support different *styles* of rendering (for example, points, edges, or filled shapes).
- creating a *view* (a collection of a group of lights, a camera, and a renderer and its styles) and rendering the model using the view to create an *image.*

Interacting

Often, modeling and rendering are not easily separable, particularly in applications that support interactive 3D modeling. When, for example, the user selects a sphere and drags it using the mouse or other pointing device, the application needs to change the model (reposition the sphere) and render a new image. (Indeed, the application may generate a series of new images to show the sphere changing location as the user drags it.) QuickDraw 3D supports a third main task, **interacting** with a model (that is, selecting and manipulating objects in the model).

In QuickDraw 3D, *interacting* involves

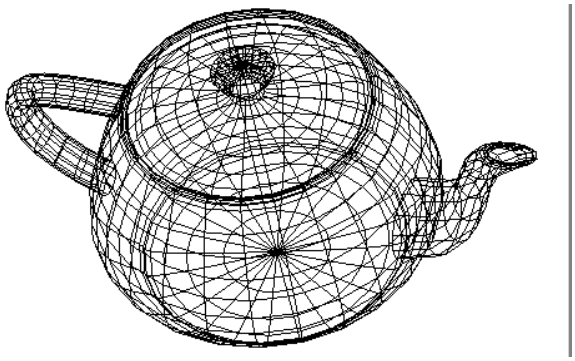
- determining what kinds of *pointing devices* are available on a particular computer and possibly configuring one or more of those devices to control items in a 3D model (such as a camera or a light).
- identifying the objects in a model that are close to the cursor when the user clicks or drags in the model's image. This is called *picking*.

QuickDraw 3D supplies an extensive set of routines that you can use to perform these tasks. For complete details, see the chapters "Pointing Device Manager" and "Pick Objects."

Extending QuickDraw 3D

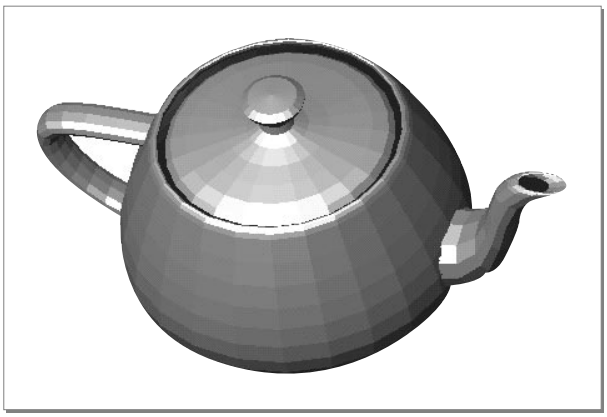
QuickDraw 3D is designed to be easily extensible, so that you can, if necessary, add capabilities that are not part of the basic QuickDraw 3D feature set. For instance, you've already seen that QuickDraw 3D supplies two types of renderers, the wireframe and interactive renderers. The wireframe renderer creates line renderings of models, as illustrated in Figure 1-2.

Figure 1-2 A model rendered by the wireframe renderer



The interactive renderer uses a more complex rendering algorithm that allows illumination and shading effects to be produced. Figure 1-3 shows the same teapot model rendered by the interactive renderer.

Figure 1-3 A model rendered by the interactive renderer



It's possible that some applications require even more complex rendering algorithms to render images adequately from a 3D model. For example, you might want to define a ray tracing renderer to support additional lighting effects such as reflection and refraction. In those cases, the application can define and register a custom renderer with QuickDraw 3D and then use that renderer in exactly the same way it would use any standard QuickDraw 3D renderer.

QuickDraw 3D is extensible in several ways:

- You can define *custom object types* to augment the standard QuickDraw 3D object types.
- You can define *custom attributes* and assign them to shapes or sets.
- You can define *custom shaders* to create special shading effects or to handle any custom attributes you've defined. (The shaders that QuickDraw 3D supplies can handle all the predefined attribute types.)
- You can define *custom renderers* to support other rendering algorithms.

In addition, QuickDraw 3D is designed to be portable to other software platforms and to support a variety of hardware accelerators:

- QuickDraw 3D is *cross-platform.* It is available for the PowerPC version of the Mac OS and for the Microsoft Win32 API (running on either Windows 95 or on the Intel processor version of Windows NT 3.51 and later). This portability

to other window systems is accomplished by isolating all window system-specific information into a layer called a *draw context*, which is associated with a view. QuickDraw 3D automatically handles system-dependent issues such as byte ordering.

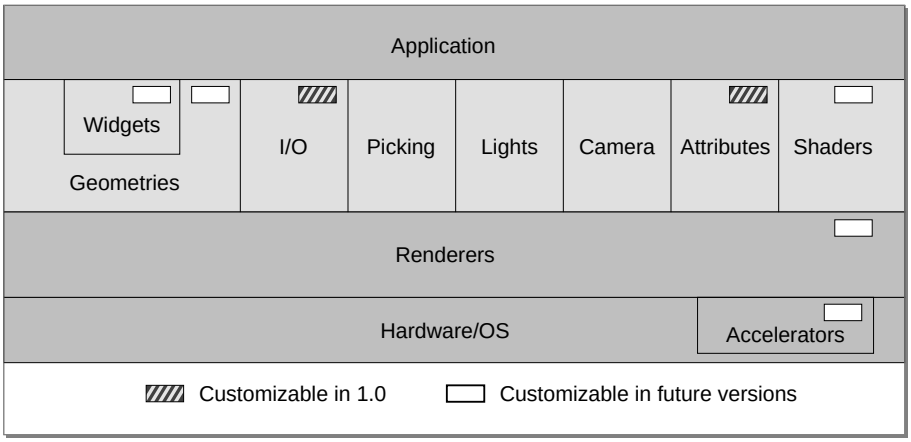
- QuickDraw 3D renderers can take advantage of *hardware accelerators*, if available.

Finally, QuickDraw 3D defines a platform-independent **metafile** (that is, a file format) for storing and interchanging 3D data. This metafile is intended to provide a standard format according to which applications can read and write 3D data, even applications that use 3D graphics systems other than QuickDraw 3D. QuickDraw 3D itself includes routines that you can use to read and write data in the metafile format. Apple Computer, Inc. also supplies a parser that you can use to read and write metafile data on operating systems that do not support QuickDraw 3D.

Note
For further information about the metafile format, see *3D Metafile 1.5 Reference.* This document is available online in the QuickDraw 3D 1.5 SDK. ◆

Figure 1-4 shows the functional components of QuickDraw 3D.

Figure 1-4 The parts of QuickDraw 3D



Naming Conventions

The QuickDraw 3D application programming interfaces are designed, as much as possible, to mirror the QuickDraw 3D class hierarchy described in the chapter "QuickDraw 3D Objects." They are also designed to exhibit as much uniformity as can reasonably be achieved by names describing a large and heterogeneous collection of objects instantiating classes in that hierarchy. Ideally, once you are acquainted with the various conventions governing the programming interfaces and the class hierarchy, you should be able to make correct guesses about the names of constants, data structures, and routines. In very many cases, the names of constants and routines are largely self-documenting, thanks to a strict adherence to the naming conventions. This section describes those conventions and provides some examples.

Constants

All constants defined in the QuickDraw 3D application programming interfaces have the prefix `kQ3`. Very simple constants consist solely of the `kQ3` prefix and a specific value indicator. Here are some examples:

```
typedef enum TQ3Boolean {
    kQ3False,
    kQ3True
} TQ3Boolean;

typedef enum TQ3Switch {
    kQ3Off,
    kQ3On
} TQ3Switch;

typedef enum TQ3Status {
    kQ3Failure,
    kQ3Success
} TQ3Status;
```

Most other enumerated constants consist of the standard `kQ3` prefix, followed by a type, followed by a specific value. Here are some examples:

```
typedef enum TQ3Axis {
    kQ3AxisX,
    kQ3AxisY,
    kQ3AxisZ
} TQ3Axis;
```

Other constants are defined using the C preprocessor `#define` mechanism. Here are some examples:

```
#define kQ3ObjectTypeElement            Q3_OBJECT_TYPE('e','l','m','n')
#define kQ3ObjectTypePick               Q3_OBJECT_TYPE('p','i','c','k')
#define kQ3ObjectTypeShared             Q3_OBJECT_TYPE('s','h','r','d')
#define kQ3ObjectTypeView               Q3_OBJECT_TYPE('v','i','e','w')
#define kQ3ObjectTypeInvalid            0
```

In general, these kinds of constants specify types of objects in the QuickDraw 3D class hierarchy or methods defining the behaviors of those types. These constants use the macros `Q3_OBJECT_TYPE` or `Q3_METHOD_TYPE`. See the header file `QD3D.h` for definitions of these macros.

Data Types

All data structures and data types defined in the QuickDraw 3D application programming interfaces have the prefix `TQ3`. Like constant names, data type names never contain the underscore character (_). When emphasis is required, subwords of a data type name are capitalized and usually proceed from general to specific.

There are four distinguishable classes in data type names.

- Opaque objects, whose definitions are private, begin with the prefix `TQ3` and end with the suffix `Object`. Between the prefix and the suffix are one or more words indicating the type of the opaque object. Here are some examples:

```
TQ3GeometryObject
TQ3ViewObject
TQ3CameraObject
TQ3StyleObject
TQ3DrawContextObject
```

- Data structures used in defining characteristics of opaque objects begin with the prefix `TQ3` and end with the suffix `Data`. Between the prefix and the suffix

are one or more words indicating the type of the object. Here are some examples:

```
TQ3TriangleData
TQ3BoxData
TQ3OrthographicCameraData
```

- Data structures that contain data not specifically used to define characteristics of an opaque object begin with the prefix `TQ3`. Following the prefix are one or more words indicating the type of the data the structure contains. Here are some examples:

```
TQ3Point3D
TQ3Vector2D
TQ3ColorRGB
TQ3ColorARGB
```

- Attributes are opaque objects, but they are named differently to distinguish them from other opaque objects. Attributes are of type `TQ3Attribute`.

IMPORTANT

All floating-point numbers used in the QuickDraw 3D application programming interfaces are single precision. ▲

Functions

All functions defined in the QuickDraw 3D application programming interfaces have the prefix `Q3`. The *class* of an identifier immediately follows its type prefix. Then the *method* occurs, separated from the class by an underscore. A method is almost always expressed as a verb-noun sequence. Here are some examples:

```
Q3Polygon_GetVertexPosition
Q3NURBCurve_SetControlPoint
Q3Light_SetBrightness
Q3SpotLight_GetFallOff
Q3View_GetLocalToWorldInverseTransposeMatrixState
Q3Triangle_New
```

Some functions are so simple that they have no distinguishable class and method. Here are some examples:

```
Q3Initialize
Q3IsInitialized
Q3Exit
```

As much as possible, function parameters are ordered consistently throughout the application programming interfaces. In virtually all cases, the first parameter is a data type that corresponds to the object being operated on. When there are two or more additional parameters, they are placed in their natural or intuitive ordering.

Most QuickDraw 3D functions return a status code, which is of type `TQ3Status`. A status code is either `kQ3Success` or `kQ3Failure`, indicating that the function has succeeded or failed. When a function fails, you can call a further function to get a specific error code. Alternatively, you can install an error-reporting callback routine to handle failures. See the chapter "Error Manager" for complete details on handling errors.

Functions that create opaque objects usually return a function result whose type is a reference to the type of the newly created object (for instance, `TQ3CameraObject` for a new camera object). An object reference is an opaque pointer to the object. When these kinds of routines fail, they return the value `NULL`.

Retained and Immediate Modes

A graphics system operates in **retained mode** if it retains a copy of all the data describing a model. In other words, a retained mode graphics system requires you to completely specify a model by passing model data to the system using predefined data structures. The graphics system organizes the data internally, usually in a hierarchical database. Once an object is added to that database, you can change the object only by calling specific editing routines provided by the graphics system.

By contrast, a graphics system operates in **immediate mode** if the application itself maintains the data that describe a model. For example, original QuickDraw is a two-dimensional graphics system that operates in immediate mode. You draw objects on the screen, using QuickDraw, by calling routines that completely specify the objects to be drawn. QuickDraw does not maintain any information about a picture internally; it simply takes the data provided by the application and immediately draws the appropriate objects.

Note
OpenGL™ is an example of a 3D graphics system that operates in immediate mode. QuickDraw GX is an example of a 2D graphics system that operates in retained mode. ◆

QuickDraw 3D supports both immediate and retained modes of specifying and drawing models. The principal advantage of immediate mode imaging is that the model data is immediately available to you and is not duplicated by the graphics system. The data is stored in whatever form you like, and you can change that data at any time. The main disadvantage of immediate mode imaging is that you need to maintain the sometimes quite lengthy object data, and you need to perform geometric operations on that data yourself. In addition, it can be difficult to accelerate immediate mode rendering, because you generally need to specify the entire model to draw a single frame, whether or not the entire model has changed since the previous frame. This can involve passing large amounts of data to the graphics system.

Retained mode imaging typically supports higher levels of abstraction than immediate mode imaging and is more amenable to hardware acceleration and caching. In addition, the hierarchical arrangement of the model data allows the graphics system to perform very quick updates whenever the data is altered. To avoid duplicating data between your application and the graphics system's database, your application should match the data types of the graphics system and use the extensive editing functions to change a model's data.

Another important advantage of retained mode imaging is that it's very easy to read and write retained objects.

To create a point, for example, in retained mode, you fill in a data structure of type `TQ3PointData` and pass it to the `Q3Point_New` function. This function copies the data in that structure and returns an object of type `TQ3GeometryObject`, which you use for all subsequent operations on the point. For example, to draw the point in retained mode, you pass that geometric object returned by `Q3Point_New` to the `Q3Geometry_Submit` function inside a rendering loop. To change the data associated with the point, you call point-editing functions, such as `Q3Point_GetPosition` and `Q3Point_SetPosition`. Finally, when you have finished using the point, you must call `Q3Object_Dispose` to have QuickDraw 3D delete the point from its internal database.

It's much simpler to draw a point in immediate mode. You do not need to call any QuickDraw 3D routine to create a point in immediate mode; instead, you merely have to maintain the point data yourself, typically in a structure of type `TQ3PointData`. To draw a point in immediate mode, you call the `Q3Point_Submit`

function, passing it a pointer to that structure. When you're using immediate mode, however, you need to know exactly what types of objects you're drawing and hard code the appropriate routines in your source code.

Note
Immediate mode rendering does not require any memory permanently allocated to QuickDraw 3D, but it might require QuickDraw 3D to perform temporary allocations while rendering is occurring. ◆

In general, if most of a model remains unchanged from frame to frame, you should use retained mode imaging to create and draw the model. If, however, many parts of the model do change from frame to frame, you should probably use immediate mode imaging, creating and rendering a model on a shape-by-shape basis. You can, of course, use a combination of retained and immediate mode imaging: you can create retained objects for the parts of a model that remain static and draw quickly changing objects in immediate mode.

Using QuickDraw 3D

This section describes the most basic ways of using QuickDraw 3D. In particular, it provides source code examples that show how you can

- determine whether QuickDraw 3D is available
- initialize a connection to QuickDraw 3D and later close that connection
- create and configure geometric objects in a three-dimensional model
- specify a group of lights to illuminate those objects
- create a camera to specify a point of view and a method of projecting the three-dimensional model to create a two-dimensional image of the model
- render (that is, draw) the model

For complete details on any of these topics, you should read the corresponding chapter later in this book. For example, see the chapter "Light Objects" for complete information about the types of lights provided by QuickDraw 3D.

IMPORTANT

The code samples shown in this section provide only very rudimentary error handling. You should read the chapter "Error Manager" to learn how to write and register an application-defined error-handling routine, or how to determine explicitly which errors have occurred during the execution of QuickDraw 3D routines. ▲

QuickDraw 3D currently is supported for the PowerPC version of the Mac OS and for the Win32 API. It exists as a shared library, in two forms:

- An optimized version of the QuickDraw 3D shared library is available for end users of those applications and other products.
- A debugging version is available for use by developers while writing their applications or other software products. The debugging version provides more extensive information than the optimized version. For instance, the debugging version of QuickDraw 3D issues errors, warnings, and notices at the appropriate times; the optimized version issues only errors and warnings.

Compiling Your Application

In order for your application's code to work correctly with the code contained in the QuickDraw 3D shared library, you need to ensure that you use the same compiler settings that were used to compile the QuickDraw 3D shared library. Otherwise, it's possible for QuickDraw 3D to misinterpret information you pass to it. For example, all the enumerated constants defined by QuickDraw 3D are of the `int` data type, where an `int` value is 4 bytes. If your application passes a value of some other size or type for one of those constants, it's likely that QuickDraw 3D will not correctly interpret that value. Accordingly, if the default setting of your compiler does not make enumerated constants to be of type `int`, you must override that default setting, typically by including `pragma` directives in your source code or by using an appropriate compiler option.

There are currently three important compiler settings:

- Enumerated constants are of the `int` data type.
- Elements of type `char` or `short` that are contained in an array that is contained in a structure may be aligned on non-longword boundaries.
- Fields in a structure that contain pointers or data of type `long`, `float`, or `double` are aligned on longword boundaries.

The interface file QD3D.h contains compiler pragmas for several popular C compilers. For example, QD3D.h contains this line for the PPCC compiler, specifying field alignment on longword boundaries for pointers or data of type long, float, or double:

```
#pragma options align=power
```

Some compilers might not provide pragmas for the three important compiler settings listed above. For example, the PPCC compiler does not currently provide a pragma for setting the size of enumerated constants. PPCC does however support the -enums compiler option, which you can use to set the size of a enumerated constants.

IMPORTANT

Consult the documentation for your compiler to determine how to specify the size of enumerated constants and to configure structure field alignment so as to conform to the settings of QuickDraw 3D. ▲

Initializing and Terminating QuickDraw 3D

Before calling any QuickDraw 3D routines, you need to verify that the QuickDraw 3D software is available in the current operating environment. Then you need to create and initialize a connection to the QuickDraw 3D software.

On the Mac OS, you can verify that QuickDraw 3D is available by calling the MyEnvironmentHasQuickDraw3D function defined in Listing 1-1.

Listing 1-1 Determining whether QuickDraw 3D is available

```
Boolean MyEnvironmentHasQuickDraw3D (void)
{
    return (long) Q3Initialize != kUnresolvedSymbolAddress;
}
```

The MyEnvironmentHasQuickDraw3D function checks to see whether the address of the Q3Initialize function has been resolved. If it hasn't been resolved (that is, if the Code Fragment Manager couldn't find the QuickDraw 3D shared library when launching your application), MyEnvironmentHasQuickDraw3D returns the

value FALSE to its caller. Otherwise, if the address of the Q3Initialize function was successfully resolved, MyEnvironmentHasQuickDraw3D returns TRUE.

Note
For the function MyEnvironmentHasQuickDraw3D to work properly, you must establish soft links (also called *weak links*) between your application and the QuickDraw 3D shared library. For information on soft links, see the book *Inside Macintosh: PowerPC System Software.* For specific information on establishing soft links, see the documentation for your software development system. ◆

On the Mac OS, you can verify that QuickDraw 3D is available in the current operating environment by calling the Gestalt function with the gestaltQD3D selector. Gestalt returns a long word whose value indicates the availability of QuickDraw 3D. Currently these values are defined:

```
enum {
    gestaltQD3DNotPresent                       = 0,
    gestaltQD3DAvailable                        = 1
}
```

You should ensure that the value gestaltQD3DAvailable is returned before calling any QuickDraw 3D routines.

Note
For more information on the Gestalt function, see *Inside Macintosh: Operating System Utilities*. ◆

You create and initialize a connection to the QuickDraw 3D software by calling the Q3Initialize function, as illustrated in Listing 1-2.

Listing 1-2 Initializing a connection with QuickDraw 3D

```
OSErr MyInitialize (void)
{
    TQ3Status           myStatus;

    myStatus = Q3Initialize();              /*initialize QuickDraw 3D*/
```

```
    if (myStatus == kQ3Failure)
        DebugStr("\pQ3Initialize returned failure.");

    return (noErr);
}
```

Once you've successfully called `Q3Initialize`, you can safely call other QuickDraw 3D routines. If `Q3Initialize` returns unsuccessfully (as indicated by the `kQ3Failure` result code), you shouldn't call any QuickDraw 3D routines other than the error-reporting routines (such as `Q3Error_Get` or `Q3Error_IsFatalError`) or the `Q3IsInitialized` function. See the chapter "Error Manager" for details on QuickDraw 3D's error-handling capabilities.

When you have finished using QuickDraw 3D, you should call `Q3Exit` to close your connection with QuickDraw 3D. In most cases, you'll do this when terminating your application. Listing 1-3 illustrates how to call `Q3Exit`.

Listing 1-3 Terminating QuickDraw 3D

```
void MyFinishUp (void)
{
    TQ3Status           myStatus;

    myStatus = Q3Exit();                /*unload QuickDraw 3D*/
    if (myStatus == kQ3Failure)
        DebugStr("\pQ3Exit returned failure.");
}
```

Creating a Model

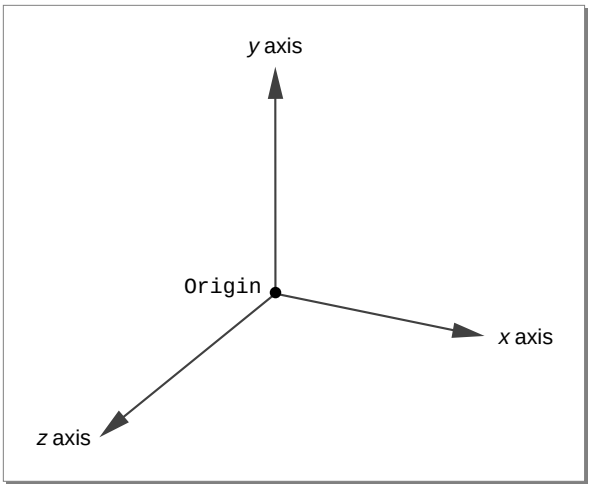
As explained in "Modeling and Rendering" (page 42), creating an image of a three-dimensional model involves several steps. You must first create a model and then specify key information about the scene (such as the lighting and camera angle). This section shows how to create a simple model containing three-dimensional objects.

Objects in QuickDraw 3D are defined using a **Cartesian coordinate system** that is **right-handed** (that is, if the thumb of the right hand points in the direction of the positive *x* axis and the index finger points in the direction of the positive *y* axis, then the middle finger, when made perpendicular to the other two fingers,

points in the direction of the positive *z* axis). Figure 1-5 shows a right-handed coordinate system.

Note
For a more complete description of the coordinate spaces used by QuickDraw 3D, see the chapter "Transform Objects" later in this book. ◆

Figure 1-5 A right-handed Cartesian coordinate system



The model created by the `MyNewModel` function defined in Listing 1-4 consists of a number of boxes that spell out the words "Hello World." The words are written in block letters, with each letter composed of a number of individual boxes. `MyNewModel` uses the inelegant but straightforward method of defining the 34 boxes by creating four arrays of 34 elements each. As described in the chapter "Geometric Objects", a box is defined by four pieces of information, an origin and three vectors that specify its sides:

```
typedef struct TQ3BoxData {
    TQ3Point3D                      origin;
    TQ3Vector3D                     orientation;
    TQ3Vector3D                     majorAxis;
```

```
    TQ3Vector3D                     minorAxis;
    TQ3AttributeSet                 *faceAttributeSet;
    TQ3AttributeSet                 boxAttributeSet;
} TQ3BoxData;
```

First, `MyNewModel` creates a new and empty ordered display group to contain all the boxes. Then the function loops through the data arrays, creating boxes and adding them to the group.

Listing 1-4 Creating a model

```
TQ3GroupObject MyNewModel (void)
{
    TQ3GroupObject                  myModel;
    TQ3GeometryObject               myBox;
    TQ3BoxData                      myBoxData;
    TQ3GroupPosition                myGroupPosition;

    /*Data for boxes comprising Hello and World block letters.*/
    long        i;
    float       xorigin[34] = {
                    -12.0, -9.0, -11.0, -7.0, -6.0, -6.0, -6.0, -2.0, -1.0,
                    3.0, 4.0, 8.0, 9.0, 9.0, 11.0, -13.0, -12.0, -11.0, -9.0,
                    -7.0, -6.0, -6.0, -4.0, -2.0, -1.0, -1.0,  1.0,  1.0,  3.0,
                    4.0, 8.0, 9.0, 9.0, 11.0};
    float       yorigin[34] = {
                    0.0, 0.0, 3.0, 0.0, 6.0, 3.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                    6.0, 0.0, 0.0, -8.0, -8.0, -7.0, -8.0, -8.0, -8.0, -2.0,
                    -8.0, -8.0, -2.0, -5.0, -4.0, -8.0, -8.0, -8.0, -8.0, -8.0,
                    -2.0, -7.0};
    float       height[34]  = {
                    7.0, 7.0, 1.0, 7.0, 1.0, 1.0, 1.0, 7.0, 1.0, 7.0, 1.0, 7.0,
                    1.0, 1.0, 7.0, 7.0, 1.0, 3.0, 7.0, 7.0, 1.0, 1.0, 7.0, 7.0,
                    1.0, 1.0, 2.0, 3.0, 7.0, 1.0, 7.0, 1.0, 1.0, 5.0};
    float       width[34]   = {
                    1.0, 1.0, 2.0, 1.0, 3.0, 2.0, 3.0, 1.0, 3.0, 1.0, 3.0, 1.0,
                    2.0, 2.0, 1.0, 1.0, 3.0, 1.0, 1.0, 1.0, 2.0, 2.0, 1.0, 1.0,
                    2.0, 2.0, 1.0, 1.0, 1.0, 3.0, 1.0, 2.0, 2.0, 1.0};
```

```
    /*Create an ordered display group for the complete model.*/
    myModel = Q3OrderedDisplayGroup_New();
    if (myModel == NULL)
        goto bail;

    /*Add all the boxes to the model.*/
    myBoxData.faceAttributeSet = NULL;
    myBoxData.boxAttributeSet = NULL;
    for (i=0; i<34; i++) {
        Q3Point3D_Set(&myBoxData.origin, xorigin[i], yorigin[i], 1.0);
        Q3Vector3D_Set(&myBoxData.orientation, 0, height[i], 0);
        Q3Vector3D_Set(&myBoxData.minorAxis, width[i], 0, 0);
        Q3Vector3D_Set(&myBoxData.majorAxis, 0, 0, 2);
        myBox = Q3Box_New(&myBoxData);
        myGroupPosition = Q3Group_AddObject(myModel, myBox);
        /*now that myBox has been added to group, dispose of our reference*/
        Q3Object_Dispose(myBox);
        if (myGroupPosition == NULL)
            goto bail;
    }

    return (myModel);                   /*return the completed model*/

bail:
    /*If any of the above failed, then return an empty model.*/
    return (NULL);
}
```

Note
The `MyNewModel` function can leak memory. Your application should use a different error-recovery strategy than is used in Listing 1-4. ◆

If successful, `MyNewModel` returns the group object containing the 34 boxes to its caller.

Configuring a Window

Usually, you'll want to display the two-dimensional image of a three-dimensional model in a window. To do this, it's useful to define a custom window information structure that holds all the information about the

QuickDraw 3D objects that are associated with the window. In the simplest cases, this information includes the model itself, the view, the illumination shading to be applied, and the desired styles of rendering the model. You might define a window information structure like this:

```
struct WindowInfo {
    TQ3ViewObject               view;
    TQ3GroupObject              model;
    TQ3ShaderObject             illumination;
    TQ3StyleObject              interpolation;
    TQ3StyleObject              backfacing;
    TQ3StyleObject              fillstyle;
};
typedef struct WindowInfo WindowInfo, *WindowInfoPtr, **WindowInfoHandle;
```

A standard way to attach an application-defined data structure (such as the `WindowInfo` structure) to a window is to set a handle to that structure as the window's reference constant. This technique is used in Listing 1-5.

Note
For a more complete description of using a window's reference constant to maintain window-specific information, see the discussion of document records in *Inside Macintosh: Overview*. ◆

Listing 1-5 Creating a new window and attaching a window information structure

```
void MyNewWindow (void)
{
    WindowPtr                       myWindow;
    Rect                            myBounds = {42, 4, 442, 604};
    WindowInfoHandle                myWinfo;

    /*Create new window.*/
    myWindow = NewCWindow(0L, &myBounds, "\pWindow!", 1, documentProc,
                                    (WindowPtr) -1, true, 0L);
    if (myWindow == NULL)
        goto bail;
    SetPort(myWindow);
```

```
/*Create storage for the new window and attach it to window.*/
myWinfo = (WindowInfoHandle) NewHandle(sizeof(WindowInfo));
if (myWinfo == NULL)
    goto bail;
SetWRefCon(myWindow, (long) myWinfo);
HLock((Handle) myWinfo);

/*Create a new view.*/
(**myWinfo).view = MyNewView(myWindow);
if ((**myWinfo).view == NULL)
    goto bail;

/*Create model to display.*/
(**myWinfo).model = MyNewModel();           /*see Listing 1-4 (page 58)*/
if ((**myWinfo).model == NULL)
    goto bail;

/*Configure an illumination shader.*/
(**myWinfo).illumination = Q3PhongIllumination_New();
if ((**myWinfo).illumination == NULL)
    goto bail;

/*Configure the rendering styles.*/
(**myWinfo).interpolation =
                    Q3InterpolationStyle_New(kQ3InterpolationStyleNone);
if ((**myWinfo).interpolation == NULL)
    goto bail;
(**myWinfo).backfacing =
                Q3BackfacingStyle_New(kQ3BackfacingStyleRemoveBackfacing);
if ((**myWinfo).backfacing == NULL)
    goto bail;
(**myWinfo).fillstyle = Q3FillStyle_New(kQ3FillStyleFilled);
if ((**myWinfo).fillstyle == NULL)
    goto bail;
HUnlock((Handle) myWinfo);

return;
```

```
bail:
    /*If failed for any reason, then close the window.*/
    if (myWinfo != NULL)
        DisposeHandle((Handle) myWinfo);
    if (myWindow != NULL)
        DisposeWindow(myWindow);
}
```

The `MyNewWindow` function creates a new window and a new window information structure, attaches the structure to the window, and then fills out several fields of that structure. In particular, `MyNewWindow` creates a new illumination shader that implements a Phong illumination model. You need an illumination shader for a view's lights to have any effect. (See the chapter "Shader Objects" for complete information on the available illumination shaders.) Then `MyNewWindow` disables interpolation between vertices of faces, removes unseen backfaces of objects in the model, and sets the renderer to render filled faces on those objects. These settings are actually passed to the renderer by *submitting* the styles during rendering. See "Rendering a Model," beginning on page 69 for details.

Note
The `MyNewWindow` function can leak memory. Your application should use a different error-recovery strategy than is used in Listing 1-5. ◆

Creating Lights

When you use any renderer more powerful than the wireframe renderer, you'll want to create and configure a set of lights to provide illumination for the object in the model. As you've seen, QuickDraw 3D provides a number of types of lights, each of which can emit light of various colors and intensities. The function `MyNewLights` defined in Listing 1-6 creates a group of lights. It creates an ambient light, a point light, and a directional light. See the chapter "Light Objects" for more details on creating lights.

Listing 1-6 Creating a group of lights

```
TQ3GroupObject MyNewLights (void)
{
    TQ3GroupPosition                myGroupPosition;
    TQ3GroupObject                  myLightList;
    TQ3LightData                    myLightData;
    TQ3PointLightData               myPointLightData;
    TQ3DirectionalLightData         myDirLightData;
    TQ3LightObject                  myAmbientLight, myPointLight, myFillLight;
    TQ3Point3D                      pointLocation = { -10.0, 0.0, 10.0 };
    TQ3Vector3D                     fillDirection = { 10.0, 0.0, 10.0 };
    TQ3ColorRGB                     WhiteLight = { 1.0, 1.0, 1.0 };

    /*Set up light data for ambient light.*/
    myLightData.isOn = kQ3True;
    myLightData.brightness = .2;
    myLightData.color = WhiteLight;

    /*Create ambient light.*/
    myAmbientLight = Q3AmbientLight_New(&myLightData);
    if (myAmbientLight == NULL)
        goto bail;

    /*Create a point light.*/
    myLightData.brightness = 1.0;
    myPointLightData.lightData = myLightData;
    myPointLightData.castsShadows = kQ3False;
    myPointLightData.attenuation = kQ3AttenuationTypeLinear;
    myPointLightData.location = pointLocation;
    myPointLight = Q3PointLight_New(&myPointLightData);
    if (myPointLight == NULL)
        goto bail;

    /*Create a directional light for fill.*/
    myLightData.brightness = .2;
    myDirLightData.lightData = myLightData;
    myDirLightData.castsShadows = kQ3False;
    myDirLightData.direction = fillDirection;
    myFillLight = Q3DirectionalLight_New(&myDirLightData);
```

```
    if (myFillLight == NULL)
        goto bail;

    /*Create light group and add each of the lights to the group.*/
    myLightList = Q3LightGroup_New();
    if (myLightList == NULL)
        goto bail;
    myGroupPosition = Q3Group_AddObject(myLightList, myAmbientLight);
    Q3Object_Dispose(myAmbientLight);               /*balance the reference count*/
    if (myGroupPosition == 0)
        goto bail;
    myGroupPosition = Q3Group_AddObject(myLightList, myPointLight);
    Q3Object_Dispose(myPointLight);                 /*balance the reference count*/
    if (myGroupPosition == 0)
        goto bail;
    myGroupPosition = Q3Group_AddObject(myLightList, myFillLight);
    Q3Object_Dispose(myFillLight);                  /*balance the reference count*/
    if (myGroupPosition == 0)
        goto bail;

    return (myLightList);

bail:
    /*If any of the above failed, then return nothing!*/
    return (NULL);
}
```

The `MyNewLights` function is straightforward. It fills out the fields of the relevant data structures (`TQ3LightData`, `TQ3PointLightData`, and `TQ3DirectionalLightData`) and calls the appropriate functions to create new light objects using the information in those structures. If successful, it adds those light objects to a group of lights. The group of lights will be added to a view, as shown in the following section.

Note
The `MyNewLights` function can leak memory. ◆

Creating a Draw Context

A draw context contains information that is specific to a particular type of window system, such as the extent of the pane to draw into and the method of clearing the window. You need to create a draw context and add it to a view in order to render a model. Listing 1-7 illustrates how to create a draw context for drawing into Macintosh windows.

Listing 1-7 Creating a Macintosh draw context

```
TQ3DrawContextObject MyNewDrawContext (WindowPtr theWindow)
{
    TQ3DrawContextObject            myDrawContext;
    TQ3DrawContextData              myDrawContextData;
    TQ3MacDrawContextData           myMacDrawContextData;
    TQ3ColorARGB                    myClearColor;

    /*Set the background color.*/
    Q3ColorARGB_Set(&myClearColor, 1.0, 0.6, 0.9, 0.9);

    /*Fill in draw context data.*/
    myDrawContextData.clearImageMethod = kQ3ClearMethodWithColor;
    myDrawContextData.clearImageColor = myClearColor;
    myDrawContextData.paneState = kQ3False;
    myDrawContextData.maskState = kQ3False;
    myDrawContextData.doubleBufferState = kQ3True;

    /*Fill in Macintosh-specific draw context data.*/
    myMacDrawContextData.drawContextData = myDrawContextData;
    myMacDrawContextData.window = (CWindowPtr) theWindow;
    myMacDrawContextData.library = kQ3Mac2DLibraryNone;
    myMacDrawContextData.viewPort = NULL;
    myMacDrawContextData.grafPort = NULL;

    /*Create draw context.*/
    myDrawContext = Q3MacDrawContext_New(&myMacDrawContextData);

    return (myDrawContext);
}
```

Essentially, `MyNewDrawContext` just fills in the fields of a `TQ3MacDrawContextData` structure and calls `Q3MacDrawContext_New` to create a new Macintosh draw context.

Creating a Camera

The remaining step before you can create a view is to create a camera object. A camera object specifies a point of view and a method of projecting the three-dimensional model into two dimensions. Listing 1-8 illustrates how to create a camera. See the chapter "Camera Objects" for complete details on the routines called in `MyNewCamera`.

Listing 1-8 Creating a camera

```
TQ3CameraObject MyNewCamera (void)
{
    TQ3CameraObject                     myCamera;
    TQ3CameraData                       myCameraData;
    TQ3ViewAngleAspectCameraData        myViewAngleCameraData;
    TQ3Point3D                          cameraFrom = { 0.0, 0.0, 15.0 };
    TQ3Point3D                          cameraTo = { 0.0, 0.0, 0.0 };
    TQ3Vector3D                         cameraUp = { 0.0, 1.0, 0.0 };

    /*Fill in camera data.*/
    myCameraData.placement.cameraLocation = cameraFrom;
    myCameraData.placement.pointOfInterest = cameraTo;
    myCameraData.placement.upVector = cameraUp;
    myCameraData.range.hither = .1;
    myCameraData.range.yon = 15.0;
    myCameraData.viewPort.origin.x = -1.0;
    myCameraData.viewPort.origin.y = 1.0;
    myCameraData.viewPort.width = 2.0;
    myCameraData.viewPort.height = 2.0;

    myViewAngleCameraData.cameraData = myCameraData;
    myViewAngleCameraData.fov = Q3Math_DegreesToRadians(100.0);
    myViewAngleCameraData.aspectRatioXToY = 1;
```

```
    myCamera = Q3ViewAngleAspectCamera_New(&myViewAngleCameraData);

    /*Return a camera.*/
    return (myCamera);
}
```

Like before, the `MyNewCamera` function simply fills out the fields of the appropriate data structures and calls the `Q3ViewAngleAspectCamera_New` function to create a new camera object.

IMPORTANT

All angles in QuickDraw 3D are specified in radians. You can use the `Q3Math_DegreesToRadians` macro to convert degrees to radians. This is illustrated in Listing 1-8, which sets the `fov` field to 100 degrees. ▲

Creating a View

A view is a collection of a model, a group of lights, a camera, a renderer, and a draw context. Now that you've defined functions that create all the requisite parts of a view (except the renderer), you can create a view, as illustrated in Listing 1-9. To do this, you create a new empty view object and then explicitly add the parts to it.

IMPORTANT

To create an image in a window, a view must contain at least a camera, a renderer, and a draw context. ▲

Listing 1-9 Creating a view

```
TQ3ViewObject MyNewView (WindowPtr theWindow)
{
    TQ3Status                       myStatus;
    TQ3ViewObject                   myView;
    TQ3DrawContextObject            myDrawContext;
    TQ3RendererObject               myRenderer;
    TQ3CameraObject                 myCamera;
    TQ3GroupObject                  myLights;
```

```
myView = Q3View_New();
if (myView == NULL)
    goto bail;

/*Create and set draw context.*/
myDrawContext = MyNewDrawContext(theWindow);
if (myDrawContext == NULL)
    goto bail;
myStatus = Q3View_SetDrawContext(myView, myDrawContext);
Q3Object_Dispose(myDrawContext);
if (myStatus == kQ3Failure)
    goto bail;

/*Create and set renderer.*/
myRenderer = Q3Renderer_NewFromType(kQ3RendererTypeInteractive);
if (myRenderer == NULL)
    goto bail;
myStatus = Q3View_SetRenderer(myView, myRenderer);
Q3Object_Dispose(myRenderer);
if (myStatus == kQ3Failure)
    goto bail;

/*Create and set camera.*/
myCamera = MyNewCamera();
if (myCamera == NULL)
    goto bail;
myStatus = Q3View_SetCamera(myView, myCamera);
Q3Object_Dispose(myCamera);
if (myStatus == kQ3Failure)
    goto bail;

/*Create and set lights.*/
myLights = MyNewLights();
if (myLights == NULL)
    goto bail;
myStatus = Q3View_SetLightGroup(myView, myLights);
Q3Object_Dispose(myLights);
if (myStatus == kQ3Failure)
    goto bail;

return (myView);
```

```
bail:
    /*If any of the above failed, then don't return a view.*/
    return (NULL);
}
```

Rendering a Model

To render a model using a view, you call QuickDraw 3D functions that submit the various shape objects (for instance, geometric objects, groups of geometric objects, and styles) that you want to appear in the view. Because a model might be too complex to process in a single pass (and for other reasons as well), you should call the rendering routines in a **rendering loop.** A rendering loop begins with a call to the `Q3View_StartRendering` function and should end when a call to the `Q3View_EndRendering` function returns some value other than `kQ3ViewStatusRetraverse`. Within the body of the rendering loop, you should submit the shapes you want rendered. Listing 1-10 shows the general structure of a rendering loop.

Listing 1-10 A basic rendering loop

```
Q3View_StartRendering(myView);
do {
    /*Submit your shape objects here.*/
    Q3DisplayGroup_Submit(myGroup, myView);
} while (Q3View_EndRendering(myView) == kQ3ViewStatusRetraverse);
```

The `Q3View_EndRendering` function returns a **view status value** that indicates whether the renderer has finished processing the model. The available view status values are defined by these constants:

```
typedef enum {
    kQ3ViewStatusDone,
    kQ3ViewStatusRetraverse,
    kQ3ViewStatusError,
    kQ3ViewStatusCancelled
} TQ3ViewStatus;
```

Listing 1-11 illustrates how to render the model defined in Listing 1-4 (page 58), using the view created and configured in Listing 1-9 (page 67). The `MyDraw`

function defined in Listing 1-11 retrieves the window information structure attached to a window and uses the information in it to render the model.

Listing 1-11 Rendering a model

```
void MyDraw (WindowPtr theWindow)
{
    WindowInfoHandle                myWinfo;
    TQ3Status                       myStat;
    TQ3DrawContextObject            myDrawContext;
    TQ3ViewStatus                   myViewStatus;

    if (theWindow == NULL)
        return;

    myWinfo = (WindowInfoHandle) GetWRefCon(theWindow);
    HLock((Handle) myWinfo);

    /*Start rendering.*/
    myStat = Q3View_StartRendering((**myWinfo).view);
    if (myStat == kQ3Failure)
        goto bail;

    do {
        myStat = Q3Shader_Submit((**myWinfo).illumination, (**myWinfo).view);
        if (myStat == kQ3Failure)
            goto bail;
        myStat = Q3Style_Submit((**myWinfo).interpolation, (**myWinfo).view);
        if (myStat == kQ3Failure)
            goto bail;
        myStat = Q3Style_Submit((**myWinfo).backfacing, (**myWinfo).view);
        if (myStat == kQ3Failure)
            goto bail;
        myStat = Q3Style_Submit((**myWinfo).fillstyle, (**myWinfo).view);
        if (myStat == kQ3Failure)
            goto bail;
        myStat = Q3DisplayGroup_Submit((**myWinfo).model, (**myWinfo).view);
        if (myStat == kQ3Failure)
            goto bail;
```

```
        myViewStatus = Q3View_EndRendering((**myWinfo).view);
    } while (myViewStatus == kQ3ViewStatusRetraverse);

    HUnlock((Handle) myWinfo);
    return;

bail:
    HUnlock((Handle) myWinfo);
    SysBeep(50);
}
```

The rendering loop allows your application to work with any current and future renderers that require multiple passes through a model's data in order to provide features such as transparency and constructive solid geometry.

For complete information about rendering loops and other kinds of submitting loops, see the chapter "View Objects" in this book.

QuickDraw 3D Reference

This section describes the basic constants and routines provided by QuickDraw 3D. See the section "QuickDraw 3D Errors, Warnings, and Notices," beginning on page 87 for a list of error, warning, and notice messages defined by QuickDraw 3D.

Constants

This section describes the basic constants provided by QuickDraw 3D.

Gestalt Selectors and Response Values

You can pass the `gestaltQD3D` selector to the `Gestalt` function to determine information about the availability of QuickDraw 3D.

```
enum {
    gestaltQD3D                                 = 'qd3d'
}
```

`Gestalt` returns information to you by returning a long word in the `response` parameter. Currently, the returned values are defined by constants:

```
enum {
    gestaltQD3DNotPresent                       = 0,
    gestaltQD3DAvailable                        = 1
}
```

Constant descriptions

`gestaltQD3DNotPresent`
QuickDraw 3D is not available.

`gestaltQD3DAvailable`
QuickDraw 3D is available.

You can pass the `gestaltQD3DVersion` selector to the `Gestalt` function to determine the installed version of QuickDraw 3D.

```
enum {
    gestaltQD3DVersion                          = 'q3v '
}
```

`Gestalt` returns version information in the `response` parameter.

Boolean Values

QuickDraw 3D defines Boolean values.

```
typedef enum TQ3Boolean {
    kQ3False,
    kQ3True
} TQ3Boolean;
```

Constant descriptions

`kQ3False` False.

`kQ3True` True.

Status Values

Most QuickDraw 3D routines return a status code, which is of type `TQ3Status`.

```
typedef enum TQ3Status {
    kQ3Failure,
    kQ3Success
} TQ3Status;
```

Constant descriptions

`kQ3Failure`	The routine failed.
`kQ3Success`	The routine succeeded.

Coordinate Axes

QuickDraw 3D provides constants for the three coordinate axes in a Cartesian coordinate system.

```
typedef enum TQ3Axis {
    kQ3AxisX,
    kQ3AxisY,
    kQ3AxisZ
} TQ3Axis;
```

Constant descriptions

`kQ3AxisX`	The *x* axis.
`kQ3AxisY`	The *y* axis.
`kQ3AxisZ`	The *z* axis.

QuickDraw 3D Routines

This section describes the routines you must call to initialize and terminate QuickDraw 3D. It also describes the routines you can use to create and manipulate sets, shapes, and strings.

Initializing and Terminating QuickDraw 3D

To use the services of QuickDraw 3D, you need to call `Q3Initialize` before calling any other QuickDraw 3D functions. When you are finished using QuickDraw 3D services, you should call `Q3Exit`.

Q3Initialize

You should call the `Q3Initialize` function to initialize a connection to QuickDraw 3D.

```
TQ3Status Q3Initialize (void);
```

DESCRIPTION

The `Q3Initialize` function initializes a connection between your application and the QuickDraw 3D graphics library. QuickDraw 3D allocates whatever internal storage it needs to manage subsequent calls to QuickDraw 3D routines, and it initializes any subcomponents it needs to call. If `Q3Initialize` returns `kQ3Failure`, you should not call any QuickDraw 3D routines other than the `Q3IsInitialized` function or the error-reporting routines provided by the Error Manager. Calling `Q3Initialize` more than once results in a warning being posted but is otherwise acceptable.

SPECIAL CONSIDERATIONS

You must call `Q3Initialize` to create a connection to the QuickDraw 3D software before calling any other QuickDraw 3D routines.

ERRORS

```
kQ3ErrorAlreadyInitialized
kQ3ErrorNotInitialized
kQ3ErrorOutOfMemory
```

Q3Exit

You should call the `Q3Exit` function to close your application's connection to QuickDraw 3D.

```
TQ3Status Q3Exit (void);
```

DESCRIPTION

The `Q3Exit` function closes your application's connection to QuickDraw 3D and deallocates any memory used by that connection. You should call `Q3Exit` when your application is finished using QuickDraw 3D routines. After calling `Q3Exit`, you should not call any QuickDraw 3D routines other than `Q3Initialize`, `Q3IsInitialized`, or the error-reporting routines provided by the Error Manager. Calling `Q3Exit` more than once results in a warning being posted but is otherwise acceptable.

ERRORS

`kQ3ErrorMemoryLeak`

Q3IsInitialized

You can use the `Q3IsInitialized` function to determine whether your application has successfully initialized a connection to QuickDraw 3D.

```
TQ3Boolean Q3IsInitialized (void);
```

DESCRIPTION

The `Q3IsInitialized` function returns a Boolean value that indicates whether your application has successfully initialized a connection to the QuickDraw 3D shared library (`kQ3True`) or not (`kQ3False`).

Getting Version Information

QuickDraw 3D provides a routine that you can use to get the installed version of QuickDraw 3D.

Q3GetVersion

You can use the `Q3GetVersion` function to get the version of the installed QuickDraw 3D software.

```
TQ3Status Q3GetVersion (
                    unsigned long *majorRevision,
                    unsigned long *minorRevision);
```

`majorRevision` On exit, a major revision number.

`minorRevision` On exit, a minor revision number.

DESCRIPTION

The `Q3GetVersion` function returns, in the `majorRevision` and `minorRevision` parameters, the major and minor revision numbers of the QuickDraw 3D software currently installed. See the description of the `'vers'` resource in the book *Inside Macintosh: Macintosh Toolbox Essentials* for information about major and minor revision numbers.

ERRORS

`kQ3ErrorNotInitialized`

Managing Sets

A set object (or, more briefly, a set) is a collection of zero or more elements, each of which has both an element type and some associated element data. QuickDraw 3D provides routines that you can use to create a new set, get the type of a set, add elements to a set, get the data associated with an element in a set, loop through all the elements in a set, and perform other operations on sets.

In general, you'll use the routines described in this section to handle sets containing elements with custom element types. You should use other QuickDraw 3D routines to handle sets that consist solely of elements with predefined element types. For example, to create a set of vertex attributes, you can use the `Q3VertexAttributeSet_New` function (to create a new empty set of vertex attributes) and the `Q3AttributeSet_Add` function (to add elements to that set). See the chapter "Attribute Objects" for information on managing attribute sets. See the section "Defining Custom Elements" (page 177) for information on handling custom element types.

Q3Set_New

You can use the `Q3Set_New` function to create a new set.

```
TQ3SetObject Q3Set_New (void);
```

DESCRIPTION

The `Q3Set_New` function returns, as its function result, a new set object. The set is initially empty. If `Q3Set_New` cannot create a new set object, it returns `NULL`.

Q3Set_GetType

You can use the `Q3Set_GetType` function to get the type of a set.

```
TQ3ObjectType Q3Set_GetType (TQ3SetObject set);
```

`set` A set object.

DESCRIPTION

The `Q3Set_GetType` function returns, as its function result, the type of the set specified by the `set` parameter. The type of set currently supported by QuickDraw 3D is defined by the constant:

```
kQ3SetTypeAttribute
```

If the type of the set cannot be determined or is invalid, `Q3Set_GetType` returns the value `kQ3ObjectTypeInvalid`.

Q3Set_Add

You can use the `Q3Set_Add` function to add an element to a set.

```
TQ3Status Q3Set_Add (
                    TQ3SetObject set,
                    TQ3ElementType type,
                    const void *data);
```

`set` A set object.

`type` An element type.

`data` A pointer to the element's data.

DESCRIPTION

The `Q3Set_Add` function adds the element specified by the `type` and `data` parameters to the set specified by the `set` parameter. The set must already exist when you call `Q3Set_Add`. Note that the element data is copied into the set. Accordingly, you can reuse the `data` parameter once you have called `Q3Set_Add`.

If the specified element type is a custom element type, `Q3Set_Add` uses the custom type's `kQ3MethodTypeElementCopyAdd` or `kQ3MethodTypeElementCopyReplace` custom methods. See the chapter "QuickDraw 3D Objects" for complete information on custom element types.

Q3Set_Get

You can use the `Q3Set_Get` function to get the data associated with an element in a set.

```
TQ3Status Q3Set_Get (TQ3SetObject set, TQ3ElementType type, void *data);
```

`set` A set object.

`type` An element type.

`data` On entry, a pointer to a structure large enough to hold the data associated with elements of the specified type. On exit, a pointer to the data of the element having the specified type.

DESCRIPTION

The `Q3Set_Get` function returns, in the `data` parameter, the data currently associated with the element whose type is specified by the `type` parameter in the set specified by the `set` parameter. If no element of that type is in the set, `Q3Set_Get` returns `kQ3Failure`.

If you pass the value `NULL` in the `data` parameter, no data is copied back to your application. (Passing `NULL` might be useful simply to determine whether a set contains a specific type of element.)

If the specified element type is a custom element type, `Q3Set_Get` uses the custom type's `kQ3MethodTypeElementCopyGet` custom method. See the chapter "QuickDraw 3D Objects" for complete information on custom element types.

Q3Set_Contains

You can use the `Q3Set_Contains` function to determine whether a set contains an element of a particular type.

```
TQ3Boolean Q3Set_Contains (TQ3SetObject set, TQ3ElementType type);
```

`set` A set object.

`type` An element type.

DESCRIPTION

The `Q3Set_Contains` function returns, as its function result, a Boolean value that indicates whether the set specified by the `set` parameter contains (`kQ3True`) or does not contain (`kQ3False`) an element of the type specified by the `type` parameter.

Q3Set_GetNextElementType

You can use the `Q3Set_GetNextElementType` function to iterate through the elements in a set.

```
TQ3Status Q3Set_GetNextElementType (
                    TQ3SetObject set,
                    TQ3ElementType *type);
```

`set` A set object.

`type` On entry, an element type, or `kQ3ElementTypeNone` to get the first element type in the specified set. On exit, the element type that immediately follows the specified element type in the set, or `kQ3ElementTypeNone` if there are no more element types.

DESCRIPTION

The `Q3Set_GetNextElementType` function returns, in the `type` parameter, the type of the element that immediately follows the element having the type specified by the `type` parameter in the set specified by the `set` parameter. To get the type of the first element in the set, pass `kQ3ElementTypeNone` in the `type` parameter. `Q3Set_GetNextElementType` returns `kQ3ElementTypeNone` when it has reached the end of the list of elements.

Q3Set_Empty

You can use the `Q3Set_Empty` function to empty a set of all the elements it contains.

```
TQ3Status Q3Set_Empty (TQ3SetObject target);
```

`target` A set object.

DESCRIPTION

The `Q3Set_Empty` function removes all the elements currently in the set specified by the `target` parameter.

If the specified element type is a custom element type, `Q3Set_Empty` uses the custom type's `kQ3MethodTypeElementDelete` custom method. See the chapter "QuickDraw 3D Objects" for complete information on custom element types.

Q3Set_Clear

You can use the `Q3Set_Clear` function to remove an element of a certain type from a set.

```
TQ3Status Q3Set_Clear (TQ3SetObject set, TQ3ElementType type);
```

`set` A set object.

`type` An element type.

DESCRIPTION

The `Q3Set_Clear` function removes the element whose type is specified by the `type` parameter from the set specified by the `set` parameter.

If the specified element type is a custom element type, `Q3Set_Clear` uses the custom type's `kQ3MethodTypeElementDelete` custom method. See the chapter "QuickDraw 3D Objects" for complete information on custom element types.

Managing Shapes

QuickDraw 3D provides routines that you can use to manage shape objects (or shapes). A shape object is any object that affects how and where a renderer renders an object in a view.

QuickDraw 3D provides six shape management routines that are identical in implementation to set routines discussed earlier:

Shape routine	Set routine	See page
`Q3Shape_GetElement`	`Q3Set_Get`	78
`Q3Shape_AddElement`	`Q3Set_Add`	77
`Q3Shape_ContainsElement`	`Q3Set_Contains`	79
`Q3Shape_GetNextElementType`	`Q3Set_GetNextElementType`	79
`Q3Shape_EmptyElements`	`Q3Set_Empty`	80
`Q3Shape_ClearElement`	`Q3Set_Clear`	81

Other shape management routines are described below.

Q3Shape_GetType

You can use the `Q3Shape_GetType` function to get the type of a shape.

```
TQ3ObjectType Q3Shape_GetType (TQ3ShapeObject shape);
```

`shape` A shape object.

DESCRIPTION

The `Q3Shape_GetType` function returns, as its function result, the type of the shape specified by the `shape` parameter. The types of shapes currently supported by QuickDraw 3D are defined by these constants:

```
kQ3ShapeTypeCamera
kQ3ShapeTypeGeometry
kQ3ShapeTypeGroup
kQ3ShapeTypeLight
kQ3ShapeTypeShader
kQ3ShapeTypeStyle
kQ3ShapeTypeTransform
kQ3ShapeTypeUnknown
```

If the type of the shape cannot be determined or is invalid, `Q3Shape_GetType` returns the value `kQ3ObjectTypeInvalid`.

Q3Shape_GetSet

You can use the `Q3Shape_GetSet` function to get the set currently associated with a shape.

```
TQ3Status Q3Shape_GetSet (TQ3ShapeObject shape, TQ3SetObject *set);
```

`shape` A shape object.

`set` On exit, the set currently associated with the specified shape.

DESCRIPTION

The `Q3Shape_GetSet` function returns, in the `set` parameter, the set of elements currently associated with the shape object specified by the `shape` parameter.

Q3Shape_SetSet

You can use the `Q3Shape_SetSet` function to set the set associated with a shape.

```
TQ3Status Q3Shape_SetSet (TQ3ShapeObject shape, TQ3SetObject set);
```

`shape` A shape object.

`set` The desired set to be associated with the specified shape.

DESCRIPTION

The `Q3Shape_SetSet` function sets the set of elements to be associated with the shape object specified by the `shape` parameter to the set specified by the `set` parameter.

Managing Strings

QuickDraw 3D provides routines that you can use to manage string objects (or strings).

Q3String_GetType

You can use the `Q3String_GetType` function to get the type of a string.

```
TQ3ObjectType Q3String_GetType (TQ3StringObject stringObj);
```

`stringObj` A string object.

DESCRIPTION

The `Q3String_GetType` function returns, as its function result, the type of the string specified by the `stringObj` parameter. The type of string currently supported by QuickDraw 3D is defined by a constant:

```
kQ3StringTypeCString
```

If the type of the string cannot be determined or is invalid, `Q3String_GetType` returns the value `kQ3ObjectTypeInvalid`.

Q3CString_New

You can use the `Q3CString_New` function to create a new C string.

```
TQ3StringObject Q3CString_New (const char *string);
```

`string` A pointer to a null-terminated C string.

DESCRIPTION

The `Q3CString_New` function returns, as its function result, a new string object of type `kQ3StringTypeCString` using the sequence of characters pointed to by the `string` parameter. That sequence of characters should be a standard C string (that is, an array of characters terminated by the null character). The characters are copied into the new string object's private data, so you can dispose of the array pointed to by the `string` parameter if `Q3CString_New` returns successfully. If `Q3CString_New` cannot allocate memory for the string, it returns the value `NULL`.

Q3CString_GetLength

You can use the `Q3CString_GetLength` function to get the length of a C string object.

```
TQ3Status Q3CString_GetLength (
                    TQ3StringObject stringObj,
                    unsigned long *length);
```

`stringObj`	A C string object.
`length`	On exit, the length of the specified C string object.

DESCRIPTION

The `Q3CString_GetLength` function returns, in the `length` parameter, the number of characters in the data associated with the C string object specified by the `stringObj` parameter. The length returned does not include the null character that terminates a C string. You should use `Q3CString_GetLength` to get the length of only string objects of type `kQ3StringTypeCString`.

Q3CString_GetString

You can use the `Q3CString_GetString` function to get the character data of a C string object.

```
TQ3Status Q3CString_GetString (
                    TQ3StringObject stringObj,
                    char **string);
```

`stringObj`	A C string object.
`string`	On entry, the value `NULL`. On exit, a pointer to a copy of the character data associated with the specified C string object.

DESCRIPTION

The `Q3CString_GetString` function returns, through the `string` parameter, a pointer to a copy of the character data associated with the C string object specified by the `stringObj` parameter. The value of the `string` parameter must be `NULL` when you call `Q3CString_GetString`, because it allocates memory and overwrites the `string` parameter. For instance, the following sequence of calls will cause a memory leak:

```
myStatus = Q3CString_GetString(myStringObj, &myString);
myStatus = Q3CString_GetString(myStringObj, &myString);
```

After the second call to `Q3CString_GetString`, the memory allocated by the first call to `Q3CString_GetString` is leaked; you cannot deallocate that memory because you've lost its address. You must make certain to call `Q3CString_EmptyData` to release the memory allocated by `Q3CString_GetString` when you are finished using the string data, and always before calling `Q3CString_GetString` with the same string pointer. Here is an example:

```
myStatus = Q3CString_GetString(myStringObj, &myString);
myStatus = Q3CString_EmptyData(&myString);
myStatus = Q3CString_GetString(myStringObj, &myString);
```

If the value of the `string` parameter is not `NULL`, `Q3CString_GetString` generates a warning.

You should use `Q3CString_GetString` only with string objects of type `kQ3StringTypeCString`.

ERRORS AND WARNINGS

`kQ3WarningPossibleMemoryLeak`

Q3CString_SetString

You can use the `Q3CString_SetString` function to set the character data of a C string object.

```
TQ3Status Q3CString_SetString (
                    TQ3StringObject stringObj,
                    const char *string);
```

`stringObj` A C string object.

`string` On entry, a pointer a C string specifying the character data to be associated with the specified C string object.

DESCRIPTION

The `Q3CString_SetString` function sets the character data associated with the C string object specified by the `stringObj` parameter to the sequence of characters

pointed to by the `string` parameter. That sequence of characters should be a standard C string (that is, an array of characters terminated by the null character). The characters are copied into the specified string object's private data, so you can dispose of the array pointed to by the `string` parameter if `Q3CString_SetString` returns successfully.

You should use `Q3CString_SetString` only with string objects of type `kQ3StringTypeCString`.

Q3CString_EmptyData

You can use the `Q3CString_EmptyData` function to dispose of the memory allocated by a previous call to `Q3CString_GetString`.

```
TQ3Status Q3CString_EmptyData (char **string);
```

`string` On entry, a pointer to a copy of the character data returned by a previous call to `Q3CString_GetString`. On exit, the value `NULL`.

DESCRIPTION

The `Q3CString_EmptyData` function deallocates the memory pointed to by the `string` parameter. The value of the `string` parameter must have been returned by a previous call to the `Q3CString_GetString` function. If successful, `Q3CString_EmptyData` sets the value of the `string` parameter to `NULL`. Thus, you can alternate calls to `Q3CString_GetString` and `Q3CString_EmptyData` without explicitly setting the character pointer to `NULL`.

You should use `Q3CString_EmptyData` only with string objects of type `kQ3StringTypeCString`.

QuickDraw 3D Errors, Warnings, and Notices

The following is a list of general QuickDraw 3D errors, warnings, and notices. More specific errors are listed at the end of each chapter.

No problem

```
kQ3ErrorNone
kQ3WarningNone
kQ3NoticeNone
```

Fatal errors

```
kQ3ErrorInternalError
kQ3ErrorNoRecovery
kQ3ErrorLastFatalError
```

System errors and warnings

```
kQ3ErrorNotInitialized
kQ3ErrorAlreadyInitialized
kQ3ErrorUnimplemented
kQ3ErrorRegistrationFailed
kQ3WarningInternalException
kQ3NoticeSystemAlreadyInitialized
```

OS errors

```
kQ3ErrorUnixError
kQ3ErrorMacintoshError
kQ3ErrorX11Error
kQ3ErrorWin32Error
```

Memory errors and warnings

```
kQ3ErrorMemoryLeak
kQ3ErrorOutOfMemory
kQ3WarningLowMemory
kQ3WarningPossibleMemoryLeak
```

Parameter errors, warnings, and notices

```
kQ3ErrorNULLParameter
kQ3ErrorParameterOutOfRange
kQ3ErrorInvalidParameter
kQ3ErrorInvalidData
```

```
kQ3ErrorAcceleratorAlreadySet
kQ3ErrorVector3DNotUnitLength
kQ3ErrorVector3DZeroLength
kQ3ErrorBadStringType
kQ3WarningParameterOutOfRange
kQ3NoticeDataAlreadyEmpty
kQ3NoticeParameterOutOfRange
```

Extension errors and warnings

```
kQ3ErrorNoExtensionsFolder
kQ3ErrorExtensionError
kQ3ErrorPrivateExtensionError
kQ3WarningExtensionNotLoading
```

Submit loop errors
(If you get one of these loop errors, check the previous error posted. If it is `kQ3ErrorOutOfMemory`, you may be able to recover by freeing up some memory and trying again.)

```
kQ3ErrorPickingLoopFailed
kQ3ErrorRenderingLoopFailed
kQ3ErrorWritingLoopFailed
kQ3ErrorBoundingLoopFailed
```

CHAPTER 1

Introduction to QuickDraw 3D

CHAPTER 2

3D Viewer

This chapter describes the 3D Viewer, which provides a high-level interface for displaying 3D objects and other data in a window and allowing users limited interaction with those objects. You can use the functions described here to present 3D data (stored either in a file or in memory) to users quickly and easily. The 3D Viewer provides controls with which the user can manipulate several aspects of the displayed data, such as the point of view.

The 3D Viewer allows you to display 3D data from metafiles (or memory) with minimal programming effort. It is analogous to the movie controller provided with QuickTime, which lets you display and control movies with little custom programming. You must specify at least one geometric object to the 3D Viewer, but it can supply default objects for other parts of the 3D environment such as the camera, lights, and renderer.

To use this chapter, you should already be familiar with the basic capabilities of QuickDraw 3D, as described in the first sections of the chapter "Introduction to QuickDraw 3D" elsewhere in this document.

IMPORTANT

If your application needs more advanced rendering or interaction capabilities, or if you want to allow users to create and manipulate objects dynamically, you can use the lower-level QuickDraw 3D application programming interfaces instead of, or in addition to, the higher-level 3D Viewer programming interfaces. ▲

The 3D Viewer supports the same platforms as the QuickDraw 3D library. Two versions of the 3D Viewer library are available: one to support the PowerPC version of the Mac OS and another to support the Win32 API (running on either Windows 95 or the Intel processor version of Windows NT 3.51 and later). The two different versions of the viewer have similar programming interfaces and can be used similarly.

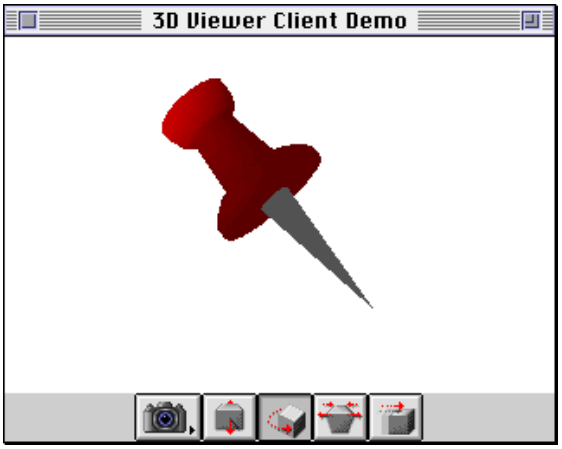
Note, however, that the two different versions of the 3D Viewer are intimately tied to the target platform through dependencies on each platform's base graphics libraries, their window management systems, and their event handling architectures. As a result, the programming interfaces are not identical; some functions and constants are unique to one platform or the other, and many of the functions take different parameters. Because of these necessary differences, the two libraries have different symbolic namespaces. The Mac OS version of the 3D Viewer uses function names beginning with `Q3Viewer` (for example, `Q3ViewerNew`), as in QuickDraw 3D version 1.0. The Win32 version of the 3D Viewer uses names beginning with `Q3WinViewer` (for example: `Q3WinViewerNew`).

This chapter has different sections discussing using the 3D Viewer on Mac OS and on Win32. The following Mac OS and Win32 reference sections describe the Mac OS and Win32 viewer routines respectively.

About the 3D Viewer

The **3D Viewer** (or, more briefly, the **Viewer**) is a shared library that provides a very simple method for displaying 3D models, together with a set of controls that permit limited interaction with those models. Figure 2-1 shows an instance of the 3D Viewer displaying a sample three-dimensional model.

Figure 2-1 An instance of the 3D Viewer displaying three-dimensional data



An instance of the 3D Viewer is a **viewer object.** Every viewer object is typically associated with exactly one window, within which the viewer object must be entirely contained. The viewer object can occupy the entire content region of the window, or it can occupy some smaller portion of the window. Your application can create more than one viewer object; indeed, it can create more than one viewer object associated with a single window.

When a viewer object is first created and displayed to the user, it consists of a **picture area** that contains the displayed image and either a controller strip or a badge. The **controller strip** is a rectangular area at the bottom of the viewer object that contains one or more controls. (See the following section for a complete explanation of these controls.) A **badge** is a visual element that is displayed in the picture area when the controller strip is not visible. The user can click on the badge to make the controller strip appear.

The part of the window that contains the picture area and the controller strip (if present) is the **viewer pane** (or **viewer frame**). In Figure 2-1, the viewer pane entirely fills the window's content region. Alternatively, you can place the viewer pane in part of the window; you would do this to embed a 3D picture in a document window.

It's important to understand that the 3D Viewer is built on top of QuickDraw 3D, but you don't need to call any QuickDraw 3D functions to use the 3D Viewer. The 3D Viewer is a shared library that is separate from the

QuickDraw 3D shared library. You can call `Q3ViewerNew` (and any other 3D Viewer functions) without having called `Q3Initialize` to initialize QuickDraw 3D. The models displayed by the Viewer must be structured according to the QuickDraw 3D Object Metafile specification, but the metafile data can be stored either in a file or in memory.

Controller Strips

The 3D Viewer provides control elements for manipulating the location and orientation of the user's point of view (that is, of the view's camera). Figure 2-2 shows a controller strip provided by the 3D Viewer.

Figure 2-2 The controller strip of the 3D Viewer

These controls are, from left to right:

- The **camera viewpoint control.** This control allows the user to view the model from a different camera viewpoint. Holding down the camera viewpoint control causes a pop-up menu to appear, listing the available predefined direction cameras as well as any perspective (that is, aspect ratio) cameras stored in the view hints of the 3DMF data. If any such cameras in the data have name attributes associated with them, the names are displayed in the menu. Otherwise the cameras are listed as "Camera #1," "Camera #2," and so forth. (The predefined direction cameras are calculated based on the front and top custom attributes if present in the 3DMF view hints. Otherwise, the predefined camera directions are calculated from the model's coordinate space.) You control whether this pop-up menu is displayed using the `kQ3ViewerButtonCamera` viewer flag.

Note
Only cameras of type `kQ3CameraTypeViewAngleAspect` are displayed in the camera viewpoint control's pop-up menu. ◆

- The **distance button.** This control allows the user to move closer to or farther away from the model. Clicking the distance button and then dragging the

cursor downward in the picture area causes the displayed object to move closer. Dragging the cursor upward in the picture area causes the displayed object to move farther away. The up and down arrow keys cause the object to move farther or closer, respectively. You control whether this button is displayed using the `kQ3ViewerButtonTruck` flag.

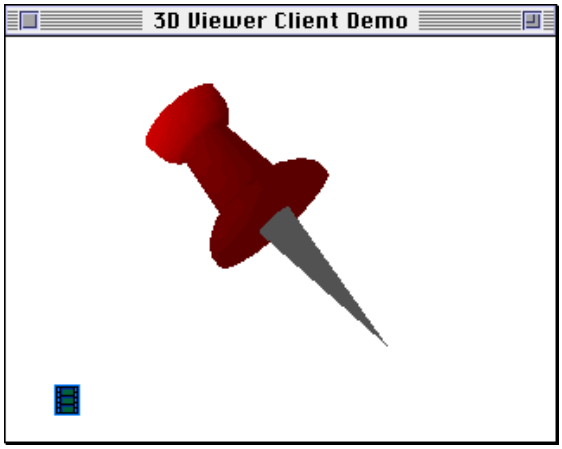
- The **rotate button.** This control allows the user to rotate an object. Clicking the rotate button and then dragging the cursor in the picture area causes the displayed object to rotate in the direction in which the cursor is dragged. The left and right arrow keys cause the object to rotate left and right. The up and down arrow keys cause the object to rotate up and down, respectively. You control whether this button is displayed using the `kQ3ViewerButtonOrbit` flag.
- The **zoom button.** This control allows the user to alter the field of view of the current camera, thereby zooming in or out on the object in the model. The up and down arrow keys cause the object to zoom out and in. This button is not displayed by default. You control whether this button is displayed using the `kQ3ViewerButtonZoom` flag
- The **move button.** This control allows the user to move an object. Clicking the move button and then dragging on the object in the picture area causes the object to be moved to a new location. The up, down, left, and right arrow keys cause the object to move up, down, left, or right, respectively. You control whether this button is displayed using the `kQ3ViewerButtonDolly` flag.
- The **reset button.** This pushbutton resets the camera viewpoint to its initial setting. You control whether this button is displayed using the `kQ3ViewerButtonReset` flag.

Your application controls which of these buttons are displayed in a viewer object's controller strip at the time you create the viewer object, or by appropriately setting a viewer's flags. See Listing 2-3 (page 101) for an example of setting a viewer's flags.

Badges

The 3D Viewer allows your application to distinguish 3D data from static graphics in documents by the use of a badge. Figure 2-3 shows a viewer pane with a badge.

Figure 2-3 A 3D model with a badge



The badge lets the user know that the image represents a 3D model rather than a static image. A badge appears when the viewer object is first displayed and the `kQ3ViewerShowBadge` flag is set in the object's viewer flags. When the user clicks the badge, the badge disappears and the standard controller strip appears.

Note
The badge control in unidirectional: it switches only from badge mode to controller strip mode. There is currently no user interface control to switch the viewer back to badge mode. If you want to switch from controller strip mode to badge mode (for instance, when a viewer object is deselected in a compound document), you must reset the viewer's flags and clear the controller strip. ◆

Your application can control whether the 3D Viewer displays a badge in a viewer pane by appropriately setting a viewer's flags. See "Viewer Flags" (page 105) for more information. Typically you won't want a viewer pane to support user interaction when the badge is displayed. To disable interaction, you must set the viewer to drag mode using the `kQ3ViewerDragMode` flag.

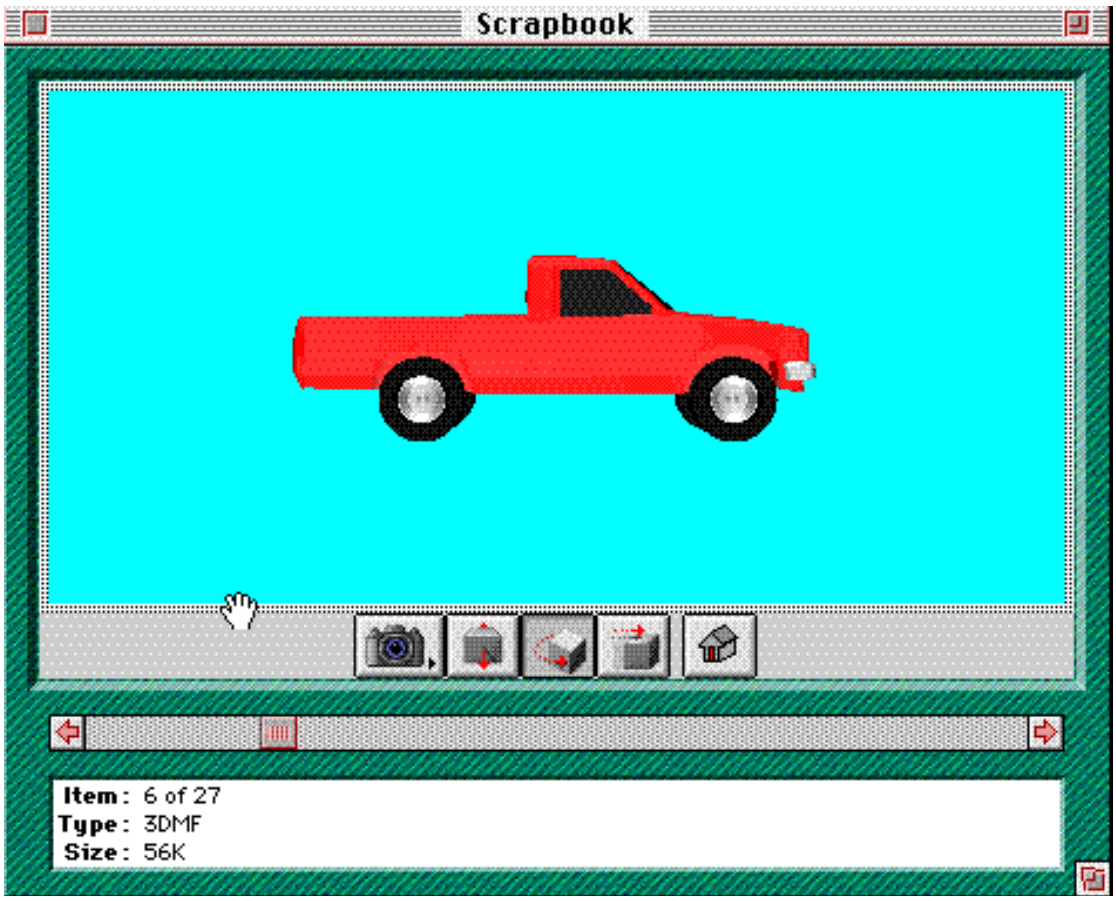
Drag and Drop

The 3D Viewer supports the Drag and Drop Manager to allow you to transfer 3DMF data between applications, the Clipboard, and the Scrapbook. The user typically initiates dragging from the viewer by dragging a special drag border that surrounds the perimeter of content area of the viewer (see Figure 2-4). The user can also always initiate a drag by holding down the Option key and dragging anywhere in the viewer content area.

Your application can also set the viewer to a special mode in which the only interaction supported is drag and drop. This mode must be explicitly set with the `kQ3ViewerDragMode` flag.

Note
When drag mode is set via `kQ3ViewerDragMode`, there is no visual indication that the viewer is in this mode (except that the cursor behaves appropriately), and none of the controls except the badge in the viewer are active. Therefore, it is only appropriate for an application to use the drag mode briefly or when the controller strip is not displayed (for example when the badge is displayed). ◆

Figure 2-4 A viewer object displaying the drag and drop border



The drag and drop functionality of the viewer is fully configurable through a set of flags using the `Q3ViewerSetFlags` function. Drag and drop can be disabled with the `kQ3ViewerDraggingOff` flag. Dragging out of and into the viewer can be independently disabled via the `kQ3ViewerDraggingOutOff` and `kQ3ViewerDraggingInOff` flags respectively. You can turn off the display of the drag border by clearing the `kQ3ViewerDrawDragBorder` flag.

Note
The version 1.1 release of the 3D Viewer for Win32 supports only dropping files onto the viewer window via the `WM_DROPFILES` mechanism. A Win32 application wanting to support full drag and drop with the 3D Viewer needs to use the OLE data transfer interfaces. ◆

Using the 3D Viewer

This section provides examples of how to use the 3D Viewer to display 3D data in a window.

Checking for the 3D Viewer

Before calling any 3D Viewer routines, you need to verify that the 3D Viewer software is available in the current operating environment. On the Macintosh Operating System, you can verify that the 3D Viewer is available by calling the `MyEnvironmentHas3DViewer` function defined in Listing 2-1.

Listing 2-1 Determining whether the 3D Viewer is available

```
long MyEnvironmentHas3DViewer (void)
{
    if (Q3ViewerNew != NULL)
        return TRUE
    else
        return FALSE;
}
```

The `MyEnvironmentHas3DViewer` function checks whether the address of the `Q3ViewerNew` function has been resolved. If it hasn't been resolved (that is, if the Code Fragment Manager couldn't find the 3D Viewer shared library when launching your application), `MyEnvironmentHas3DViewer` returns the value `FALSE` to its caller. Otherwise, if the address of the `Q3ViewerNew` function was successfully resolved, `MyEnvironmentHas3DViewer` returns `TRUE`.

Note
For the function `MyEnvironmentHas3DViewer` to work properly, you must establish soft links (also called *weak links*) between your application and the 3D Viewer shared library. For information on soft links, see the book *Inside Macintosh: PowerPC System Software.* For specific information on establishing soft links, see the documentation for your software development system. ◆

On the Macintosh Operating System, you can also verify that the 3D Viewer is available in the current operating environment by calling the `Gestalt` function with the `gestaltQuickDraw3DViewer` selector. `Gestalt` returns a long word whose value indicates the availability of the 3D Viewer. Currently these values are defined:

```
enum {
    gestaltQD3DViewer                       = 'q3vc',
    gestaltQD3DViewerNotPresent             = 0,
    gestaltQD3DViewerAvailable              = 1
}
```

You should ensure that the value `gestaltQD3DViewerAvailable` is returned before calling any 3D Viewer routines.

Note
For more information on the `Gestalt` function, see *Inside Macintosh: Operating System Utilities*. ◆

Checking the Version of the 3D Viewer

Some of the features described in this chapter are available only in versions 1.1 and later of the 3D Viewer. As a result, you might need to check the version of the 3D Viewer available in the current operating environment. Version 1.1 provides the `Q3ViewerGetVersion` function, which you can call to determine the version of the 3D Viewer. Because this function is not available in version 1.0, however, you must first determine that it is available before you can call it. Listing 2-2 defines a function, `MyGet3DViewerVersion`, that you can use to determine which version of the 3D Viewer is installed on a computer.

Listing 2-2 Determining the version of the 3D Viewer

```
void MyGet3DViewerVersion (unsigned long *major, unsigned long *minor)
{
    unsigned long       version;

    /*Version 1.0 did not have a get version call.*/
    if ((Boolean)Q3ViewerGetVersion == kUnresolvedSymbolAddress) {
        *major = 1;
        *minor = 0;
    } else {
        version = Q3ViewerGetVersion();
        *major = version >> 16;
        *minor = version & 0xFFFF;
    }
    return;
}
```

`MyGet3DViewerVersion` first checks to see whether the `Q3ViewerGetVersion` function is available. If it isn't, then version 1.0 must be running. Otherwise, `MyGet3DViewerVersion` calls `Q3ViewerGetVersion` to get the current version number.

Creating a Viewer

You can create a viewer object by calling the `Q3ViewerNew` function. You pass `Q3ViewerNew` a pointer to the window in which you want the viewer to appear, the rectangle that is to contain the viewer pane, and a selector indicating which viewer features to enable. `Q3ViewerNew` returns a reference to a viewer object. Listing 2-3 illustrates one way to call `Q3ViewerNew`. The function `MyCreateViewer` defined in Listing 2-3 creates a viewer pane that occupies the entire content region of the window whose address is passed to it as a parameter.

Listing 2-3 Creating a viewer object

```
TQ3ViewerObject MyCreateViewer (WindowPtr myWindow)
{
    TQ3ViewerObject         myViewer;
    Rect                    myRect;
```

```
    /*Get rectangle enclosing the window's content region.*/
    myRect = myWindow->portRect;
    if (EmptyRect(&myRect))             /*make sure we got a nonempty rect*/
        goto bail;

    /*Create a new viewer object in entire content region.*/
    myViewer = Q3ViewerNew((CGrafPtr)myWindow, &myRect, kQ3ViewerDefault);
    if (myViewer == NULL)
        goto bail;

    return(myViewer);                   /*return new viewer object*/

bail:
    /*If any of the above failed, return an empty viewer object.*/
    return(NULL);
}
```

The third parameter to the call to `Q3ViewerNew` is a set of **viewer flags** that specify information about the appearance and behavior of the new viewer object. In Listing 2-3, the viewer flag parameter is set to the value `kQ3ViewerDefault`, indicating that the default values of the viewer flags are to be used. See "Viewer Flags," beginning on page 105 for a complete description of the available viewer flags.

Attaching Data to a Viewer

You specify the 3D model to be displayed in a viewer pane's picture area by calling either the `Q3ViewerUseFile` or `Q3ViewerUseData` function. `Q3ViewerUseFile` takes a reference to an existing viewer object and a file reference number of an open metafile, as follows:

```
myErr = Q3ViewerUseFile(myViewer, myFsRefNum);
```

You use the `Q3ViewerUseData` function to specify a 3D model whose data is already in memory (either on the Clipboard or elsewhere in RAM). `Q3ViewerUseData` takes a reference to an existing viewer object, a pointer to the metafile data in RAM, and the number of bytes occupied by that data. Here's an example of calling `Q3ViewerUseData`:

```
myErr = Q3ViewerUseData(myViewer, myDataPtr, myDataSize);
```

IMPORTANT
The data in the buffer whose address and size you pass to `Q3ViewerUseData` must be in the QuickDraw 3D Object Metafile format. ▲

Once you attach the metafile data to a visible viewer object, the user is able to see the 3D model in the viewer pane. If, however, the viewer pane was invisible when it was created, you need to call the `Q3ViewerDraw` function to make it visible.

The 3D Viewer treats the model data as a single group. You can get a reference to the model data currently displayed in the viewer's picture area by calling the `Q3ViewerGetGroup` function. You can change that model data by calling the `Q3ViewerUseGroup` function.

You can also retrieve the view object associated with a viewer object by calling the `Q3ViewerGetView` function. You can then modify some of the view settings, such as the lights or the camera. If you wish, you can also restore the view settings to their original values by calling the `Q3ViewerRestoreView` function.

In versions 1.1 and later, your application can also set the viewer to display one of several predefined points of view by calling the `Q3ViewerSetCameraByView` and `Q3ViewerSetCameraByNumber` functions.

Handling Viewer Events

The final thing you need to do to support the 3D Viewer is to modify your main event loop so that events in the viewer controller strip and in the viewer pane can be handled. You need to add a line like this to your event loop:

```
isViewerEvent = Q3ViewerEvent(myViewer, myEvent);
```

The `Q3ViewerEvent` function determines whether the event specified by the `myEvent` event record affects the specified viewer object. If so, `Q3ViewerEvent` handles the event and returns `TRUE` as it function result. Otherwise, `Q3ViewerEvent` returns `FALSE`.

Your application should also call the `Q3AdjustCursor` function during idle-time processing to ensure that the 3D Viewer has an opportunity to update the cursor. If your application calls `SetCursor` to change the cursor while the 3D Viewer is active, it needs to call the `Q3ViewerCursorChanged` function immediately after it calls `SetCursor`, to inform the 3D Viewer that the cursor has changed shape.

IMPORTANT

The functions `Q3AdjustCursor` and `Q3ViewerCursorChanged` are available only in versions 1.1 and later of the 3D Viewer. ▲

3D Viewer Reference

This section describes the constants and routines that you can use to create and manage instances of the 3D Viewer.

Constants

This section describes the constants you might need to use when creating and managing a viewer object.

Gestalt Selector and Response Values

You can pass the `gestaltQuickDraw3DViewer` selector to the `Gestalt` function to determine information about the availability of the 3D Viewer.

```
enum {
    gestaltQD3DViewer                           = 'q3vc'
}
```

Constant descriptions

`gestaltQD3DViewer` Return information about the 3D Viewer.

`Gestalt` returns information to you by returning a long word in the `response` parameter. Currently, the returned values are defined by constants:

```
enum {
    gestaltQD3DViewerNotPresent                 = 0,
    gestaltQD3DViewerAvailable                  = 1
}
```

Constant descriptions

`gestaltQD3DViewerNotPresent`
The 3D Viewer is not available.

`gestaltQD3DViewerAvailable`
The 3D Viewer is available.

Viewer Flags

When you create a new viewer object (by calling `Q3ViewerNew`), you need to specify a set of viewer flags that control various aspects of the new viewer object.

IMPORTANT
All flags with values greater than or equal to `kQ3ViewerButtonReset` were introduced in version 1.1 of the 3D Viewer. In addition, the value of the flag `kQ3ViewerDefault` is different in version 1.0 than in all later versions. ▲

```
enum {
    kQ3ViewerShowBadge                  = 1<<0,
    kQ3ViewerActive                     = 1<<1,
    kQ3ViewerControllerVisible          = 1<<2,
    kQ3ViewerDrawFrame                  = 1<<3,
    kQ3ViewerDraggingOff                = 1<<4,
    kQ3ViewerButtonCamera               = 1<<5,
    kQ3ViewerButtonTruck                = 1<<6,
    kQ3ViewerButtonOrbit                = 1<<7,
    kQ3ViewerButtonZoom                 = 1<<8,
    kQ3ViewerButtonDolly                = 1<<9,
    kQ3ViewerButtonReset                = 1<<10,
    kQ3ViewerOutputTextMode             = 1<<11,
    kQ3ViewerDragMode                   = 1<<12,
    kQ3ViewerDrawGrowBox                = 1<<13,
    kQ3ViewerDrawDragBorder             = 1<<14,
    kQ3ViewerDraggingInOff              = 1<<15,
    kQ3ViewerDraggingOutOff             = 1<<16,
    kQ3ViewerDefault                    = 1<<31
};
```

Constant descriptions

`kQ3ViewerShowBadge` If this flag is set, a badge is displayed in the viewer pane whenever the controller strip is not visible. See "Badges" (page 95) for complete details on when the badge appears and disappears. If this flag is clear, no badge is displayed. By default, this flag is clear.

`kQ3ViewerActive` If this flag is set, the viewer object is active. If this flag is clear, the viewer object is inactive and the controller strip, if displayed, is dimmed. By default, this flag is set.

`kQ3ViewerControllerVisible`
If this flag is set, the controller strip is visible. If this flag is clear, the controller strip is not visible. If the `kQ3ViewerShowBadge` flag is set, the controller strip should be made invisible by clearing this flag. By default, this flag is set.

`kQ3ViewerDrawFrame`
If this flag is set, a one-pixel frame is drawn within the viewer pane. If this flag is clear, no frame is drawn within the viewer pane. By default, this flag is clear.

`kQ3ViewerDraggingOff`
If this flag is set, drag and drop is turned off in the viewer pane (that is, both dragging out of the viewer pane and dragging into the viewer pane are disabled). You can also independently set the states for dragging out and dragging in, by using the `kQ3ViewerDraggingOutOff` and `kQ3ViewerDraggingInOff` flags. By default, this flag is clear.

`kQ3ViewerButtonCamera`
If this flag is set, the camera viewpoint control in the controller strip is displayed. By default, this flag is set.

`kQ3ViewerButtonTruck`
If this flag is set, the distance button in the controller strip is displayed. By default, this flag is set.

`kQ3ViewerButtonOrbit`
If this flag is set, the rotate button in the controller strip is displayed. By default, this flag is set.

`kQ3ViewerButtonZoom`
If this flag is set, the zoom button in the controller strip is displayed. By default, this flag is clear.

`kQ3ViewerButtonDolly`
: If this flag is set, the move button in the controller strip is displayed. By default, this flag is set.

`kQ3ViewerButtonReset`
: If this flag is set, the reset button in the controller strip is displayed. By default, this flag is set.

`kQ3ViewerOutputTextMode`
: If this flag is set, the `Q3ViewerWriteFile` function writes 3DMF files in text format (not in binary format). By default, this flag is clear.

`kQ3ViewerDragMode`
: If this flag is set, the viewer object is in drag and drop mode, where the viewer responds only to drag and drop interaction. By default, this flag is clear.

`kQ3ViewerDrawGrowBox`
: If this flag is set, the 3D Viewer draws a grow box in the lower-right corner of the viewer pane. By default, this flag is clear.

`kQ3ViewerDrawDragBorder`
: If this flag is set, the 3D Viewer draws a drag border around the perimeter of the viewer pane. When the user clicks on the border and drags, a drag operation is initiated. By default, this flag is set.

`kQ3ViewerDraggingInOff`
: If this flag is set, dragging into the viewer pane is disabled. By default, this flag is clear.

`kQ3ViewerDraggingOutOff`
: If this flag is set, dragging out of the viewer pane is disabled. By default, this flag is clear.

`kQ3ViewerDefault`
: The default configuration for a viewer object.

Note
Applications that were compiled using version 1.0 of the 3D Viewer and that specify the `kQ3ViewerDefault` value when creating a view object (or resetting a viewer's flags) will be configured using the default flags defined for version 1.0, regardless of the version of the 3D Viewer installed. You must recompile your application using the interface file and shared library for version 1.1 or later the 3D Viewer to receive the new default behavior. ◆

Viewer State Flags

The `Q3ViewerGetState` function returns a long integer that encodes information about the current state of a viewer object. Bits of the returned long integer are addressed using these **viewer state flags:**

IMPORTANT

All flags with values greater than or equal to `kQ3ViewerHasUndo` were introduced in version 1.1 of the 3D Viewer. ▲

```
enum {
    kQ3ViewerEmpty                      = 0,
    kQ3ViewerHasModel                   = 1<<0,
    kQ3ViewerHasUndo                    = 1<<1
};
```

Constant descriptions

`kQ3ViewerEmpty`	If this flag is set, there is no image currently displayed by the specified viewer object.
`kQ3ViewerHasModel`	If this flag is set, there is an image currently displayed by the specified viewer object.
`kQ3ViewerHasUndo`	If this flag is set, the viewer's camera viewpoint has been modified and can be undone. You can use this information to determine whether to enable the Undo menu item in the Edit menu. See the description of the `Q3ViewerGetUndoString` function page 133.

Camera View Commands

The `viewType` parameter to the `Q3SetCameraView` function page 124 is a **camera view command** that specifies how to change the current camera view. These commands set the viewer to a predefined camera view.

IMPORTANT

`Q3SetCameraView` and the associated camera view commands were introduced in version 1.1 of the 3D Viewer. ▲

```
typedef enum TQ3ViewerCameraView {
    kQ3ViewerCameraRestore,
    kQ3ViewerCameraFit,
    kQ3ViewerCameraFront,
    kQ3ViewerCameraBack,
    kQ3ViewerCameraLeft,
    kQ3ViewerCameraRight,
    kQ3ViewerCameraTop,
    kQ3ViewerCameraBottom
} TQ3ViewerCameraView;
```

Constant descriptions

`kQ3ViewerCameraRestore`
: Set the camera view to its original position. Calling `Q3SetCameraView` with this camera view command is the same as calling the `Q3ViewerRestoreView` function.

`kQ3ViewerCameraFit`
: Set the camera view so that the 3D model fits entirely within the content area of the viewer.

`kQ3ViewerCameraFront`
: Set the camera view to look at the front of the model.

`kQ3ViewerCameraBack`
: Set the camera view to look at the back of the model.

`kQ3ViewerCameraLeft`
: Set the camera view to look at the left side of the model.

`kQ3ViewerCameraRight`
: Set the camera view to look at the right side of the model.

`kQ3ViewerCameraTop`
: Set the camera view to look at the top of the model.

`kQ3ViewerCameraBottom`
: Set the camera view to look at the bottom of the model.

Note
The six final camera view commands set the camera to predefined positions based on the front and top attributes in the model only if they are present in the model's data. Otherwise, the camera positions are calculated from the model's coordinate space. ◆

3D Viewer Routines

This section describes the routines provided by the 3D Viewer. You can use these routines to

- create a new viewer object
- dispose of a viewer object
- attach a file or block of data to a viewer object
- handle editing operations associated with a viewer object

You don't need to use all of these routines in order to use the 3D Viewer. For a description of which routines are required, see "Using the 3D Viewer," beginning on page 99.

Note
Most Macintosh 3D Viewer routines have equivalent routines for the Windows environment. A few significant programming differences are noted in the routine descriptions below. ◆

Creating and Destroying Viewers

This section describes the routines you can use to create and destroy viewer objects. See "Creating a Viewer" (page 101) for complete source code examples that illustrate how to use these routines.

Q3ViewerNew

You can use the `Q3ViewerNew` function to create a new viewer object.

MAC OS VERSION

```
TQ3ViewerObject Q3ViewerNew (
                    CGrafPtr         port,
                    Rect             *rect,
                    unsigned long    flags);
```

WINDOWS VERSION

```
TQ3ViewerObject Q3WinViewerNew (
                    HWND            window,
                    const RECT      *rect,
                    unsigned long   flags );
```

PARAMETERS

`port`	A pointer to a color graphics port that specifies the window with which the new viewer is to be associated, or a pointer to an offscreen graphics world. You can also pass the value `NULL` in this parameter to create an empty viewer; you can associate a port with the empty viewer by calling the `Q3ViewerSetPort` function.
`window`	A window handle.
`rect`	The desired viewer pane for the new viewer object. This rectangle is specified in window coordinates, where the origin (0, 0) is the upper-left corner of the window and values increase to the right and down the window.
`flags`	A set of viewer flags.
return value	A viewer object.

DESCRIPTION

The `Q3ViewerNew` function returns, as its function result, a reference to a new viewer object that is to be drawn in the window specified by the `port` parameter, in the location specified by the `rect` parameter. The `flags` parameter specifies the desired set of viewer flags. See "Viewer Flags" (page 105) for information on the flags you can specify when calling `Q3ViewerNew`.

The `Q3ViewerNew` function calls the QuickDraw 3D function `Q3Initialize` if your application has not already called it.

The object returned by `Q3ViewerNew`, of type `TQ3ViewerObject`, is not a general QuickDraw 3D object. Accordingly, you cannot call QuickDraw 3D object management functions, such as `Q3Object_Dispose` or `Q3Object_Duplicate`, on it. The type `TQ3ViewerObject` is used as a parameter type in other viewer routines, to refer to a viewer object.

Q3ViewerDispose

You can use the `Q3ViewerDispose` function to dispose of a viewer object.

MAC OS VERSION

```
OSErr Q3ViewerDispose (TQ3ViewerObject theViewer);
```

WINDOWS VERSION

```
TQ3Status Q3WinViewerDispose (TQ3ViewerObject viewer );
```

PARAMETER

`theViewer` A viewer object.

DESCRIPTION

The `Q3ViewerDispose` function disposes of the viewer object specified by the `theViewer` parameter.

Attaching Data to a Viewer

This section describes the routines you can use to attach data to viewer objects.

Q3ViewerUseFile

You can use the `Q3ViewerUseFile` function to set the file containing the 3D model to be displayed in a viewer object.

MAC OS VERSION

```
OSErr Q3ViewerUseFile (
                    TQ3ViewerObject    theViewer,
                    long               refNum);
```

WINDOWS VERSION

```
TQ3Status Q3WinViewerUseFile (
                    TQ3ViewerObject    viewer,
                    HANDLE             fileHandle );
```

PARAMETERS

`theViewer`	A viewer object.
`refNum`	The file reference number of an open file.
`fileHandle`	A handle to an open file.

DESCRIPTION

The `Q3ViewerUseFile` function sets the 3D data file to be displayed in the viewer object specified by the `theViewer` parameter to the open file having the file reference number specified by the `refnum` parameter.

Q3ViewerUseData

You can use the `Q3ViewerUseData` function to set the memory-based data displayed in a viewer object.

MAC OS VERSION

```
OSErr Q3ViewerUseData (
                    TQ3ViewerObject    theViewer,
                    void               *data,
                    long               size);
```

WINDOWS VERSION

```
TQ3Status Q3WinViewerUseData (
                    TQ3ViewerObject    viewer,
                    void               *data,
                    unsigned long      size );
```

PARAMETERS

`theViewer`	A viewer object.
`data`	A pointer to the beginning of a block of data in memory.
`size`	The size, in bytes, of the specified block of data.

DESCRIPTION

The `Q3ViewerUseData` function sets the 3D data to be displayed in the viewer object specified by the `theViewer` parameter to the data block beginning at the address specified by the `data` parameter and having the size specified by the `size` parameter.

Drawing a Viewer and its Contents

This section describes the routines you can use to draw a viewer object and its contents.

Q3ViewerDraw

You can use the `Q3ViewerDraw` function to draw a viewer object.

MAC OS VERSION

```
OSErr Q3ViewerDraw (TQ3ViewerObject theViewer);
```

WINDOWS VERSION

```
TQ3Status Q3WinViewerDraw (TQ3ViewerObject viewer);
```

PARAMETER

`theViewer` A viewer object.

DESCRIPTION

The `Q3ViewerDraw` function draws the viewer object specified by the `theViewer` parameter. You need to call this function only if the viewer flags or other visible features of a viewer have changed. For example, to change a viewer's pane, you need to call `Q3ViewerSetBounds` followed by `Q3ViewerDraw`. Similarly, if the viewer flags of a new viewer object have the `kQ3ViewerActive` flag clear, then to make the viewer object active you need to set that flag by calling `Q3ViewerSetFlags` and then draw the viewer object by calling `Q3ViewerDraw`.

Q3ViewerDrawContent

You can use the `Q3ViewerDrawContent` function to draw the content region of a viewer object.

MAC OS VERSION

```
OSErr Q3ViewerDrawContent (TQ3ViewerObject theViewer);
```

WINDOWS VERSION

```
TQ3Status Q3WinViewerDrawContent (TQ3ViewerObject viewer);
```

PARAMETER

`theViewer` A viewer object.

DESCRIPTION

The `Q3ViewerDrawContent` function forces the 3D Viewer to rerender and redraw the 3D data displayed in the content region of the viewer object specified by the `theViewer` parameter. You should call `Q3ViewerDrawContent` only if you have directly modified the model associated with that viewer object using QuickDraw 3D functions. `Q3ViewerDrawContent` redraws only the content region of the viewer object and is preferable to calling `Q3ViewerDraw`, which also redraws the controller strip and other user interface elements.

SPECIAL CONSIDERATIONS

The `Q3ViewerDrawContent` function is available only in versions 1.1 and later of the 3D Viewer.

Q3ViewerDrawControlStrip

You can use the `Q3ViewerDrawControlStrip` function to draw the controller strip and other user interface elements of a viewer object.

MAC OS VERSION

```
OSErr Q3ViewerDrawControlStrip (TQ3ViewerObject theViewer);
```

WINDOWS VERSION

```
TQ3Status Q3WinViewerDrawControlStrip (TQ3ViewerObject viewer);
```

PARAMETER

`theViewer` A viewer object.

DESCRIPTION

The `Q3ViewerDrawControlStrip` function forces the 3D Viewer to redraw the controller strip and other user interface elements of the viewer object specified by the `theViewer` parameter. (The user interface elements of a viewer object are

its controller strip, its badge, and its drag border.) You might use `Q3ViewerDrawControlStrip` when you want to update the controller strip but do not want to rerender and redraw the model in the content region of the viewer object.

SPECIAL CONSIDERATIONS

The `Q3ViewerDrawControlStrip` function is available only in versions 1.1 and later of the 3D Viewer.

Q3ViewerSetDrawingCallbackMethod

You can use the `Q3ViewerSetDrawingCallbackMethod` function to set a drawing completion callback routine for a viewer object. This function has no equivalent in the Windows environment.

```
OSErr Q3ViewerSetDrawingCallbackMethod (
                    TQ3ViewerObject theViewer,
                    TQ3ViewerDrawingCallbackMethod callbackMethod,
                    const void *data);
```

`theViewer` A viewer object.

`callbackMethod`
A pointer to the drawing completion callback routine for the specified viewer object. See `TQ3ViewerDrawingCallbackMethod` on page 161 for a description of this routine.

`data` A pointer to an application-defined block of data. This pointer is passed to the callback routine when it is called.

DESCRIPTION

The `Q3ViewerSetDrawingCallbackMethod` function registers the function pointed to by the `callbackMethod` parameter as a drawing completion callback routine for the viewer object specified by the `theViewer` parameter. This callback routine is called each time the 3D Viewer completes a drawing operation requested by a call to `Q3ViewerDraw`, `Q3ViewerDrawContent`, or `Q3ViewerDrawControlStrip`.

You can use a callback routine to perform any operations that should follow a completed drawing operation. For instance, if a viewer is associated with an offscreen graphics world, you can use the drawing completion callback routine to copy the rendered image to its final destination.

SPECIAL CONSIDERATIONS

The `Q3ViewerSetDrawingCallbackMethod` function is available only in versions 1.1 and later of the 3D Viewer.

Managing Viewer Information and State

The 3D Viewer provides a number of functions that you can use to get and set information about a viewer object and to manage its states.

Q3ViewerGetVersion

You can use the `Q3ViewerGetVersion` function to get the version of the 3D Viewer.

MAC OS VERSION

```
OSErr Q3ViewerGetVersion (
                    unsigned long    *majorRevision,
                    unsigned long    *minorRevision);
```

WINDOWS VERSION

```
TQ3Status Q3WinViewerGetVersion(
                    unsigned long    *majorRevision,
                    unsigned long    *minorRevision);
```

PARAMETERS

`majorRevision` On entry, a pointer to an unsigned long integer. On exit, that long integer is set to the major revision number of the 3D Viewer.

`minorRevision` On entry, a pointer to an unsigned long integer. On exit, that long integer is set to the minor revision number of the 3D Viewer.

DESCRIPTION

The `Q3ViewerGetVersion` function returns, in the long integers pointed to by the `majorRevision` and `minorRevision` parameters, the major and minor revision numbers of the 3D Viewer installed in the current operating environment.

SPECIAL CONSIDERATIONS

The `Q3ViewerGetVersion` function is available only in versions 1.1 and later of the 3D Viewer.

Q3ViewerGetReleaseVersion

You can use the `Q3ViewerGetReleaseVersion` function to get the release version number of the 3D Viewer.

MAC OS VERSION

```
OSErr Q3ViewerGetReleaseVersion (unsigned long *releaseRevision);
```

WINDOWS VERSION

```
TQ3Status Q3WinViewerGetReleaseVersion (unsigned long *releaseRevision);
```

PARAMETERS

`releaseRevision`
On entry, a pointer to an unsigned long integer. On exit, that long integer is set to the release revision number of the 3D Viewer in `'vers'` format.

DESCRIPTION

The `Q3ViewerGetReleaseVersion` function returns, in the long integer pointed to by the `releaseRevision` parameter, the release version number of the 3D Viewer installed in the current operating environment.

Note
For release 1.5.1 the release version number is 0x01518000. ◆

Q3ViewerGetView

You can use the `Q3ViewerGetView` function to get the view object associated with a viewer object.

MAC OS VERSION

```
TQ3ViewObject Q3ViewerGetView (TQ3ViewerObject theViewer);
```

WINDOWS VERSION

```
TQ3ViewObject Q3WinViewerGetView (TQ3ViewerObject viewer);
```

PARAMETER

`theViewer` A viewer object.

DESCRIPTION

The `Q3ViewerGetView` function returns, as its function result, the view object currently associated with the viewer specified by the `theViewer` parameter.

Q3ViewerRestoreView

You can use the `Q3ViewerRestoreView` function to restore the camera associated with a viewer object.

MAC OS VERSION

```
OSErr Q3ViewerRestoreView (TQ3ViewerObject theViewer);
```

WINDOWS VERSION

```
TQ3Status Q3WinViewerRestoreView (TQ3ViewerObject viewer);
```

PARAMETER

`theViewer` A viewer object.

DESCRIPTION

The `Q3ViewerRestoreView` function restores the camera settings of the viewer specified by the `theViewer` parameter to the original camera specified in the associated view hints object. If there is no view hints object associated with the specified viewer, `Q3ViewerRestoreView` creates a new default camera.

Note
`Q3ViewerRestoreView` performs the same operations as the reset button in the controller strip. ◆

Q3ViewerGetCameraCount

You can use the `Q3ViewerGetCameraCount` function to determine how many camera objects are currently associated with a viewer object.

MAC OS VERSION

```
OSErr Q3ViewerGetCameraCount (
                    TQ3ViewerObject    theViewer,
                    unsigned long      *cnt);
```

WINDOWS VERSION

```
TQ3Status Q3WinViewerGetCameraCount (
                    TQ3ViewerObject    viewer,
                    unsigned long      *count );
```

PARAMETERS

`theViewer` A viewer object.

`cnt` On entry, a pointer to an unsigned long integer. On exit, that long integer is set to the number of camera objects associated with the specified viewer object.

DESCRIPTION

The `Q3ViewerGetCameraCount` function returns, in the unsigned long integer pointed to by the `cnt` parameter, the number of camera objects associated with the viewer object specified by the `theViewer` parameter. If there is no view hints object associated with that viewer object, then there are no camera associated with that viewer object and `Q3ViewerGetCameraCount` returns 0.

IMPORTANT

`Q3ViewerGetCameraCount` counts only cameras of type `kQ3CameraViewAngleAspect`. ▲

SPECIAL CONSIDERATIONS

The `Q3ViewerGetCameraCount` function is available only in versions 1.1 and later of the 3D Viewer.

Q3ViewerSetCameraByNumber

You can use the `Q3ViewerSetCameraByNumber` function to set a viewer's camera to a camera specified by its index in the list of the viewer's cameras.

MAC OS VERSION

```
OSErr Q3ViewerSetCameraByNumber (
                    TQ3ViewerObject    theViewer,
                    unsigned long      cameraNo);
```

WINDOWS VERSION

```
TQ3Status Q3WinViewerSetCameraNumber (
                    TQ3ViewerObject    viewer,
                    unsigned long      cameraNo );
```

PARAMETERS

`theViewer` A viewer object.

`cameraNo` The index of a camera in the list of cameras associated with the specified viewer object. This list is 1-based (that is, the first camera has the index 1). The value of this parameter must be less than or equal to the number returned by the `Q3ViewerGetCameraCount` function.

DESCRIPTION

The `Q3ViewerSetCameraByNumber` function sets the camera of the viewer object specified by the `theViewer` parameter to the camera whose index in the list of the viewer's cameras is `cameraNo`.

Note
`Q3ViewerSetCameraByNumber` performs the same operations as the camera viewpoint pop-up menu in the controller strip. ◆

SPECIAL CONSIDERATIONS

The `Q3ViewerSetCameraByNumber` function is available only in versions 1.1 and later of the 3D Viewer.

Q3ViewerSetCameraByView

You can use the `Q3ViewerSetCameraByView` function to set a camera to a predefined camera view.

MAC OS VERSION

```
OSErr Q3ViewerSetCameraByView (
                    TQ3ViewerObject        theViewer,
                    TQ3ViewerCameraView    viewType);
```

WINDOWS VERSION

```
TQ3Status Q3WinViewerSetCameraView (
                    TQ3ViewerObject        viewer,
                    TQ3ViewerCameraView    viewType );
```

PARAMETERS

`theViewer` A viewer object.

`viewType` A camera view command. See "Camera View Commands" (page 108) for a description of the available camera view commands.

DESCRIPTION

The `Q3ViewerSetCameraByView` function sets the camera of the viewer object specified by the `theViewer` parameter to the camera viewpoint specified by the `viewType` parameter. For instance, if the value of the `viewType` parameter is `kQ3ViewerCameraTop`, then `Q3ViewerSetCameraByView` sets the camera to a viewpoint that is directly above the model in the viewer object.

SPECIAL CONSIDERATIONS

The `Q3ViewerSetCameraByView` function is available only in versions 1.1 and later of the 3D Viewer.

Q3ViewerGetFlags

You can use the `Q3ViewerGetFlags` function to get the current viewer flags for a viewer object.

MAC OS VERSION

```
unsigned long Q3ViewerGetFlags (TQ3ViewerObject theViewer);
```

WINDOWS VERSION

```
unsigned long Q3WinViewerGetFlags (TQ3ViewerObject viewer);
```

PARAMETER

`theViewer` A viewer object.

DESCRIPTION

The `Q3ViewerGetFlags` function returns, as its function result, the current set of viewer flags for the viewer specified by the `theViewer` parameter.

Q3ViewerSetFlags

You can use the `Q3ViewerSetFlags` function to set the viewer flags for a viewer object.

MAC OS VERSION

```
OSErr Q3ViewerSetFlags (
                    TQ3ViewerObject    theViewer,
                    unsigned long      flags);
```

WINDOWS VERSION

```
TQ3Status Q3WinViewerSetFlags (
                    TQ3ViewerObject    viewer,
                    unsigned long      flags );
```

PARAMETERS

`theViewer` A viewer object.

`flags` A set of viewer flags. See "Viewer Flags" (page 105) for a description of the constants you can use to set or clear individual viewer flags.

DESCRIPTION

The `Q3ViewerSetFlags` function sets the viewer flags associated with the viewer object specified by the `theViewer` parameter to the values passed in the `flags` parameter.

IMPORTANT

Any changes to a viewer's flags will not be visible until you call `Q3ViewerDraw` with the specified viewer object. ▲

Q3ViewerGetBounds

You can use the `Q3ViewerGetBounds` function to get the rectangle that bounds a viewer's pane.

MAC OS VERSION

```
OSErr Q3ViewerGetBounds (
                    TQ3ViewerObject    theViewer,
                    Rect               *bounds);
```

WINDOWS VERSION

```
TQ3Status Q3WinViewerGetBounds (
                    TQ3ViewerObject    viewer,
                    RECT               *bounds );
```

PARAMETERS

`theViewer` A viewer object.

`bounds` On exit, the rectangle that bounds the pane currently associated with the specified viewer object.

DESCRIPTION

The `Q3ViewerGetBounds` function returns, through the `bounds` parameter, the rectangle that currently bounds the pane associated with the viewer object specified by the `bounds` parameter.

Q3ViewerSetBounds

You can use the `Q3ViewerSetBounds` function to set the rectangle that bounds a viewer's pane.

MAC OS VERSION

```
OSErr Q3ViewerSetBounds (
                    TQ3ViewerObject    theViewer,
                    Rect               *bounds);
```

WINDOWS VERSION

```
TQ3Status Q3WinViewerSetBounds (
                    TQ3ViewerObject    viewer,
                    RECT               *bounds );
```

PARAMETERS

`theViewer` A viewer object.

`bounds` The desired viewer pane for the specified viewer object. This rectangle is specified in window coordinates, where the origin (0, 0) is the upper-left corner of the window and values increase to the right and down the window.

DESCRIPTION

The `Q3ViewerSetBounds` function sets the bounds of the viewer pane of the viewer object specified by the `theViewer` parameter to the rectangle specified by the `bounds` parameter.

IMPORTANT

Any changes to a viewer's bounds will not be visible until you call `Q3ViewerDraw` with the specified viewer object. ▲

Q3ViewerGetMininumDimension

You can use the `Q3ViewerGetMininumDimension` function to get the sides of the smallest rectangle that can contain the controller strip of a viewer object.

MAC OS VERSION

```
OSErr Q3ViewerGetMininumDimension (
                    TQ3ViewerObject    theViewer,
                    unsigned long      *width,
                    unsigned long      *height);
```

WINDOWS VERSION

```
TQ3Status Q3WinViewerGetMinimumDimension (
                    TQ3ViewerObject    viewer,
                    unsigned long      *width,
                    unsigned long      *height );
```

PARAMETERS

`theViewer`	A viewer object.
`width`	On exit, the width of the minimum viewer pane required to contain the entire contents of the controller strip.
`height`	On exit, the height of the minimum viewer pane required to contain the entire contents of the controller strip.

DESCRIPTION

The `Q3ViewerGetMininumDimension` function returns, in the `width` and `height` parameters, the width and height of the minimum viewer pane required to contain the entire contents of the controller strip associated with the viewer object specified by the `theViewer` parameter. If your application allows the viewer pane to be resized, you should ensure that it is not sized smaller than the dimensions returned by `Q3ViewerGetMininumDimension`; otherwise, some of the buttons in the controller strip will be clipped.

SPECIAL CONSIDERATIONS

The `Q3ViewerGetMininumDimension` function is available only in versions 1.1 and later of the 3D Viewer.

Q3ViewerGetPort

You can use the `Q3ViewerGetPort` function to get the Macintosh graphics port associated with a viewer object. This function has no equivalent in the Windows environment.

```
CGrafPtr Q3ViewerGetPort (TQ3ViewerObject theViewer);
```

`theViewer` A viewer object.

DESCRIPTION

The `Q3ViewerGetPort` function returns, as its function result, a pointer to the port currently associated with the viewer object specified by the `theViewer` parameter. The returned pointer may be a a pointer to a color graphics port, a pointer to an offscreen graphics world, or the value `NULL`, indicating that no port is currently associated with the viewer object.

Q3ViewerSetPort

You can use the `Q3ViewerSetPort` function to set the graphics port associated with a viewer object. This function has no equivalent in the Windows environment.

```
OSErr Q3ViewerSetPort (TQ3ViewerObject theViewer, CGrafPtr port);
```

`theViewer` A viewer object.

`port` A pointer to a color graphics port that specifies the window with which the specified viewer is to be associated, or a pointer to an offscreen graphics world. You can also pass the value `NULL` in this parameter to indicate that port is to be associated with the viewer object.

DESCRIPTION

The `Q3ViewerSetPort` function sets the graphics port associated with the viewer object specified by the `theViewer` parameter to the port specified by the `port` parameter.

Q3ViewerGetGroup

You can use the `Q3ViewerGetGroup` function to get the group of objects currently associated with a viewer.

MAC OS VERSION

```
TQ3GroupObject Q3ViewerGetGroup (TQ3ViewerObject theViewer);
```

WINDOWS VERSION

```
TQ3GroupObject Q3WinViewerGetGroup (TQ3ViewerObject viewer);
```

PARAMETER

`theViewer` A viewer object.

DESCRIPTION

The `Q3ViewerGetGroup` function returns, as its function result, a reference to the group containing the objects currently associated with the viewer specified by the `theViewer` parameter. The reference count of that group is incremented. You should therefore dispose of the group when you have finished using it.

Q3ViewerUseGroup

You can use the `Q3ViewerUseGroup` function to set the group of objects associated with a viewer.

MAC OS VERSION

```
OSErr Q3ViewerUseGroup (
                    TQ3ViewerObject    theViewer,
                    TQ3GroupObject     group);
```

WINDOWS VERSION

```
TQ3Status Q3WinViewerUseGroup (
                    TQ3ViewerObject    viewer,
                    TQ3GroupObject     group );
```

PARAMETERS

`theViewer` A viewer object.

`group` A group.

DESCRIPTION

The `Q3ViewerUseGroup` function sets the group of objects associated with the viewer specified by the `theViewer` parameter to the group specified by the `group` parameter.

Q3ViewerGetState

You can use the `Q3ViewerGetState` function to get the current state of a viewer object.

MAC OS VERSION

```
unsigned long Q3ViewerGetState (TQ3ViewerObject theViewer);
```

WINDOWS VERSION

```
unsigned long Q3WinViewerGetState (TQ3ViewerObject viewer);
```

PARAMETER

`theViewer` A viewer object.

DESCRIPTION

The `Q3ViewerGetState` function returns a long integer that encodes information about the current state of the viewer object specified by the `theViewer` parameter. Bits of the returned long integer are addressed using these constants, which define the **viewer state flags:**

```
enum {
    kQ3ViewerEmpty                  = 0,
    kQ3ViewerHasModel               = 1<<0,
    kQ3ViewerHasUndo                = 1<<1
};
```

If `Q3ViewerGetState` returns the value `kQ3ViewerEmpty`, there is no image currently displayed by the specified viewer object. If `Q3ViewerGetState` returns the value `kQ3ViewerHasModel`, there is an image currently displayed by the specified viewer object. If `Q3ViewerGetState` returns the value `kQ3ViewerHasUndo`, the user has modified the camera state using the interactive controls. You can use this information to determine whether Edit menu commands such as Undo, Cut, Clear, and Copy should be enabled or disabled.

SEE ALSO

Use the `Q3ViewerGetUndoString` function to get a string that describes the most recent user operation that can be undone.

Q3ViewerGetUndoString

You can use the `Q3ViewerGetUndoString` function to get a string that describes the most recent user operation that can be undone.

MAC OS VERSION

```
Boolean Q3ViewerGetUndoString (
                    TQ3ViewerObject    theViewer,
                    char               *string,
                    unsigned long      *cnt);
```

WINDOWS VERSION

```
TQ3Boolean Q3WinViewerGetUndoString (
                    TQ3ViewerObject    viewer,
                    char               *string,
                    unsigned long      stringSize,
                    unsigned long      *actualSize );
```

PARAMETERS

`theViewer` A viewer object.

`string` On entry, a pointer to a buffer. On exit, the buffer is filled with a localized string that describes the most recent user operation in the specified viewer that can be undone. Note that this string does not contain the substring "Undo."

`cnt` On entry, a pointer to an unsigned long integer that specifies the size, in bytes, of the buffer pointed to by the `string` parameter. On exit, that long integer is set to the number of bytes actually copied into that buffer.

`stringSize` On entry, an unsigned long integer that specifies the size, in bytes, of the buffer pointed to by the `string` parameter.

`actualSize` On exit, a pointer to a long integer that is set to the number of bytes actually copied into the buffer.

DESCRIPTION

The `Q3ViewerGetUndoString` function returns, through the `string` parameter, a localized string that describes the most recent user action in the viewer specified by the `theViewer` parameter that can be undone.

`Q3ViewerGetUndoString` also returns, as its function result, a Boolean value that indicates whether you can call the `Q3ViewerUndo` function to perform the undo operation (`TRUE`) or not (`FALSE`).

Typically, you'll use the string returned by `Q3ViewerGetUndoString` to generate the text for the Undo menu item in the Edit menu. Note, however, that the string returned through the `string` parameter does not contain the substring "Undo." You should get the appropriate substring (perhaps from a resource) and conjoin it with the string returned by `Q3ViewerGetUndoString` to construct the menu item text.

SPECIAL CONSIDERATIONS

The `Q3ViewerGetUndoString` function is available only in versions 1.1 and later of the 3D Viewer.

SEE ALSO

Use the `Q3ViewerUndo` function to undo a user operation.

Q3ViewerGetPict

You can use the `Q3ViewerGetPict` function to get a Macintosh `'pict'` representation of the image currently displayed by a viewer object. This function has no equivalent in the Windows environment.

```
PicHandle Q3ViewerGetPict (TQ3ViewerObject theViewer);
```

`theViewer` A viewer object.

DESCRIPTION

The `Q3ViewerGetPict` function returns, as its function result, a handle to a Macintosh `'pict'` structure that contains a representation of the image currently displayed by the viewer object specified by the `theViewer` parameter. You should call `DisposeHandle` to dispose of the memory occupied by the `pict` when you're done using it.

Q3ViewerGetButtonRect

You can use the `Q3ViewerGetButtonRect` function to get the rectangle that encloses a viewer button.

MAC OS VERSION

```
OSErr Q3ViewerGetButtonRect (
                    TQ3ViewerObject    theViewer,
                    unsigned long      button,
                    Rect               *rect);
```

WINDOWS VERSION

```
TQ3Status Q3WinViewerGetButtonRect (
                    TQ3ViewerObject    viewer,
                    unsigned long      button,
                    RECT               *rectangle );
```

PARAMETERS

`theViewer` A viewer object.

`button` A button.

`rect` On exit, the rectangle that enclosed the specified button in the specified viewer.

DESCRIPTION

The `Q3ViewerGetButtonRect` function returns, in the `rect` parameter, the rectangle that encloses the button specified by the `button` parameter in the viewer object specified by the `theViewer` parameter. You can use these constants to specify the button whose rectangle you want returned:

```
kQ3ViewerButtonCamera
kQ3ViewerButtonTruck
kQ3ViewerButtonOrbit
```

```
kQ3ViewerButtonZoom
kQ3ViewerButtonDolly
kQ3ViewerButtonReset
```

Q3ViewerGetCurrentButton

You can use the `Q3ViewerGetCurrentButton` function to get the active button of a viewer.

MAC OS VERSION

```
unsigned long Q3ViewerGetCurrentButton (TQ3ViewerObject theViewer);
```

WINDOWS VERSION

```
unsigned long Q3WinViewerGetCurrentButton (TQ3ViewerObject viewer);
```

PARAMETER

`theViewer` A viewer object.

DESCRIPTION

The `Q3ViewerGetCurrentButton` function returns, as its function result, the active button of the viewer object specified by the `theViewer` parameter. `Q3ViewerGetCurrentButton` returns one of these constants:

```
kQ3ViewerButtonTruck
kQ3ViewerButtonOrbit
kQ3ViewerButtonZoom
kQ3ViewerButtonDolly
```

Q3ViewerSetCurrentButton

You can use the `Q3ViewerSetCurrentButton` function to set the active button of a viewer pane.

MAC OS VERSION

```
OSErr Q3ViewerSetCurrentButton (
                    TQ3ViewerObject    theViewer,
                    unsigned long      button);
```

WINDOWS VERSION

```
TQ3Status Q3WinViewerSetCurrentButton (
                    TQ3ViewerObject    viewer,
                    unsigned long      button );
```

PARAMETERS

`theViewer` A viewer object.

`button` A button.

DESCRIPTION

The `Q3ViewerSetCurrentButton` function sets the active button of the viewer object specified by the `theViewer` parameter to the button specified by the `button` parameter. You can use these constants to specify a button:

```
kQ3ViewerButtonTruck
kQ3ViewerButtonOrbit
kQ3ViewerButtonZoom
kQ3ViewerButtonDolly
```

Q3ViewerGetDimension

You can use the `Q3ViewerGetDimension` function to get the current dimensions of the model space in a viewer's view hints object.

MAC OS VERSION

```
OSErr Q3ViewerGetDimension (
                    TQ3ViewerObject    theViewer,
                    unsigned long      *width,
                    unsigned long      *height);
```

WINDOWS VERSION

```
TQ3Status Q3WinViewerGetDimension (
                    TQ3ViewerObject    viewer,
                    unsigned long      *width,
                    unsigned long      *height );
```

PARAMETERS

`theViewer`	A viewer object.
`width`	On exit, the width of the pane of the specified viewer.
`height`	On exit, the height of the pane of the specified viewer.

DESCRIPTION

The `Q3ViewerGetDimension` function returns, in the `width` and `height` parameters, the current width and height of the model space in the view hints object associated with the viewer object specified by the `theViewer` parameter. If there is no such view hints object, `Q3ViewerGetDimension` returns the width and height of the viewer pane.

Q3ViewerSetDimension

You can use the `Q3ViewerSetDimension` function to set the current dimensions of the model space in a viewer's view hints object.

MAC OS VERSION

```
OSErr Q3ViewerSetDimension (
                    TQ3ViewerObject    theViewer,
                    unsigned long      width,
                    unsigned long      height);
```

WINDOWS VERSION

```
TQ3Status Q3WinViewerSetDimension (
                    TQ3ViewerObject    viewer,
                    unsigned long      width,
                    unsigned long      height );
```

PARAMETERS

`theViewer`	A viewer object.
`width`	The desired width of the pane of the specified viewer.
`height`	The desired height of the pane of the specified viewer.

DESCRIPTION

The `Q3ViewerSetDimension` function sets the width and height of the model space in the view hints object associated with the viewer object specified by the `theViewer` parameter to the values specified by the `width` and `height` parameters.

Q3ViewerGetBackgroundColor

You can use the `Q3ViewerGetBackgroundColor` function to get the background color of a viewer.

MAC OS VERSION

```
OSErr Q3ViewerGetBackgroundColor (
                    TQ3ViewerObject    theViewer,
                    TQ3ColorARGB       *color);
```

WINDOWS VERSION

```
TQ3Status Q3WinViewerGetBackgroundColor (
                    TQ3ViewerObject    viewer,
                    TQ3ColorARGB       *color );
```

PARAMETERS

`theViewer` A viewer object.

`color` On exit, the current background color.

DESCRIPTION

The `Q3ViewerGetBackgroundColor` function returns, in the `color` parameter, the background color of the viewer specified by the `theViewer` parameter.

Q3ViewerSetBackgroundColor

You can use the `Q3ViewerSetBackgroundColor` function to set the background color of a viewer.

MAC OS VERSION

```
OSErr Q3ViewerSetBackgroundColor (
                    TQ3ViewerObject    theViewer,
                    TQ3ColorARGB       *color);
```

WINDOWS VERSION

```
TQ3Status Q3WinViewerSetBackgroundColor (
                    TQ3ViewerObject    viewer,
                    TQ3ColorARGB       *color );
```

PARAMETERS

`theViewer`	A viewer object.
`color`	The desired background color.

DESCRIPTION

The `Q3ViewerSetBackgroundColor` function sets the background color of the viewer specified by the `theViewer` parameter to the color specified by the `color` parameter.

Updating Viewer Data

The 3D Viewer provides routines that you can use to update the file or memory copy of the 3D data displayed in a viewer.

Q3ViewerWriteFile

You can use the `Q3ViewerWriteFile` function to update the file data being displayed in a viewer.

MAC OS VERSION

```
OSErr Q3ViewerWriteFile (
                    TQ3ViewerObject    theViewer,
                    long               refNum);
```

WINDOWS VERSION

```
TQ3Status Q3WinViewerWriteFile (
                    TQ3ViewerObject    viewer,
                    HANDLE             fileHandle);
```

PARAMETERS

`theViewer`	A viewer object.
`refnum`	The file reference number of an open file.
`fileHandle`	A handle to an open file.

DESCRIPTION

The `Q3ViewerWriteFile` function writes the 3D data currently associated with the viewer object specified by the `theViewer` parameter in 3DMF format to the file specified by the `refnum` parameter. If the `kQ3ViewerOutputTextMode` flag has been set, the 3DMF data is written out in text mode; otherwise, it is written out in binary mode.

Note
If the camera viewpoint has been modified, a new camera is added to the 3D metafile data's view hints. ◆

Q3ViewerWriteData

You can use the `Q3ViewerWriteData` function to update the memory data being displayed in a viewer.

MAC OS VERSION

```
unsigned long Q3ViewerWriteData (
                    TQ3ViewerObject    theViewer,
                    void               **data);
```

WINDOWS VERSION

```
TQ3Status Q3WinViewerWriteData (
                    TQ3ViewerObject    viewer,
                    void               *data,
                    unsigned long      dataSize,
                    unsigned long      *actualDataSize );
```

PARAMETERS

`theViewer` A viewer object.

`data` On exit, a pointer to the beginning of a block of 3DMF data that describes the model currently displayed in the specified viewer object. This block of memory is allocated by the 3D Viewer and is automatically disposed of when your application destroys its last viewer object.

`dataSize` On entry, an unsigned long integer that specifies the size, in bytes, of the buffer pointed to by the `data` parameter.

`actualDataSize`
On exit, a pointer to a long integer that is set to the number of bytes actually copied into the buffer.

DESCRIPTION

The `Q3ViewerWriteData` function allocates a block of memory large enough to hold a 3DMF description of the model currently displayed in the viewer object specified by the `theViewer` parameter, writes that description into the block of memory, and returns the address of that block of memory in the `data` parameter. The Mac OS version of `Q3ViewerWriteData` returns, as its function result, the size (in bytes) of that block of memory.

If the `kQ3ViewerOutputTextMode` flag has been set, the 3DMF data is written out in text mode; otherwise, it is written out in binary mode.

Note
If the camera viewpoint has been modified, a new camera is added to the 3DMF data's view hints. ◆

SPECIAL CONSIDERATIONS

The block of memory that contains the 3D data is allocated by the 3D Viewer and must not be disposed of by your application. You should copy the 3D data into your own storage if you will need to access it after all viewer objects created by your application have been destroyed.

Handling Viewer Events

Viewer objects support several routines for handling events that occur in a viewer pane. The 3D Viewer provides two different event handling models: closed-loop event handling and open-loop event handling. You should decide which model best fits your needs and use it exclusively.

Q3ViewerEvent

You can use the `Q3ViewerEvent` function to give the 3D Viewer an opportunity to handle Macintosh events involving a viewer object. This function has no equivalent in the Windows environment.

```
Boolean Q3ViewerEvent (
                    TQ3ViewerObject    theViewer,
                    EventRecord        *evt);
```

`theViewer` A viewer object.

`evt` An event record.

DESCRIPTION

The `Q3ViewerEvent` function returns, as its function result, a Boolean value that indicates whether the event specified by the `evt` parameter relates to the viewer object specified by the `theViewer` parameter and was successfully handled (`TRUE`) or whether that event either does not relate to that viewer object or could not be

handled by the 3D Viewer (FALSE). The evt parameter is a pointer to an event record, which you usually obtain by calling the Event Manager function WaitNextEvent.

Q3ViewerEvent can handle most of the events relating to a viewer object. For example, it handles all user events relating to the controller strip displayed with a viewer object. For information on how to handle editing commands in a viewer pane, see "Handling Edit Commands," beginning on page 153.

SPECIAL CONSIDERATIONS

You should call Q3ViewerEvent in your main event loop to give the 3D Viewer an opportunity to handle events in a window that relate to a viewer object.

The Q3ViewerEvent function implements a closed-loop event handling model and should therefore not be used in conjunction with functions that implement an open-loop event handling model (namely, Q3ViewerMouseDown, Q3ViewerMouseUp, Q3ViewerContinueTracking, and Q3ViewerHandleKeyEvent).

Q3ViewerMouseDown

You can use the Q3ViewerMouseDown function to notify the 3D Viewer that a mouse-down event has occurred.

MAC OS VERSION

```
Boolean Q3ViewerMouseDown (
                    TQ3ViewerObject    theViewer,
                    long               x,
                    long               y);
```

WINDOWS VERSION

```
BOOL Q3WinViewerMouseDown (
                    TQ3ViewerObject    viewer,
                    long               x,
                    long               y );
```

PARAMETERS

`theViewer`	A viewer object.
`x`	The horizontal position, in global coordinates, of the mouse at the time the mouse-down event occurred.
`y`	The vertical position, in global coordinates, of the mouse at the time the mouse-down event occurred.

DESCRIPTION

The `Q3ViewerMouseDown` function informs the 3D Viewer that a mouse-down event has occurred at the screen location specified by the `x` and `y` parameters in the pane associated with the viewer specified by the `theViewer` parameter. `Q3ViewerMouseDown` returns a Boolean value indicating whether the 3D Viewer handled the event (`TRUE`) or not (`FALSE`).

SPECIAL CONSIDERATIONS

The `Q3ViewerMouseDown` function is available only in versions 1.1 and later of the 3D Viewer.

The `Q3ViewerMouseDown` function implements an open-loop event handling model and should therefore not be used in conjunction with `Q3ViewerEvent`, which implements a closed-loop event handling model.

Q3ViewerMouseUp

You can use the `Q3ViewerMouseUp` function to notify the 3D Viewer that a mouse-up event has occurred.

MAC OS VERSION

```
Boolean Q3ViewerMouseUp (
                    TQ3ViewerObject    theViewer,
                    long               x,
                    long               y);
```

WINDOWS VERSION

```
BOOL Q3WinViewerMouseUp (
                    TQ3ViewerObject    viewer,
                    long               x,
                    long               y );
```

PARAMETERS

`theViewer`	A viewer object.
`x`	The horizontal position, in global coordinates, of the mouse at the time the mouse-up event occurred.
`y`	The vertical position, in global coordinates, of the mouse at the time the mouse-up event occurred.

DESCRIPTION

The `Q3ViewerMouseUp` function informs the 3D Viewer that a mouse-up event has occurred at the screen location specified by the `x` and `y` parameters in the pane associated with the viewer specified by the `theViewer` parameter. `Q3ViewerMouseUp` returns a Boolean value indicating whether the 3D Viewer handled the event (`TRUE`) or not (`FALSE`).

SPECIAL CONSIDERATIONS

The `Q3ViewerMouseUp` function is available only in versions 1.1 and later of the 3D Viewer.

The `Q3ViewerMouseUp` function implements an open-loop event handling model and should therefore not be used in conjunction with `Q3ViewerEvent`, which implements a closed-loop event handling model.

Q3ViewerContinueTracking

You can use the `Q3ViewerContinueTracking` function to notify the 3D Viewer that an event has occurred and the mouse is still down.

MAC OS VERSION

```
Boolean Q3ViewerContinueTracking (
                    TQ3ViewerObject    theViewer,
                    long               x,
                    long               y);
```

WINDOWS VERSION

```
BOOL Q3WinViewerContinueTracking (
                    TQ3ViewerObject    viewer,
                    long               x,
                    long               y );
```

PARAMETERS

`theViewer`	A viewer object.
`x`	The horizontal position, in global coordinates, of the mouse at the current time.
`y`	The vertical position, in global coordinates, of the mouse at the current time.

DESCRIPTION

The `Q3ViewerContinueTracking` function informs the 3D Viewer that an event has occurred at the screen location specified by the `x` and `y` parameters in the pane associated with the viewer specified by the `theViewer` parameter. `Q3ViewerContinueTracking` returns a Boolean value indicating whether the 3D Viewer handled the event (`TRUE`) or not (`FALSE`).

SPECIAL CONSIDERATIONS

The `Q3ViewerContinueTracking` function is available only in versions 1.1 and later of the 3D Viewer.

The `Q3ViewerContinueTracking` function implements an open-loop event handling model and should therefore not be used in conjunction with `Q3ViewerEvent`, which implements a closed-loop event handling model.

Q3ViewerHandleKeyEvent

You can use the `Q3ViewerHandleKeyEvent` function to give the 3D Viewer an opportunity to handle Macintosh keyboard events involving a viewer object. This function has no equivalent in the Windows environment.

```
Boolean Q3ViewerHandleKeyEvent (
                    TQ3ViewerObject theViewer,
                    EventRecord *evt);
```

`theViewer` A viewer object.

`evt` An event record.

DESCRIPTION

The `Q3ViewerHandleKeyEvent` function returns, as its function result, a Boolean value that indicates whether the keyboard event specified by the `evt` parameter relates to the viewer object specified by the `theViewer` parameter and was successfully handled (`TRUE`) or whether that event either does not relate to that viewer object or could not be handled by the 3D Viewer (`FALSE`). The `evt` parameter is a pointer to an event record, which you usually obtain by calling the Event Manager function `WaitNextEvent`. This event should be a key-up, key-down, or auto-key event.

SPECIAL CONSIDERATIONS

The `Q3ViewerHandleKeyEvent` function is available only in versions 1.1 and later of the 3D Viewer.

The `Q3ViewerHandleKeyEvent` function implements an open-loop event handling model and should therefore not be used in conjunction with `Q3ViewerEvent`, which implements a closed-loop event handling model.

Managing Cursors

The 3D Viewer provides routines that you can use to ensure that the cursor is properly synchronized with the state of a viewer object.

Q3ViewerAdjustCursor

You can use the `Q3ViewerAdjustCursor` function to allow the 3D Viewer to adjust the cursor when it is inside a viewer object.

MAC OS VERSION

```
Boolean Q3ViewerAdjustCursor (
                    TQ3ViewerObject    theViewer,
                    Point              *pt);
```

WINDOWS VERSION

```
TQ3Boolean Q3WinViewerAdjustCursor (
                    TQ3ViewerObject    viewer,
                    long               x,
                    long               y );
```

PARAMETERS

`theViewer`	A viewer object.
`pt`	The location of the cursor, in the local coordinates of the window that contains the specified viewer object.
`x`	The horizontal position of the cursor, in the local coordinates of the window that contains the specified viewer object.
`y`	The vertical position of the cursor, in the local coordinates of the window that contains the specified viewer object..

DESCRIPTION

The `Q3ViewerAdjustCursor` function adjusts the cursor to whatever shape is appropriate when the cursor is located at the point specified by the `pt` parameter inside the viewer object specified by the `theViewer` parameter. You should call `Q3ViewerAdjustCursor` in response to a mouse-moved event or during your application's idle-time processing. `Q3ViewerAdjustCursor` returns a

Boolean value that indicates whether the shape of the cursor was changed (`TRUE`) or not (`FALSE`).

Q3ViewerCursorChanged

You can use the `Q3ViewerCursorChanged` function to notify the 3D Viewer that you have changed the cursor.

MAC OS VERSION

```
OSErr Q3ViewerCursorChanged (TQ3ViewerObject theViewer);
```

WINDOWS VERSION

```
TQ3Status Q3WinViewerCursorChanged (TQ3ViewerObject viewer);
```

PARAMETER

`theViewer` A viewer object.

DESCRIPTION

The `Q3ViewerCursorChanged` function notifies the 3D Viewer that you have changed the cursor while the viewer object specified by the `theViewer` parameter is active. You should call `Q3ViewerCursorChanged` whenever you change the cursor by calling the `SetCursor` routine.

SPECIAL CONSIDERATIONS

The `Q3ViewerCursorChanged` function is available only in versions 1.1 and later of the 3D Viewer.

Handling Edit Commands

The 3D Viewer provides routines that you can use to handle editing commands that apply to a viewer object.

Q3ViewerCut

You can use the `Q3ViewerCut` function to handle the Cut editing command when applied to data selected in a viewer object.

MAC OS VERSION

```
OSErr Q3ViewerCut (TQ3ViewerObject theViewer);
```

WINDOWS VERSION

```
TQ3Status Q3WinViewerCut (TQ3ViewerObject viewer);
```

PARAMETER

`theViewer` A viewer object.

DESCRIPTION

The `Q3ViewerCut` function cuts the data currently selected in the viewer object specified by the `theViewer` parameter. The cut data is placed on the Clipboard. You should call `Q3ViewerCut` when the user chooses the Cut command in your application's Edit menu (or types the appropriate keyboard equivalent) and the selected data is inside a viewer pane.

Q3ViewerCopy

You can use the `Q3ViewerCopy` function to handle the Copy editing command when applied to data selected in a viewer object.

MAC OS VERSION

```
OSErr Q3ViewerCopy (TQ3ViewerObject theViewer);
```

WINDOWS VERSION

```
TQ3Status Q3WinViewerCopy (TQ3ViewerObject viewer);
```

PARAMETER

`theViewer` A viewer object.

DESCRIPTION

The `Q3ViewerCopy` function copies the data currently selected in the viewer object specified by the `theViewer` parameter. The data is copied onto the Clipboard. You should call `Q3ViewerCopy` when the user chooses the Copy command in your application's Edit menu (or types the appropriate keyboard equivalent) and the selected data is inside a viewer pane.

Q3ViewerPaste

You can use the `Q3ViewerPaste` function to handle the Paste editing command when applied to data previously cut or copied from a viewer object.

MAC OS VERSION

```
OSErr Q3ViewerPaste (TQ3ViewerObject theViewer);
```

WINDOWS VERSION

```
TQ3Status Q3WinViewerPaste (TQ3ViewerObject viewer);
```

PARAMETER

`theViewer` A viewer object.

DESCRIPTION

The `Q3ViewerPaste` function pastes 3D data from the Clipboard into the viewer object specified by the `theViewer` parameter. You should call `Q3ViewerPaste` when the user chooses the Paste command in your application's Edit menu (or types the appropriate keyboard equivalent) and the data on the Clipboard was placed there by a previous call to `Q3ViewerCut` or `Q3ViewerCopy`.

SEE ALSO

To determine whether the data on the Clipboard is 3D data or not, you can use the `Q3ViewerGetState` function (page 132).

Q3ViewerClear

You can use the `Q3ViewerClear` function to handle the Clear editing command when applied to data selected in a viewer object.

MAC OS VERSION

```
OSErr Q3ViewerClear (TQ3ViewerObject theViewer);
```

WINDOWS VERSION

```
TQ3Status Q3WinViewerClear (TQ3ViewerObject viewer);
```

PARAMETER

`theViewer` A viewer object.

DESCRIPTION

The `Q3ViewerClear` function clears the data currently selected in the viewer object specified by the `theViewer` parameter. No data is copied onto the Clipboard. You should call `Q3ViewerClear` when the user chooses the Clear command in your application's Edit menu (or types the appropriate keyboard equivalent) and the selected data is inside a viewer pane.

Q3ViewerUndo

You can use the `Q3ViewerUndo` function to handle the Undo command when applied to the most recent editing change to a viewer object.

MAC OS VERSION

```
OSErr Q3ViewerUndo (TQ3ViewerObject theViewer);
```

WINDOWS VERSION

```
TQ3Status Q3WinViewerUndo (TQ3ViewerObject viewer);
```

PARAMETER

`theViewer` A viewer object.

DESCRIPTION

The `Q3ViewerUndo` function undoes the most recent editing operation on the viewer object specified by the `theViewer` parameter. You should call `Q3ViewerUndo` when the user chooses the Undo command in your application's Edit menu (or types the appropriate keyboard equivalent).

SPECIAL CONSIDERATIONS

The `Q3ViewerUndo` function is available only in versions 1.1 and later of the 3D Viewer.

SEE ALSO

Use `Q3ViewerGetUndoString` page 133 to determine what string to display as part of the Undo item in the Edit menu. Use `Q3ViewerGetState` (page 132) to determine if the Undo command is currently available.

Windows-Specific API

This section describes 3D Viewer data structures, definitions, and routines that are used only in the Win32 Windows environment.

Window and Clipboard Definitions

The following defines the Win32 Windows class name, which may be passed as a parameter to `CreateWindow` or `CreateWindowEx`:

```
#define kQ3ViewerClassName "QD3DViewerWindow"
```

The following defines the Win32 clipboard type:

```
#define kQ3ViewerClipboardFormat "QuickDraw 3D Metafile"
```

WM_NOTIFY Data Structures

```
typedef struct TQ3ViewerDropFiles {
    NMHDR                   nmhdr;
    HANDLE                  hDrop;
} TQ3ViewerDropFiles;

typedef struct TQ3ViewerSetView {
    NMHDR                   nmhdr;
    TQ3ViewerCameraView     view;
} TQ3ViewerSetView;

typedef struct TQ3ViewerSetViewNumber {
    NMHDR                   nmhdr;
    unsigned long           number;
} TQ3ViewerSetViewNumber;
```

```
typedef struct TQ3ViewerButtonSet {
    NMHDR                   nmhdr;
    unsigned long           button;
} TQ3ViewerButtonSet;
```

WM_NOTIFY Definitions

```
#define Q3VNM_DROPFILES         0x5000
#define Q3VNM_CANUNDO           0x5001
#define Q3VNM_DRAWCOMPLETE      0x5002
#define Q3VNM_SETVIEW           0x5003
#define Q3VNM_SETVIEWNUMBER     0x5004
#define Q3VNM_BUTTONSET         0x5005
#define Q3VNM_BADGEHIT          0x5006
```

Functions

The functions described in this section are used only in the Win32 Windows environment.

Q3WinViewerGetWindow

You can use the `Q3WinViewerGetWindow` function to obtain the window for a Windows viewer.

```
HWND Q3WinViewerGetWindow (TQ3ViewerObject viewer);
```

`viewer` A viewer object.

DESCRIPTION

The `Q3WinViewerGetWindow` function returns a handle to the window for `viewer`.

Q3WinViewerSetWindow

You can use the `Q3WinViewerSetWindow` function to set the window for a Windows viewer.

```
TQ3Status Q3WinViewerSetWindow (
                    TQ3ViewerObject    viewer,
                    HWND               window );
```

`viewer` A viewer object.

`window` A window.

DESCRIPTION

The `Q3WinViewerSetWindow` function sets the window for the viewer object `viewer` to `window`.

Q3WinViewerGetViewer

You can use the `Q3WinViewerGetViewer` function to get the viewer for a given window.

```
TQ3ViewerObject Q3WinViewerGetViewer (HWND theWindow);
```

`theWindow` A window handle.

DESCRIPTION

The `Q3WinViewerGetViewer` function returns the viewer for the window designated by `theWindow`.

Q3WinViewerGetBitmap

You can use the `Q3WinViewerGetBitmap` function to obtain a bitmap of the contents of a Windows viewer's window.

```
HBITMAP Q3WinViewerGetBitmap (TQ3ViewerObject viewer);
```

`viewer` A viewer object.

DESCRIPTION

The `Q3WinViewerGetBitmap` function returns a bitmap of the contents of the window for `viewer`. The caller should dispose of the bitmap.

Q3WinViewerGetControlStrip

You can use the `Q3WinViewerGetControlStrip` function to obtain the control strip window for a given viewer.

```
HWND Q3WinViewerGetControlStrip (TQ3ViewerObject viewer);
```

`viewer` A viewer object.

DESCRIPTION

The `Q3WinViewerGetControlStrip` function returns a handle to the control strip window for the viewer designated by `viewer`.

Application-Defined Routine

This section describes a routine your application might need to define when using the 3D Viewer.

TQ3ViewerDrawingCallbackMethod

You can define a drawing completion callback routine to perform any necessary post-drawing operations.

MAC OS VERSION

```
typedef OSErr (*TQ3ViewerDrawingCallbackMethod) (
                    TQ3ViewerObject    theViewer,
                    const void         *data);
```

WINDOWS VERSION

```
typedef TQ3Status (*TQ3ViewerDrawingCallbackMethod) (
                    TQ3ViewerObject    theViewer,
                    const void         *data);
```

PARAMETERS

`theViewer` A viewer object

`data` A pointer to an application-defined block of data. Your application passes this pointer to the `Q3ViewerSetDrawingCallbackMethod` function when installing the drawing completion callback routine.

DESCRIPTION

Your drawing completion callback routine is called each time the 3D Viewer completes a drawing operation requested by a call to `Q3ViewerDraw`, `Q3ViewerDrawContent`, or `Q3ViewerDrawControlStrip` for the viewer object specified by the `theViewer` parameter.

You install a drawing completion callback routine by calling the `Q3ViewerSetDrawingCallbackMethod` function.

CHAPTER 2

3D Viewer

CHAPTER 3

QuickDraw 3D Objects

This chapter describes QuickDraw 3D objects, which occupy the root level of the QuickDraw 3D class hierarchy. It also describes shared objects and the basic functions you can use to manage QuickDraw 3D objects and shared objects and to define custom objects.

You should read this chapter for a basic understanding of the QuickDraw 3D class hierarchy. You should also read this chapter if you want to learn how to define custom objects, such as custom attributes.

This chapter begins by describing the QuickDraw 3D class hierarchy. The section "Using QuickDraw 3D Objects," beginning on page 175 provides source code examples illustrating how to determine the type of an object and how to define an object metahandler. The section "QuickDraw 3D Objects Reference," beginning on page 178 describes the most basic routines associated with the QuickDraw 3D class hierarchy. These routines allow you to manage objects and shared objects. The section "Creating Custom Object Subclasses," beginning on page 194 tells you how you can extend the QuickDraw 3D object hierarchy by creating your own classes and objects.

About QuickDraw 3D Objects

QuickDraw 3D is *object oriented* in the sense that many of QuickDraw 3D's capabilities (introduced in the previous sections) are accessed by creating and manipulating QuickDraw 3D objects. A **QuickDraw 3D object** is an instance of a **QuickDraw 3D class,** which defines a data structure and a behavior for objects in the class. The behavior of a QuickDraw 3D object is determined by the set of **methods** associated with the object's class. In other words, a QuickDraw 3D object is a set of data defining the specific characteristics of the object and a set of methods defining the behaviors of the object.

Note
Currently, only C language interfaces are available for creating and manipulating QuickDraw 3D objects. ◆

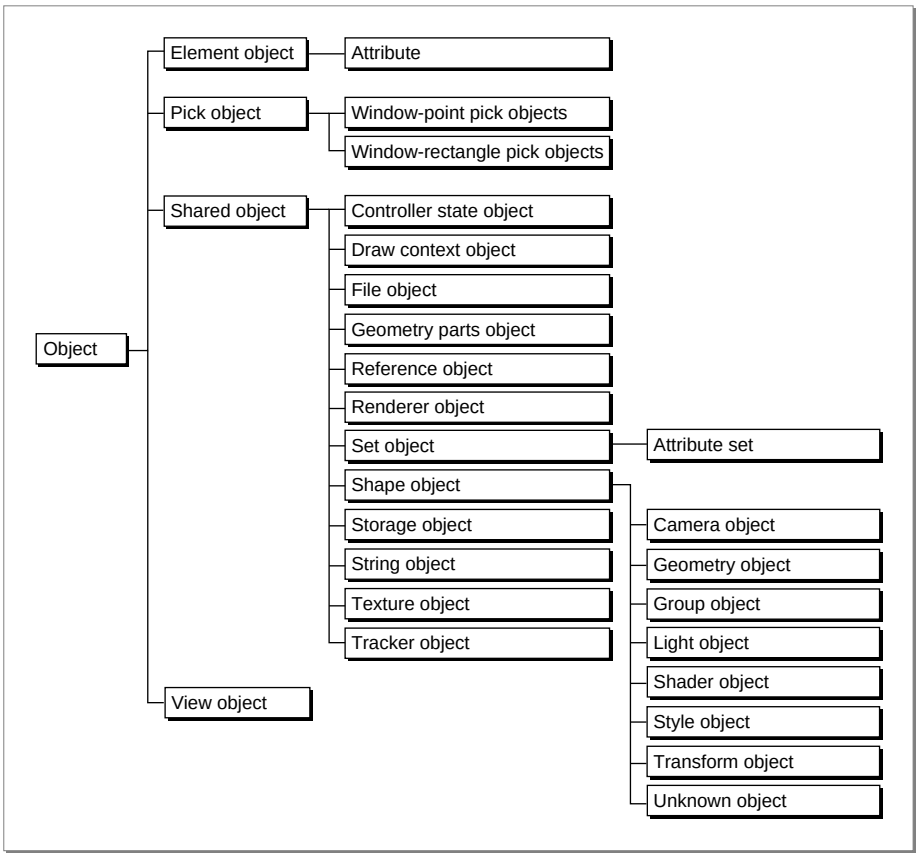
In keeping with QuickDraw 3D's object orientation, QuickDraw 3D objects are **opaque** (or **private**): the structure of the object's data and the implementation of the object's methods are not publicly defined. QuickDraw 3D provides routines that you can use to modify some of an object's private data or to have an object act upon itself using a class method.

The QuickDraw 3D Class Hierarchy

All QuickDraw 3D classes are arranged in the **QuickDraw 3D class hierarchy,** a hierarchical structure that provides for inheritance and overriding of class data and methods. Figure 3-1 illustrates the top levels of the QuickDraw 3D class hierarchy.

Note
Figure 3-1 does not show the entire QuickDraw 3D class hierarchy. ◆

Figure 3-1 The top levels of the QuickDraw 3D class hierarchy



Any particular class in the QuickDraw 3D class hierarchy can be a parent class, a child class, or both. A **parent class** is a class that is immediately above some other class in the class hierarchy. A **child class** is a class that has a parent. A child class that has no children is a **leaf class.**

A child class can either inherit or override the data and methods of its parent class. By default, a child class **inherits** data and methods from its parent (that is, the data and methods of the parent also apply to the child). Occasionally, the child class **overrides** the data or methods of its parent (that is, it defines data or methods to replace those of the parent class).

The following sections briefly describe the classes and subclasses of the QuickDraw 3D class hierarchy. You can find complete information on these classes in the remainder of this book.

QuickDraw 3D Objects

At the very top of the QuickDraw 3D class hierarchy is the common root of all QuickDraw 3D objects, the class `TQ3Object`.

```
typedef struct TQ3ObjectPrivate                    *TQ3Object;
```

The `TQ3Object` class provides methods for all its members, including dispose, duplicate, draw, and file I/O methods. For example, you dispose of any QuickDraw 3D object by calling the function `Q3Object_Dispose`. Similarly, you can duplicate any QuickDraw 3D object by calling `Q3Object_Duplicate`. It's important to understand that the methods defined at the root level of the QuickDraw 3D class hierarchy may be applied to any object in the class hierarchy, regardless of how far removed from the root level it may be. For instance, if the variable `mySpotLight` contains a reference to a spot light, then the code `Q3Object_Dispose(mySpotLight)` disposes of that light.

Note
Actually, using `Q3Object_Dispose` to dispose of a spot light simply reduces the light's *reference count* by 1. (This is because a light is a type of shared object.) The light is not disposed of until its reference count falls to 0. See "Reference Counts" (page 171) for complete details on reference counts. ◆

The methods defined for all QuickDraw 3D objects begin with the prefix `Q3Object`. Here are the root level methods defined for all objects:

```
Q3Object_Dispose
Q3Object_Duplicate
Q3Object_Submit
Q3Object_IsDrawable
Q3Object_GetType
Q3Object_GetLeafType
Q3Object_IsType
```

You'll use the `Q3Object_GetType`, `Q3Object_GetLeafType`, and `Q3Object_IsType` functions to determine the type or leaf type of an object. See "Determining the Type of a QuickDraw 3D Object" (page 175) for further information about object types and leaf types.

You'll use the `Q3Object_Submit` function to submit a QuickDraw 3D object for various operations. To **submit** an object is to make an object eligible for rendering, picking, writing, or bounding box or sphere calculation. Submission is always done in a loop, known as a **submitting loop.** For example, you submit an object for rendering by calling the `Q3Object_Submit` function inside of a submitting loop. See "Rendering a Model" (page 69) for complete information on submitting loops.

QuickDraw 3D Object Subclasses

There are four subclasses of the `TQ3Object` class: shared objects, element objects, view objects, and pick objects.

```
typedef TQ3Object          TQ3ElementObject;
typedef TQ3Object          TQ3PickObject;
typedef TQ3Object          TQ3SharedObject;
typedef TQ3Object          TQ3ViewObject;
```

An **element object** (or, more briefly, an **element**) is any QuickDraw 3D object that can be part of a set. Elements are not shared and hence have no reference count; they are always removed from memory whenever they are disposed of. Element objects are stored in sets (objects of type `TQ3SetObject`), which generally store such information as colors, positions, or application-defined data.

A **pick object** (or, more briefly, a **pick**) is a QuickDraw 3D object that is used to specify and return information related to picking (that is, selecting objects in a model that are close to a specified geometric object). In general, you'll use pick objects to retrieve data about objects selected by the user in a view.

A **shared object** is a QuickDraw 3D object that may be referenced by many objects or the application at the same time. For example, a particular renderer can be associated with several views. Similarly, a single pixmap can be used as a texture for several different objects in a model. The `TQ3SharedObject` class overrides the dispose method of the `TQ3Object` class by using a **reference count** to keep track of the number of times an object is being shared. When a shared object is referred to by some other object (for example, when a renderer is

associated with a view), the reference count is incremented, and whenever a shared object is disposed of, the reference count is decremented. A shared object is not removed from memory until its reference count falls to 0.

Note
For more information on reference counts, see "Reference Counts" (page 171). ◆

A **view object** (or more briefly, a **view**) is a type of QuickDraw 3D object used to collect state information that controls the appearance and position of objects at the time of rendering. A view binds together geometric objects in a model and other drawable QuickDraw 3D objects to produce a coherent image. A view is essentially a collection of a single camera, a (possibly empty) group of lights, a draw context, a renderer, styles, and attributes.

Shared Object Subclasses

There are many subclasses of the `TQ3SharedObject` class.

```
typedef TQ3SharedObject         TQ3AttachmentObject;
typedef TQ3SharedObject         TQ3ControllerStateObject;
typedef TQ3SharedObject         TQ3DrawContextObject;
typedef TQ3SharedObject         TQ3FileObject;
typedef TQ3SharedObject         TQ3RendererObject;
typedef TQ3SharedObject         TQ3SetObject;
typedef TQ3SharedObject         TQ3ShapeObject;
typedef TQ3SharedObject         TQ3ShapePartObject;
typedef TQ3SharedObject         TQ3StorageObject;
typedef TQ3SharedObject         TQ3StringObject;
typedef TQ3SharedObject         TQ3TextureObject;
typedef TQ3SharedObject         TQ3TrackerObject;
typedef TQ3SharedObject         TQ3ViewHintsObject;
```

Attachment objects and texture objects are used in the QuickDraw 3D shading architecture to provide shading in a model. See the chapter "Shader Objects" for information about these types of objects.

Controller state objects and tracker objects are used to support user interaction with the objects in a model. See the chapter "Pointing Device Manager" for complete information about these types of objects.

A **draw context object** (or more briefly, a **draw context**) is a QuickDraw 3D object that maintains information specific to a particular window system or drawing destination.

A **file object** (or, more briefly, a **file**) is used to access disk- or memory-based data stored in a container. A file object serves as the interface between the metafile and the storage object.

A **renderer object** (or, more briefly, a **renderer**) is used to render a model—that is, to create an image from a view and a model. A renderer controls various aspects of the model and the resulting image, such as the parts of objects that are drawn (for example, only the edges or filled faces).

A **set object** (or, more briefly, a **set**) is a collection of zero or more elements, each of which has both an element type and some associated element data. Sets may contain only one element of a given element type.

A **shape object** (or, more briefly, a **shape**) is a type of QuickDraw 3D object that affects what or how a renderer renders an object in a view. For example, a light is a shape object because it affects the illumination of the objects in a model. See "Shape Object Subclasses" (page 170) for a description of the available shapes.

A **shape part object** (or, more briefly, a **shape part**) is a distinguishable part of a shape. For example, a mesh (which is a geometric object and hence a shape object) can be distinguished into faces, edges, and vertices. When a user selects some part of a mesh, you can call shape part routines to determine what part of the mesh was selected. See the chapter "Pick Objects" for more information about shape parts and mesh parts.

A **storage object** represents any piece of storage in a computer (for example, a file on disk, an area of memory, or some data on the Clipboard).

A **string object** (or, more briefly, a **string**) is a QuickDraw 3D object that contains a sequence of characters. Strings can be referenced multiple times to maintain common descriptive information.

A **view hints object** (or, more briefly, a **view hint**) is a QuickDraw 3D object in a metafile that gives hints about how to render a scene. You can use that information to configure a view object, or you can choose to ignore it.

Set Object Subclasses

There is one subclass of the `TQ3SetObject` class, the attribute set.

```
typedef TQ3SetObject            TQ3AttributeSet;
```

Shape Object Subclasses

There are numerous subclasses of the `TQ3ShapeObject` class.

```
typedef TQ3ShapeObject          TQ3CameraObject;
typedef TQ3ShapeObject          TQ3GeometryObject;
typedef TQ3ShapeObject          TQ3GroupObject;
typedef TQ3ShapeObject          TQ3LightObject;
typedef TQ3ShapeObject          TQ3ReferenceObject;
typedef TQ3ShapeObject          TQ3ShaderObject;
typedef TQ3ShapeObject          TQ3StyleObject;
typedef TQ3ShapeObject          TQ3TransformObject;
typedef TQ3ShapeObject          TQ3UnknownObject;
```

A **camera object** (or, more briefly, a **camera**) is used to define a point of view, a range of visible objects, and a method of projection for generating a two-dimensional image of those objects from a three-dimensional model.

A **geometric object** is a type of QuickDraw 3D object that describes a particular kind of drawable shape, such as a triangle or a mesh. QuickDraw 3D defines many types of primitive geometric objects. See the chapter "Geometric Objects" for a complete description of the primitive geometric objects.

A **group object** (or, more briefly, a **group**) is a type of QuickDraw 3D object that you can use to collect objects together into lists or hierarchical models.

A **light object** (or, more briefly, a **light**) is a type of QuickDraw 3D object that you can use to provide illumination to the surfaces in a scene.

A **reference object** contains a reference to an object in a file object. Currently, however, there are no functions provided by QuickDraw 3D that you can use to create or manipulate reference objects.

Shader objects are used in the QuickDraw 3D shading architecture to provide shading in a model. See the chapter "Shader Objects" for information about these types of objects.

A **style object** (or more briefly, a **style**) is a type of QuickDraw 3D object that determines some of the basic characteristics of the renderer used to render the curves and surfaces in a scene.

A **transform object** (or, more briefly, a **transform**) is an object that you can use to modify or transform the appearance or behavior of a QuickDraw 3D object. You can use transforms to alter the coordinate system containing geometric shapes, thereby permitting objects to be repositioned and reoriented in space.

An **unknown object** is created when QuickDraw 3D encounters data it doesn't recognize while reading objects from a metafile. (This might happen, for instance, if you application reads a metafile created by another application that has defined a custom attribute or object type.) You cannot create an unknown object explicitly, but QuickDraw 3D provides routines that you can use to look at the contents of an unknown object.

Group Object Subclasses

There is only one subclass of the `TQ3GroupObject` class: the display group object.

```
typedef TQ3GroupObject                  TQ3DisplayGroupObject;
```

A **display group** is a group of objects that are drawable.

Shader Object Subclasses

There are several subclasses of the `TQ3ShaderObject` class.

```
typedef TQ3ShapeObject                  TQ3SurfaceShaderObject;
typedef TQ3ShapeObject                  TQ3IlluminationShaderObject;
```

Surface shader objects and illumination shader objects are used in the QuickDraw 3D shading architecture to provide shading in a model. See the chapter "Shader Objects" for information about these types of objects.

Reference Counts

As mentioned in "QuickDraw 3D Object Subclasses" (page 167), a shared object is a QuickDraw 3D object that can be shared by two or more other QuickDraw 3D objects. QuickDraw 3D maintains an internal reference count for each shared object to keep track of the number of times an object is being shared. Certain operations on the object increase the reference count, and other operations decrease it. For example, when you first create a spot light (by calling `Q3SpotLight_New`), its reference count is set to 1. If you later share that light (for example, by adding it to a group object), the reference count of the light is increased to indicate the additional link to the light. Figure 3-2 illustrates a series of operations involving a spot light and a group.

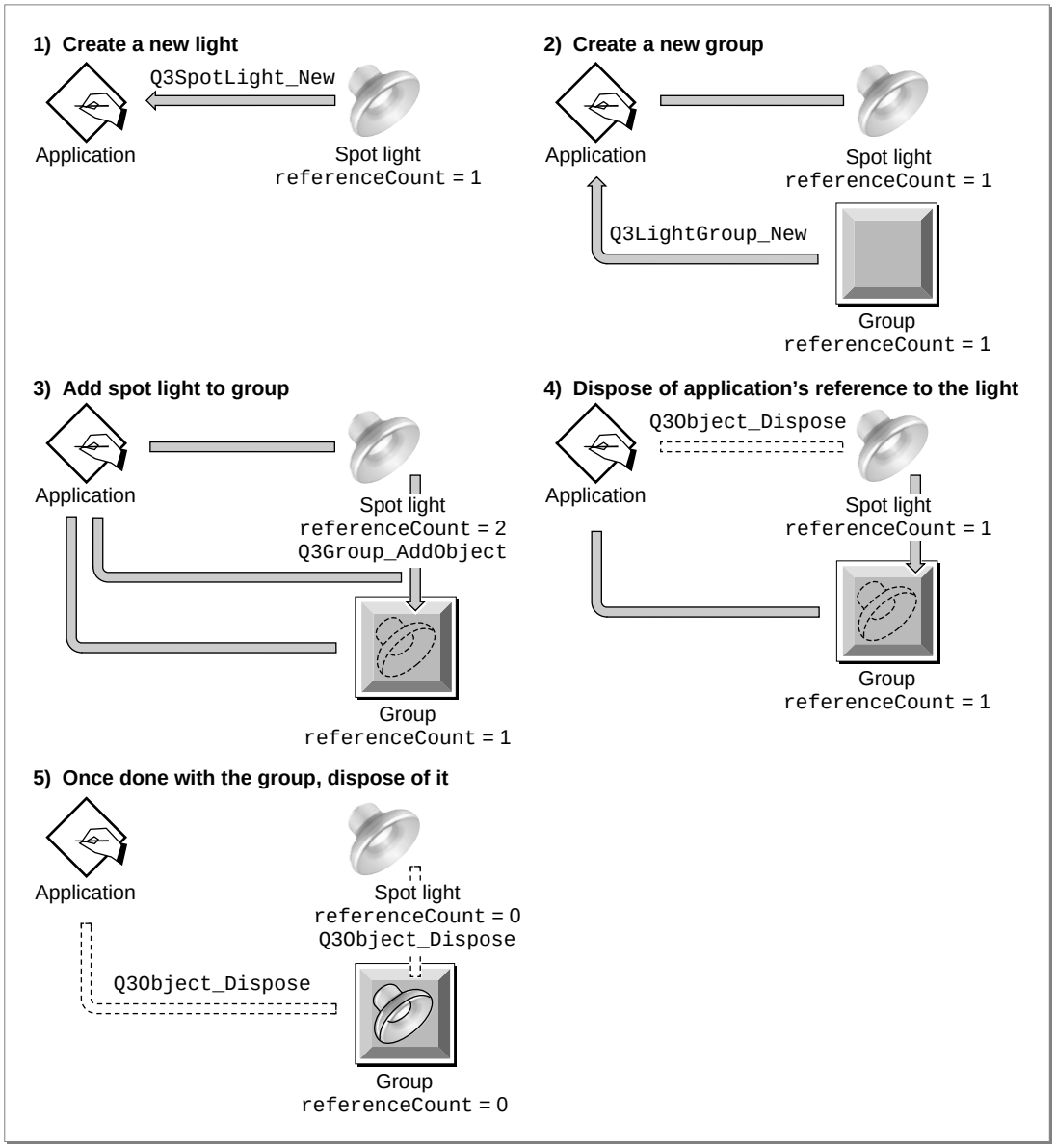
In step 1, an application creates a new spot light by calling `Q3SpotLight_New`. As indicated above, the reference count of the new spot light is set to 1. Then, in

step 2, the application creates a new light group. A light group is a shared object and hence also has a reference count, which is set to 1 upon its creation. In step 3, the application adds the spot light to the light group by calling `Q3Group_AddObject`. The reference count of the spot light is therefore increased to 2, because both the application and the light group possess references to the spot light. Note that the reference count of the group remains at 1.

In general, when you create a light and add it to a group, you can dispose of your application's reference to the light by calling `Q3Object_Dispose`. When this is done, in step 4, the reference count of the light is decremented to 1. The only remaining reference to the light is maintained by the group, not by the application. Finally, when you have finished using the light, you can dispose of the group object by calling `Q3Object_Dispose` once again (step 5). When that happens, the objects in the group are disposed of and the group itself is disposed of. The reference counts of both the light and the group fall to 0, in which case they are both removed from memory.

If the application had *not* explicitly disposed of the spot light (as happened in step 4), the reference count of the light would have remained at 2 until the group was disposed of (step 5), at which time it would have decreased to 1. The application could then call `Q3Object_Dispose` to decrease the reference count to 0, thereby disposing of the light object. In effect, `_New` and `_Dispose` calls define the scope of an object inside your application. You cannot operate on the object until you've created it using a `_New` call, and you cannot in general operate on an object after you've disposed of it by calling `Q3Object_Dispose`.

Figure 3-2 Incrementing and decrementing reference counts



Certain operations increase the reference counts of shared objects, including

- creating a new shared object (the reference count is set to 1)
- getting a reference to a shared object
- adding a shared object to a group
- setting the shared object located at a certain position in a group

Naturally, the inverse operations decrease the reference counts of shared objects, including

- disposing of a shared object
- removing a shared object from a group
- disposing of a group that contains a shared object
- replacing a shared object in any object (for example, a group or a view) with another shared object

For example, the following code gets and disposes of the camera object associated with a view:

```
TQ3ViewObject       view;
TQ3CameraObject     camera;

Q3View_GetCamera(view, &camera);
Q3Object_Dispose(camera);
camera = NULL;
```

The following code shows how a reference count is increased when obtaining an object at a given position in a group. Note that the transform which `Q3Group_GetPositionObject` returned from the group must be disposed of:

```
TQ3GroupObject          group;
TQ3GroupPosition        position;
TQ3TransformObject      transform;
TQ3Matrix4x4            matrix;

Q3Group_GetPositionObject(group, position, &transform);
Q3Transform_GetMatrix(transform, &matrix);
Q3Object_Dispose(transform);
transform = NULL;
```

If you do not directly or indirectly balance every operation that increments an object's reference count with an operation that decrements the reference count, you risk creating memory leaks. See the Listing 1-6 (page 63) for examples of how to balance an object's reference count.

You need to directly dispose only of an object reference that your application receives when it creates a QuickDraw 3D object. Any other reference to the object must be indirectly disposed of. For example, suppose that you create a translate transform object and then add it to a group twice, as follows:

```
myTransform = Q3TranslateTransform_New(&myVector3D);
Q3Group_AddObject(myGroup, myTransform);
Q3Group_AddObject(myGroup, myTransform);
```

In this example, the reference count is incremented each time you call `Q3Group_AddObject`. However, you should dispose of the transform object only once, because the transform's reference count is decremented twice when you dispose of the group.

Using QuickDraw 3D Objects

This section describes the most basic ways of using QuickDraw 3D objects. In particular, it tells you how you can

- determine the type of a QuickDraw 3D object
- define a simple object metahandler to support a custom object type

Determining the Type of a QuickDraw 3D Object

Every class in the QuickDraw 3D class hierarchy has a unique type identifier associated with it. For example, the triangle class has the type identifier `kQ3GeometryTypeTriangle`. For objects you create, of course, you'll generally know the type of the object. In some instances, however, you might need to determine an object's type, so that you know what methods apply to the object. For example, when you read an object from a file, you don't usually know what kind of object you've read.

The QuickDraw 3D class hierarchy supports `_GetType` methods at all levels of the hierarchy. At the root level, the function `Q3Object_GetType` returns a constant

of the form `kQ3ObjectType`*SubClass*, where *SubClass* is replaced by the appropriate subclass identifier.

For example, suppose you've read an object (which happens to be a triangle) from a file and you want to determine what kind of object it is. You can call the `Q3Object_GetType` function, which returns the value `kQ3ObjectTypeShared`. To determine what kind of shared object it is, you can call the `Q3Shared_GetType` function, which in this case returns the value `kQ3SharedTypeShape`. To determine what kind of shape object it is, you can call the `Q3Shape_GetType` function, which in this case returns the value `kQ3ShapeTypeGeometry`. Finally, you can determine what kind of geometric object it is by calling `Q3Geometry_GetType`; in this case, `Q3Geometry_GetType` returns the value `kQ3GeometryTypeTriangle`.

Instead of descending the class hierarchy in this way, you can also determine the leaf type of an object by calling the `Q3Object_GetLeafType` function. (An object's **leaf type** is the identifier of a leaf class.) In this example, calling `Q3Object_GetLeafType` returns the constant `kQ3GeometryTypeTriangle`.

You can also use the `Q3Object_IsType` function to determine if an object is of a particular type.

Defining an Object Metahandler

QuickDraw 3D allows you to define object types in addition to those it provides itself. For example, you can add a custom type of attribute so that you can attach custom data to objects or parts of objects in a model. Similarly, you can add custom types of geometric objects if those supplied by QuickDraw 3D are not sufficient for your needs.

Custom objects use the following type definition:

```
typedef struct TQ3ObjectClassPrivate *TQ3ObjectClass;
```

To define a custom object type, you first define the structure of the data associated with your custom object type. Then you must write an object metahandler to define a set of object-handling methods. QuickDraw 3D calls those methods at certain times to handle operations on your custom object. For example, when someone calls `Q3Object_Submit` to draw an object of your custom type, QuickDraw 3D must call your object's drawing method.

How Your Metahandler is Called

When you pass a metahandler to QuickDraw 3D, it is called multiple times to build method tables and is then thrown away. You are guaranteed that your metahandler will never be called again after the call that was passed to it returns.

Your metahandler should contain a switch on the method type passed to it and should return the corresponding method as a `TQ3XFunctionPointer`:

```
typedef void (QD3D_CALLBACK *TQ3XFunctionPointer)(void);

typedef unsigned long TQ3XMethodType;

typedef TQ3XFunctionPointer (QD3D_CALLBACK *TQ3XMetaHandler)
                    (TQ3XMethodType    methodType);
```

For a description of the `QD3D_CALLBACK` macro, see the `QD3D.h` header file.

IMPORTANT
A metahandler must always return a value. If it is passed a method type that it does not understand, it must return a value of `NULL`. ▲

Defining Custom Elements

You can define custom element types if you'd like to support types of attributes other than those provided by QuickDraw 3D. You define custom attributes as custom elements because attributes are almost always contained in an attribute set, of type `TQ3AttributeSet`. More generally, you can define custom element types that can be included in a set of type `TQ3SetObject`.

See "Creating Custom Object Subclasses," beginning on page 194 for complete details of the methods you need to define to support a custom element type.

QuickDraw 3D Objects Reference

This section describes the routines provided by QuickDraw 3D for managing objects and shared objects. This section also describes the methods your application can define to allow QuickDraw 3D to work with custom objects.

QuickDraw 3D Objects Routines

This section describes the routines you can use with QuickDraw 3D objects in general and with shared objects.

Managing Objects

QuickDraw 3D provides several routines that you can use to operate on any QuickDraw 3D object. The top level of the QuickDraw 3D class hierarchy (`TQ3Object`) supports dispose, duplicate, draw, and file I/O methods.

Q3Object_Submit

You can use the `Q3Object_Submit` function to submit a QuickDraw 3D object for drawing, picking, bounding, or writing.

```
TQ3Status Q3Object_Submit (TQ3Object object, TQ3ViewObject view);
```

`object` A QuickDraw 3D object.

`view` A view.

DESCRIPTION

The `Q3Object_Submit` function submits the QuickDraw 3D object specified by the `object` parameter for drawing, picking, bounding, or writing in the view specified by the `view` parameter.

SPECIAL CONSIDERATIONS

You should call this function only in a submitting loop.

ERRORS

```
kQ3ErrorInvalidObjectParameter
kQ3ErrorOutOfMemory
kQ3ErrorBeginWriteNotCalled
kQ3ErrorNoWriteMethods
kQ3ErrorEndOfFile
```

Q3Object_Dispose

You can use the `Q3Object_Dispose` function to dispose of a QuickDraw 3D object.

```
TQ3Status Q3Object_Dispose (TQ3Object object);
```

`object` A QuickDraw 3D object.

DESCRIPTION

The `Q3Object_Dispose` function disposes of the QuickDraw 3D object specified by the `object` parameter. If the specified object is not a shared object, QuickDraw 3D disposes of any memory occupied by that object. If the specified object is a shared object, QuickDraw 3D reduces by 1 the reference count associated with that object. When the reference count is reduced to 0, `Q3Object_Dispose` disposes of the memory occupied by the object.

In general, you need to call `Q3Object_Dispose` for any objects returned by a `Get` call (for example, `Q3View_GetDrawContext`). Failure to call `Q3Object_Dispose` on such objects will result in a memory leak.

ERROR

```
kQ3ErrorInvalidObjectParameter
```

Q3Object_Duplicate

You can use the `Q3Object_Duplicate` function to duplicate a QuickDraw 3D object.

```
TQ3Object Q3Object_Duplicate (TQ3Object object);
```

`object` A QuickDraw 3D object.

DESCRIPTION

The `Q3Object_Duplicate` function returns, as its function result, a QuickDraw 3D object that is an exact duplicate of the QuickDraw 3D object specified by the `object` parameter. If the new object is a shared object, its reference count is set to 1.

IMPORTANT

The `Q3Object_Duplicate` function cannot duplicate `DrawContext` or `View` objects. These object classes contain resources created by the application that belong to the window system. Because the QuickDraw 3D library cannot duplicate windows or colormaps, it cannot duplicate objects containing them. ▲

ERRORS

```
kQ3ErrorInvalidObjectParameter
kQ3ErrorOutOfMemory
```

Q3Object_IsDrawable

You can use the `Q3Object_IsDrawable` function to determine whether a QuickDraw 3D object is drawable.

```
TQ3Boolean Q3Object_IsDrawable (TQ3Object object);
```

`object` A QuickDraw 3D object.

DESCRIPTION

The `Q3Object_IsDrawable` function returns, as its function result, a Boolean value that indicates whether the QuickDraw 3D object specified by the `object` parameter is drawable (`kQ3True`) or not (`kQ3False`).

ERROR

`kQ3ErrorInvalidObjectParameter`

Q3Object_IsWritable

You can use the `Q3Object_IsWritable` function to determine whether a QuickDraw 3D object is writable.

```
TQ3Boolean Q3Object_IsWritable (TQ3Object object);
```

`object` A QuickDraw 3D object.

DESCRIPTION

The `Q3Object_IsWritable` function returns, as its function result, a Boolean value that indicates whether the QuickDraw 3D object specified by the `object` parameter can be written to a file object (`kQ3True`) or not (`kQ3False`).

ERROR

`kQ3ErrorInvalidObjectParameter`

Determining Object Types

QuickDraw 3D provides routines that you can use to determine the type and name of a QuickDraw 3D object. Object types are declared as follows:

```
typedef long        TQ3ObjectType;
```

Q3Object_GetType

You can use the `Q3Object_GetType` function to get the type of a core QuickDraw 3D object.

```
TQ3ObjectType Q3Object_GetType (TQ3Object object);
```

`object` A QuickDraw 3D object.

DESCRIPTION

The `Q3Object_GetType` function returns, as its function result, the type identifier of the QuickDraw 3D object specified by the `object` parameter. If successful, `Q3Object_GetType` returns one of these constants:

```
kQ3ObjectTypeElement
kQ3ObjectTypePick
kQ3ObjectTypeShared
kQ3ObjectTypeView
```

If the type cannot be determined or is invalid, `Q3Object_GetType` returns the value `kQ3ObjectTypeInvalid`.

ERRORS

```
kQ3ErrorNULLParameter
kQ3ErrorInvalidObjectParameter
```

Q3Object_GetLeafType

You can use the `Q3Object_GetLeafType` function to get the leaf type of a QuickDraw 3D object.

```
TQ3ObjectType Q3Object_GetLeafType (TQ3Object object);
```

`object` A QuickDraw 3D object.

DESCRIPTION

The `Q3Object_GetLeafType` function returns, as its function result, the leaf type identifier of the QuickDraw 3D object specified in the `object` parameter. You should call this function only when the specified object is a leaf object (for example, when you've read the object in from a file). If the leaf type cannot be determined or is invalid, `Q3Object_GetLeafType` returns the value `kQ3ObjectTypeInvalid`.

ERRORS

```
kQ3ErrorNULLParameter
kQ3ErrorInvalidObjectParameter
```

Q3Object_IsType

You can use the `Q3Object_IsType` function to determine whether a QuickDraw 3D object is of a specific type.

```
TQ3Boolean Q3Object_IsType (
                    TQ3Object object,
                    TQ3ObjectType type);
```

`object` A QuickDraw 3D object.

`type` A type identifier.

DESCRIPTION

The `Q3Object_IsType` function returns a Boolean value that indicates whether the QuickDraw 3D object specified by the `object` parameter is of the type specified by the `type` parameter (`kQ3True`) or is of some other type (`kQ3False`). You can pass any valid QuickDraw 3D type identifier in the `type` parameter (not just those that are returned by the `Q3Object_GetType` function). For example, you can use `Q3Object_IsType` like this:

```
if (Q3Object_IsType(
        object,
        kQ3ShapeType_Geometry) {
    ...
```

```
} else if (Q3Object_IsType(
        object,
        kQ3SharedType_File)) {
    ...
```

ERRORS

```
kQ3ErrorNULLParameter
kQ3ErrorInvalidObjectParameter
```

Analyzing the Object Hierarchy

QuickDraw 3D provides routines to help you analyze its object hierarchy.

An example of using object hierarchy analysis functions is given in Listing 3-1. This example recursively prints all the subclasses for a particular class to `stdout`, assuming that an ANSI C support library is available. If you wanted to print out the entire class hierarchy for QuickDraw 3D, you could use this routine in the way shown at the end of the example.

Listing 3-1 Example of hierarchy analysis

```
void PrintClassAndRecurse(
    TQ3ObjectType   objectClassType,
    int             depth)
{
    TQ3SubClassData             subClassData;
    TQ3ObjectClassNameString    objectClassString;
    unsigned long               index;

    depth++;
    if (objectClassType != kQ3ObjectTypeInvalid) {

        Q3ObjectHierarchy_GetStringFromType(objectClassType,
            objectClassString);

        for (index = 0; index < depth; index++) {
            printf(" ");
        }
```

```
        printf("%s\n", objectClassString);

        Q3ObjectHierarchy_GetSubClassData(objectClassType, &subClassData);

        for (index = 0; index < subClassData.numClasses; index++) {
            /* recurse on each subclass type */
            PrintClassAndRecurse(subClassData.classTypes[index], depth);
        }

        Q3ObjectHierarchy_EmptySubClassData(&subClassData);
    }
    depth--;
}

/*
*   The class "Object" is in fact a virtual base class -- it is not
*   possible to instantiate this class. At the root of the hierarchy
*   are four classes: View, Pick, Element, and Shared. So we can go from
*   each of these classes, instead of going from "Object".
*/

    printf("Root Object (virtual metaclass)\n");
    PrintClassAndRecurse(kQ3ObjectTypeView, 0);
    PrintClassAndRecurse(kQ3ObjectTypeElement, 0);
    PrintClassAndRecurse(kQ3ObjectTypePick, 0);
    PrintClassAndRecurse(kQ3ObjectTypeShared, 0);
```

Q3ObjectHierarchy_GetTypeFromString

You can use the `Q3ObjectHierarchy_GetTypeFromString` function to obtain the class type for a given class name.

```
typedef char    TQ3ObjectClassNameString[kQ3StringMaximumLength];

kQ3StringMaximumLength = 1024

TQ3Status Q3ObjectHierarchy_GetTypeFromString(
                    TQ3ObjectClassNameString    objectClassString,
                    TQ3ObjectType               *objectClassType);
```

`objectClassString`
A class name as a C string.

`objectClassType`
On return, the class type.

DESCRIPTION

The `Q3ObjectHierarchy_GetTypeFromString` function returns, in the `objectClassType` parameter, the class type associated with the name in the `objectClassString` parameter. If `objectClassString` is invalid, the routine will return `kQ3Failure`.

Q3ObjectHierarchy_GetStringFromType

You can use the `Q3ObjectHierarchy_GetStringFromType` function to obtain the class name for a given class type.

```
TQ3Status Q3ObjectHierarchy_GetStringFromType(
                    TQ3ObjectType               objectClassType,
                    TQ3ObjectClassNameString    objectClassString);
```

`objectClassType`
A class type.

`objectClassString`
On return, a class name as a C string.

DESCRIPTION

The `Q3ObjectHierarchy_GetStringFromType` function returns, in the `objectClassString` parameter, the class name associated with the type in the `objectClassType` parameter. If `objectClassType` is invalid, the routine will return `kQ3Failure`.

Q3ObjectHierarchy_IsTypeRegistered

You can use the `Q3ObjectHierarchy_IsTypeRegistered` function to determine if a class type is registered.

```
TQ3Boolean Q3ObjectHierarchy_IsTypeRegistered(
                    TQ3ObjectType    objectClassType);
```

`objectClassType`
A class type.

DESCRIPTION

The `Q3ObjectHierarchy_IsTypeRegistered` function returns `TRUE` if the class type specified by `objectClassType` is registered and `FALSE` otherwise.

Q3ObjectHierarchy_IsNameRegistered

You can use the `Q3ObjectHierarchy_IsNameRegistered` function to determine if a class name is registered.

```
TQ3Boolean Q3ObjectHierarchy_IsNameRegistered(
                    const char    *objectClassName);
```

`objectClassName`
A class name as a C string.

DESCRIPTION

The `Q3ObjectHierarchy_IsNameRegistered` function returns `TRUE` if the class name specified by `objectClassName` is registered and `FALSE` otherwise.

Q3ObjectHierarchy_GetSubClassData

You can use the `Q3ObjectHierarchy_GetSubClassData` function to obtain the number and class types of all the subclasses immediately below a class in the QuickDraw 3D class hierarchy.

```
typedef struct TQ3SubClassData {
    unsigned long    numClasses;    /* the # of subclass types found */
    TQ3ObjectType    *classTypes;   /* an array of class types */
} TQ3SubClassData;

TQ3Status Q3ObjectHierarchy_GetSubClassData(
                    TQ3ObjectType      objectClassType,
                    TQ3SubClassData    *subClassData);
```

`objectClassType`
An object class type.

`subClassData` Pointer to a `TQ3SubClassData` struct containing the number and class types of the subclasses below `objectClassType`.

DESCRIPTION

The `Q3ObjectHierarchy_GetSubClassData` function returns, in the `subClassData` parameter, the number and class types of all the subclasses immediately below the class designated by `objectClassType`.

This call must be followed by a call to `Q3ObjectHierarchy_EmptySubClassData` to avoid memory leaks.

Q3ObjectHierarchy_EmptySubClassData

You must use the `Q3ObjectHierarchy_EmptySubClassData` function to free memory allocated by `Q3ObjectHierarchy_GetSubClassData`.

```
TQ3Status Q3ObjectHierarchy_EmptySubClassData(
                    TQ3SubClassData    *subClassData);
```

`subClassData` Pointer to a `TQ3SubClassData` struct.

DESCRIPTION

The `Q3ObjectHierarchy_EmptySubClassData` function frees memory allocated for `subClassData` by a previous call to `Q3ObjectHierarchy_GetSubClassData`.

Managing Shared Objects

QuickDraw 3D provides routines that you can use to get a reference to a shared object or to get the type of a shared object.

Q3Shared_GetReference

You can use the `Q3Shared_GetReference` function to get a reference to a shared object.

```
TQ3SharedObject Q3Shared_GetReference (TQ3SharedObject sharedObject);
```

`sharedObject` A shared object.

DESCRIPTION

The `Q3Shared_GetReference` function returns, as its function result, a reference to the shared object specified by the `sharedObject` parameter. You can use this function to prevent QuickDraw 3D from deleting an object twice.

ERRORS

```
kQ3ErrorNULLParameter
kQ3ErrorInvalidObjectParameter
```

Q3Shared_IsReferenced

You can use the `Q3Shared_IsReferenced` function to determine whether a shared object has more than one reference to it.

```
TQ3Boolean Q3Shared_IsReferenced (TQ3SharedObject sharedObject);
```

`sharedObject` A shared object.

DESCRIPTION

The `Q3Shared_IsReferenced` function returns, as its function result, a Boolean value that indicates whether the shared object specified by the `sharedObject` parameter has more than one reference to it (`kQ3True`) or has only one reference to it (`kQ3False`).

The `Q3Shared_IsReferenced` function is intended for use by an application or other code that needs to determine whether it has the only existing reference to a shared object.

SPECIAL CONSIDERATIONS

You should never call `Q3Shared_IsReferenced` as follows:

```
while (Q3Shared_IsReferenced(mySharedObject)) {
    Q3Object_Dispose(mySharedObject);
}
```

This code will cause your application to crash.

ERRORS

```
kQ3ErrorNULLParameter
kQ3ErrorInvalidObjectParameter
```

Q3Shared_GetType

You can use the `Q3Shared_GetType` function to get the type of a shared object.

```
TQ3ObjectType Q3Shared_GetType (TQ3SharedObject sharedObject);
```

`sharedObject` A shared object.

DESCRIPTION

The `Q3Shared_GetType` function returns, as its function result, the type identifier of the shared object specified by the `sharedObject` parameter. If successful, `Q3Shared_GetType` returns one of these constants:

```
kQ3SharedTypeAttachment
kQ3SharedTypeControllerState
kQ3SharedTypeDrawContext
kQ3SharedTypeFile
kQ3SharedTypeReference
kQ3SharedTypeRenderer
kQ3SharedTypeSet
kQ3SharedTypeShape
kQ3SharedTypeShapePart
kQ3SharedTypeStorage
kQ3SharedTypeString
kQ3SharedTypeTexture
kQ3SharedTypeTracker
kQ3SharedTypeViewHints
```

If the type cannot be determined or is invalid, `Q3Shared_GetType` returns the value `kQ3ObjectTypeInvalid`.

ERRORS

```
kQ3ErrorNULLParameter
kQ3ErrorInvalidObjectParameter
```

Q3Shared_GetEditIndex

You can use the `Q3Shared_GetEditIndex` function to get the edit index for a shared object.

```
unsigned long Q3Shared_GetEditIndex (TQ3SharedObject sharedObject);
```

`sharedObject` A shared object.

DESCRIPTION

The `Q3Shared_GetEditIndex` function returns, as its function result, the current edit index of the shared object specified by the `sharedObject` parameter. An **edit index** is a unique number associated with a shared object that changes each time the object is edited. You can use the edit index to determine whether an object you are caching has changed since the object was cached, using code such as the following:

```
struct -> editIndex = Q3Shared_GetEditIndex(foo);
struct -> object = Q3Shared_GetReference(foo);
```

Later you can determine if the object has been edited:

```
if (struct->editIndex !=
        Q3Shared_GetEditIndex(struct->object)) {
                /* Has been edited -- update or re-create caches. */
} else {
                /* Not edited */
    }
```

SEE ALSO

Use the `Q3Shared_Edited` function to manually change an object's edit index.

ERRORS

```
kQ3ErrorNULLParameter
kQ3ErrorInvalidObjectParameter
kQ3ErrorInvalidObjectType
```

Q3Shared_Edited

You can use the `Q3Shared_Edited` function to change a shared object's edit index.

```
TQ3Status Q3Shared_Edited (TQ3SharedObject sharedObject);
```

`sharedObject` A shared object.

DESCRIPTION

The `Q3Shared_Edited` function changes the edit index of the shared object specified by the `sharedObject` parameter. This function is designed for use by shared plug-in objects; those objects should call `Q3Shared_Edited` whenever their private data changes.

ERRORS

```
kQ3ErrorNULLParameter
kQ3ErrorInvalidObjectParameter
```

Extending Shapes and Sets

The QuickDraw 3D shape and set routines are discussed in "Managing Shapes," beginning on page 81.

The `Q3Shape_GetSet` and `Q3ShapeSetSet` calls are implemented via the element type `kQ3ElementTypeSet`:

```
typedef long    TQ3ElementType;

#define kQ3ElementTypeNone      0
#define kQ3ElementTypeUnknown   32
#define kQ3ElementTypeSet       33
#define kQ3ElementTypeName      34
#define kQ3ElementTypeURL       35
```

The expression `Q3Shape_GetSet(s,&o)` is eqivalent to

```
Q3Shape_GetElement(s, kQ3ElementTypeSet, &o)
```

The expression `Q3Shape_SetSet(s,o)` is eqivalent to

```
Q3Shape_SetElement(s, kQ3ElementTypeSet, &o)
```

It is important to note that a `Q3Shape_...Element...` call does not create a set on a shape and then add the element to it. The data is attached directly to the shape. Therefore, it is possible for an element to exist on a shape without a set existing on it as well.

In your application, if you attach an element to a shape in this way:

```
set = Q3Set_New();
Q3Set_AddElement(set, ElemType, &data);
Q3Shape_SetSet(shape, set);
```

You should retrieve it in the same manner:

```
Q3Shape_GetSet(shape, &set);
if (Q3Set_Contains(set, ElemType) == kQ3True) {
    Q3Set_Get(set, ElemType, &data);
}
```

Similarly, if you attach data to a shape with the call

```
Q3Shape_AddElement(shape, ElemType, &data);
```

You should retrieve it in the same manner:

```
if (Q3Shape_ContainsElement(
    set, ElemType) == kQ3True) {
    Q3Shape_GetElement(set, ElemType, &data);
}
```

When attempting to find a particular element on a shape, you should first check with `Q3Shape_GetNextElementType` or `Q3Shape_GetElement`. Then you should call

`Q3Shape_GetSet(s, &set)` or

`Q3Shape_GetElement(s, kQ3ElementTypeSet, &set))`.

Finally, you should call `Q3Shape_GetElement(set, ...)`.

In terms of implementation, `Q3Shape_SetSet` and `Q3Shape_GetSet` should be used only for sets of information that are shared among multiple shapes. `Q3Shape_AddElement`, `Q3Shape_GetElement`, and similar calls should be used only for elements that are unique to a particular shape.

Creating Custom Object Subclasses

In QuickDraw 3D 1.5 and later releases, the object system is extensible. This lets you increase functionality in certain supported areas by adding plug-in

subclasses to QuickDraw 3D's native classes. The three types of plug-ins currently supported are:

- elements and attributes
- groups
- renderers

Element and attribute plug-ins let you share custom elements and attributes easily between applications. For example, you may wish to tag a geometry with a string, using a 'name' attribute. Previously, you would publish the format of the attribute or element, but this meant that each application had to reimplement the code, leading to incompatibility between different versions of custom attributes. With plug-ins, one set of elements and attributes can be shared.

Plug-in groups add powerful new capabilities. There may be times when an application needs to configure a group in a way that is not currently support by QuickDraw 3D. For example, you may wish to store references to light objects along with the geometric objects used to denote those lights, as in an interior design package where you may want to keep a light object together with a geometric model representing an angle-poised lamp. As the user manipulates the position of the lamp, the application wants to have easy access to the associated QuickDraw 3D light objects. For this purpose you can create a custom group that adds the lights. Another kind of plug-in group would be a level-of-detail group that changes the geometry of an object as the object changes its distance from the camera.

Plug-in renderers are interesting for developers because it is likely that there will be an end-user market for them as standalone products. Renderers require complex software design, but a quantity of technical literature is available to shorten your learning curve. Plug-in renderers are discussed in Chapter 11, "Renderer Objects."

You can create and use a subclass of any group, renderer, element, or attribute object class in the QuickDraw 3D hierarchy. The methods and routines described in this section let you integrate your own objects of these types with the rest of QuickDraw 3D as fully-featured objects.

Custom Class Metahandlers

Any plug-in class that you define must have a metahandler. The metahandler is the method that the system uses to associate user-supplied routines with the required methods that a class needs to implement.

When you give a metahandler to QuickDraw 3D, it is called multiple times to build method tables and then is thrown away. You are guaranteed that your metahandler will never be called again after a call that was passed a metahandler returns.

Your metahandler should contain a switch on the `methodType` passed to it and should return the corresponding method as a `TQ3XFunctionPointer`. All return values from the metahandler are cast to this type.

```
typedef void (*TQ3FunctionPointer)(void);
```

IMPORTANT

A metahandler must always return a value. If it is passed a `methodType` that it does not understand, it must return `NULL`. ▲

Generally a metahandler is implemented as one case statement. Certain types of classes, such as renderers, may have multiple levels of metahandlers.

TQ3MetaHandler

You must define an object metahandler to specify methods for custom object types or custom element types.

```
typedef unsigned long TQ3XMethodType;

typedef TQ3FunctionPointer (*TQ3MetaHandler) (
                    TQ3MethodType    methodType);
```

`methodType` A method type.

DESCRIPTION

Your `TQ3MetaHandler` function should return a function pointer (a value of type `TQ3FunctionPointer`) to the custom method whose type is specified by the

`methodType` parameter. If you do not define a method of the specified type, your metahandler should return the value `NULL`.

In general, your metahandler should contain a `switch` statement that branches on the `methodType` parameter. QuickDraw 3D calls your metahandler repeatedly to build a method table when you first pass it to a QuickDraw 3D routine. Once QuickDraw 3D has finished building the method table, your metahandler is never called again.

When any one of your custom methods is called, you can be certain that your metahandler will not be called again.

EXAMPLE

This example of a metahandler is edited from the `NameAttribute` sample on the QuickDraw 3D SDK. It is essentially a big switch statement that maps a method type selector onto a function call for each implementation method. The default return value is `NULL`, so if the metahandler is called with a method type that is not recognised it returns `NULL`. All of the values are cast to type `TQ3XFunctionPointer`, even when they are not function pointers. For example, the return value for `kQ3XMethodTypeObjectClassVersion` is a `long` that has the version of the plug-in packed into it.

```
TQ3XFunctionPointer NameAttribute_MetaHandler(
    TQ3XMethodType      methodType)
{
    switch (methodType) {
        case kQ3XMethodTypeObjectClassVersion:
            return (TQ3XFunctionPointer)Q3_OBJECT_CLASS_VERSION(
                        majorVersion, minorVersion);
        case kQ3XMethodTypeObjectTraverse:
            return (TQ3XFunctionPointer) NameAttribute_Traverse;
        case kQ3XMethodTypeObjectReadData:
            return (TQ3XFunctionPointer) NameAttribute_ReadData;
        case kQ3XMethodTypeElementCopyAdd:
        case kQ3XMethodTypeElementCopyGet:
        case kQ3XMethodTypeElementCopyDuplicate:
            return (TQ3XFunctionPointer) NameAttribute_CopyAdd;
        case kQ3XMethodTypeElementCopyReplace:
            return (TQ3XFunctionPointer) NameAttribute_CopyReplace;
        case kQ3XMethodTypeElementDelete:
            return (TQ3XFunctionPointer) NameAttribute_Delete;
```

```
            default:
                return (TQ3XFunctionPointer) NULL;
        }
    }
```

Object Types and Names

QuickDraw 3D adheres to various typing and naming schemes for its object system:

- For each unique `TQX3ObjectClass`, there is a unique `TQ3ObjectType` and a unique object name embodied in a C string. No two object classes can have the same type or the same name.
- Elements and attributes have additional naming conventions, used in calls that take parameters of type `TQ3ElementType` or `TQ3AttributeType`. The `TQ3ElementType` and `TQ3AttributeType` values are used solely to access attributes and elements from applications. An element class or attribute class also has a `TQ3ObjectType` and object name, used when reading and writing QuickDraw 3D metafiles.
- For internal element and attribute types, the `TQ3ElementType` and its corresponding `TQ3ObjectType` are generally different. For example, the attribute `kQ3AttributeTypeDiffuseColor` has an element type of 5, an object type of `'kdif'`, and an object name of `DiffuseColor`.
- For external element and attribute types, the `TQ3ElementType` is currently identical to the `TQ3ObjectType`.

Thus, each unique `TQX3ObjectClass` of type `kQ3ObjectTypeElement` has a unique `TQ3ElementType`. However, each name space is unique only within itself; for example, there can be a `TQ3ObjectType` and `TQ3ElementType` that are identical for a particular object. You can even name an object identically to the object type, if you wish. For example, you could register an element class with `TQ3ObjectType` `'foob'`, object name `foob`, and element type `'foob'`.

The public object types in the unextended QuickDraw 3D hierarchy are shown in Listing 3-2.

Listing 3-2 QuickDraw 3D object types

```
kQ3ObjectTypeInvalid        0
kQ3ObjectTypeView
kQ3ObjectTypeElement
    kQ3ElementTypeAttribute
kQ3ObjectTypePick
    kQ3PickTypeWindowPoint
    kQ3PickTypeWindowRect
kQ3ObjectTypeShared
    kQ3SharedTypeRenderer
        kQ3RendererTypeWireFrame
        kQ3RendererTypeGeneric
        kQ3RendererTypeInteractive
    kQ3SharedTypeShape
        kQ3ShapeTypeGeometry
            kQ3GeometryTypeBox
            kQ3GeometryTypeGeneralPolygon
            kQ3GeometryTypeLine
            kQ3GeometryTypeMarker
            kQ3GeometryTypePixmapMarker
            kQ3GeometryTypeMesh
            kQ3GeometryTypeNURBCurve
            kQ3GeometryTypeNURBPatch
            kQ3GeometryTypePoint
            kQ3GeometryTypePolygon
            kQ3GeometryTypePolyLine
            kQ3GeometryTypeTriangle
            kQ3GeometryTypeTriGrid
            kQ3GeometryTypeCone
            kQ3GeometryTypeCylinder
            kQ3GeometryTypeDisk
            kQ3GeometryTypeEllipse
            kQ3GeometryTypeEllipsoid
            kQ3GeometryTypePolyhedron
            kQ3GeometryTypeTorus
            kQ3GeometryTypeTriMesh
        kQ3ShapeTypeShader
            kQ3ShaderTypeSurface
                kQ3SurfaceShaderTypeTexture
            kQ3ShaderTypeIllumination
```

```
            kQ3IlluminationTypePhong
            kQ3IlluminationTypeLambert
            kQ3IlluminationTypeNULL
    kQ3ShapeTypeStyle
        kQ3StyleTypeBackfacing
        kQ3StyleTypeInterpolation
        kQ3StyleTypeFill
        kQ3StyleTypePickID
        kQ3StyleTypeReceiveShadows
        kQ3StyleTypeHighlight
        kQ3StyleTypeSubdivision
        kQ3StyleTypeOrientation
        kQ3StyleTypePickParts
        kQ3StyleTypeZCompare
        kQ3StyleTypeAntiAlias
    kQ3ShapeTypeTransform
        kQ3TransformTypeMatrix
        kQ3TransformTypeScale
        kQ3TransformTypeTranslate
        kQ3TransformTypeRotate
        kQ3TransformTypeRotateAboutPoint
        kQ3TransformTypeRotateAboutAxis
        kQ3TransformTypeQuaternion
        kQ3TransformTypeReset

    kQ3ShapeTypeLight
        kQ3LightTypeAmbient
        kQ3LightTypeDirectional
        kQ3LightTypePoint
        kQ3LightTypeSpot
    kQ3ShapeTypeCamera
        kQ3CameraTypeOrthographic
        kQ3CameraTypeViewPlane
        kQ3CameraTypeViewAngleAspect
    kQ3ShapeTypeGroup
        kQ3GroupTypeDisplay
            kQ3DisplayGroupTypeOrdered
            kQ3DisplayGroupTypeIOProxy
        kQ3GroupTypeLight
        kQ3GroupTypeInfo
    kQ3ShapeTypeUnknown
```

```
            kQ3UnknownTypeText
            kQ3UnknownTypeBinary
        kQ3ShapeTypeReference
    kQ3SharedTypeSet
        kQ3SetTypeAttribute
    kQ3SharedTypeDrawContext
        kQ3DrawContextTypePixmap
        kQ3DrawContextTypeMacintosh
        kQ3DrawContextTypeWin32DC
        kQ3DrawContextTypeDDSurface
    kQ3SharedTypeTexture
        kQ3TextureTypePixmap
        kQ3TextureTypeMipmap
    kQ3SharedTypeFile
    kQ3SharedTypeStorage
        kQ3StorageTypeMemory
            kQ3MemoryStorageTypeHandle
        kQ3StorageTypeUnix
            kQ3UnixStorageTypePath
        kQ3StorageTypeMacintosh
            kQ3MacintoshStorageTypeFSSpec
    kQ3SharedTypeString
        kQ3StringTypeCString
    kQ3SharedTypeShapePart

        kQ3ShapePartTypeMeshPart
            kQ3MeshPartTypeMeshFacePart
            kQ3MeshPartTypeMeshEdgePart
            kQ3MeshPartTypeMeshVertexPart
    kQ3SharedTypeControllerState
    kQ3SharedTypeTracker
    kQ3SharedTypeViewHints
kQ3ObjectTypeEndGroup
```

Data Structures Associated With a Class

An object class usually has several public data structures that the object system makes available to other code. A class may also have private class data that it maintains to track methods and other internal information. This private data, of

`methodsSize` bytes, is passed to the `Q3ObjectHierarchy_RegisterClass` call. It generally contains or references the public data in some way.

A class's public data structure is passed in external `_New` and `_Submit` calls. The data is passed around the system for rendering, I/O, and other functions. The data structure, which is not specified in the `Q3ObjectHierarchy_RegisterClass` call, should be published so other applications may use the object.

Registering a Custom Class

This section describes the routines that QuickDraw 3D provides to register and unregister custom object classes.

When a plug-in custom type is registered, its type parameter is allocated dynamically. Types are registered with `Q3XObjectHierarchy_RegisterClass` or `Q3XElementClass_Register`, or with the `Q3XAttributeClass_Register` routine as described in "Adding Application-Defined Attribute and Element Types," beginning on page 537. In the case of `Q3XObjectHierarchy_RegisterClass`, the second parameter is the address of an object type—`TQ3ObjectType`, `TQ3ElementType` or `TQ3AttributeType`, depending on the call used for registration.

IMPORTANT

When your custom class uses a shared library, you should coordinate class registration with library registration. For more information, including sample code, see "Registering a Shared Library," beginning on page 207. ▲

Q3XObjectHierarchy_RegisterClass

You can use the `Q3XObjectHierarchy_RegisterClass` routine to register a class in the QuickDraw 3D hierarchy.

```
TQ3XObjectClass Q3XObjectHierarchy_RegisterClass(
                    TQ3ObjectType     parentType,
                    TQ3ObjectType     *objectType,
                    char              *objectName,
                    TQ3MetaHandler    metaHandler,
```

```
                    TQ3MetaHandler    virtualMetaHandler,
                    unsigned long     methodsSize,
                    unsigned long     instanceSize);
```

`parentType` The object type for which you want to create a subclass. The parent class must be currently registered with QuickDraw 3D. Pass in `kQ3ObjectTypeInvalid` (value 0) to create a subclass of the class `TQ3Object`.

`objectType` On return, the object type of your subclass. This value is used as the binary type in a QuickDraw 3D metafiles. It is also returned in `_GetType` and `Q3Object_GetLeafType` calls and may be used in the `Q3Object_IsType` call.

`objectName` The object name of your subclass. This C string must be unique among all registered classes in QuickDraw 3D, including parent classes. This value is used as the ASCII type in QuickDraw 3D metafiles.

`metaHandler` A metahandler used to retrieve object methods and nonvirtual methods. This metahandler will be called repeatedly with a selector of type `TQ3MethodType` to retrieve methods for this object class. With some classes, this value will be `NULL`.

`virtualMetaHandler`
A metahandler used to retrieve virtual methods for your object and any of its subclasses. If you are registering a leaf class only, pass `NULL`. Classes that register as a subclass of this class will use the methods supplied here unless overridden by the subclass metahandlers. Methods that take a pointer to an object's private data should not be returned in this metahandler, as the methods only apply to the data in this class.

`methodsSize` Indicates the size of any private class data in the class. If you are registering a leaf class only, pass 0. If you have private class data, a method of type `kQ3MethodTypeObjectClassRegister` must be registered to initialize the private data structure. A pointer to the structure is returned by `Q3ObjectClass_GetClassPrivate` or `Q3Object_GetClassPrivate`. If `methodsSize` is 0, these calls always return `NULL`.

`instanceSize` The size of private instance data for your object. If this class has no instance data, this value may be 0. This would happen only if the class is used just abstractly to be subclassed or exists solely

as a type. If a nonzero value is passed in, a method of type `kQ3MethodTypeObjectNew` must be registered to initialize the data. A pointer to the structure is returned by `Q3Object_GetPrivate`. If `dataSize` is 0, `Q3Object_GetPrivate` always returns `NULL`.

DESCRIPTION

The `Q3XObjectHierarchy_RegisterClass` routine registers the custom class detailed by its parameters. The object type is assigned at run time and returned to you in the `objectType` parameter. Often it is a good idea to store this type locally in a static variable, since it is used by many object system routines.

The `Q3XObjectHierarchy_RegisterClass` routine returns `NULL` if the class could not be registered.

SPECIAL CONSIDERATIONS

You should generally call `Q3XObjectHierarchy_RegisterClass` only in a function that has been registered by the `Q3XSharedLibrary_Register` call. Register the existence of this routine instead of calling it directly from a shared library registration routine.

EXAMPLE

The following is an example of a registration function, taken from the plug-in renderer sample in the QuickDraw 3D SDK. In this example the return value of the `Q3XObjectHierarchy_RegisterClass` function is stored in the global variable `SRgRendererClass`. To make this variable readily available to other code, it is declared static to the file in which the routine is implemented.

```
TQ3Status SR_Register(
    void)
{
    /* Create/register the class */
    SRgRendererClass =
        Q3XObjectHierarchy_RegisterClass(
            kQ3SharedTypeRenderer,
            &SRgClassType,
            "SampleRenderer",
            SR_MetaHandler,
```

```
            NULL,
            0,
            sizeof(TSRPrivate));

    /* Make sure it worked */
    if (SRgRendererClass == NULL) {
        return (kQ3Failure);
    }

    return (kQ3Success);
}
```

Q3XObjectHierarchy_UnregisterClass

You can use the `Q3XObjectHierarchy_UnregisterClass` function to remove a custom object class registered with `Q3XObjectHierarchy_RegisterClass`.

```
TQ3Status Q3XObjectHierarchy_UnregisterClass (
                    TQX3ObjectClass objectClass);
```

`objectClass` An object class.

DESCRIPTION

The `Q3XObjectHierarchy_UnregisterClass` function unregisters the custom object class specified by the `objectClass` parameter.

You should dispose of all instances of the custom object class you want to unregister before calling `Q3XObjectHierarchy_UnregisterClass`. If this is not done, `Q3XObjectHierarchy_UnregisterClass` returns `kQ3Failure` and the class remains registered.

You can also call `Q3XObjectHierarchy_UnregisterClass` to unregister a custom attribute type previously registered by the function `Q3AttributeClass_Register`.

SPECIAL CONSIDERATIONS

The best way to unload the class is by unloading the shared library, using the `Q3XSharedLibrary_Unregister` routine.

Q3ElementClass_Register

You can use the `Q3ElementClass_Register` function to register an application-defined element class.

```
TQ3ObjectClass Q3ElementClass_Register (
                    TQ3ElementType elementType,
                    const char *name,
                    unsigned long sizeOfElement,
                    TQ3MetaHandler metaHandler);
```

`elementType` An element type.

`name` A pointer to a null-terminated string containing the name of the element's creator and the name of the type of element being registered.

`sizeOfElement` The size of the data associated with the specified custom element type.

`metaHandler` A pointer to an application-defined metahandler that QuickDraw 3D calls to handle the new custom element type.

DESCRIPTION

The `Q3ElementClass_Register` function returns, as its function result, an object class reference for a new custom element type having a type specified by the `elementType` parameter and a name specified by the `name` parameter. The `metaHandler` parameter is a pointer to the metahandler for your custom element type. See "Defining an Object Metahandler," beginning on page 176 for information on writing a metahandler. If `Q3ElementClass_Register` cannot create a new element type, it returns the value `NULL`.

The `name` parameter should be a pointer to null-terminated C string that contains your (or your company's) name and the name of the type of element you are defining. Use the colon character (:) to delimit fields within this string. The string should not contain any spaces or punctuation other than the colon character, and it cannot end with a colon. Here are some examples of valid creator names:

```
"MyCompany:SurfDraw:Wavelength"
"MyCompany:SurfWorks:VRModule:WaterTemperature"
```

The `sizeOfElement` parameter specifies the fixed size of the data associated with your custom element type. If you wish to associate dynamically sized data with your element type, put a pointer to a dynamically sized block of data into the set and have your handler's copy method duplicate the data. (In this case, you would set the `sizeOfElement` parameter to `sizeof(Ptr)`.) You also need to have your handler's dispose method deallocate any dynamically sized blocks.

Q3ElementType_GetElementSize

You can use the `Q3ElementType_GetElementSize` function to get the size of an application-defined element type.

```
TQ3Status Q3ElementType_GetElementSize (
                    TQ3ElementType elementType,
                    unsigned long *sizeOfElement);
```

`elementType` An element type.

`sizeOfElement`
On exit, the number of bytes occupied by an element of the specified element object class.

DESCRIPTION

The `Q3ElementType_GetElementSize` function returns, in the `sizeOfElement` parameter, the number of bytes occupied by an element of the type specified by the `elementType` parameter.

Registering a Shared Library

QuickDraw 3D provides routines to register and unregister a shared library.

These routines let you provide an entry point for Windows dynamic link libraries or an initialization function for Mac OS shared libraries. Some libraries need to be loaded in a specific order; the registration mechanism lets QuickDraw 3D load libraries in the order that it needs to, and not be bound by

the order of shared libraries on disk (for example, in the Mac OS Extensions folder).

The `Q3XSharedLibrary_Register` routine notifies QuickDraw 3D that there is a library to be loaded and provides the entry point to the registration function. To use this function you need to fill out a `TQ3XSharedLibraryInfo` block, which provides information that the system will need to load the library.

Usually the `Q3XSharedLibrary_Register` function will be called from the shared library entry point. See the documentation for your development system, or the examples on the QuickDraw 3D SDK, for information about how to set this up. The `Q3XSharedLibrary_Unregister` function is used to unregister the library. It is usually called from the shared library termination routine. Windows programming is slightly different because a single `DLLMain` function is called with a selector for registration and unregistration.

The code in Listing 3-3 illustrates register and unregister functions for both Windows and Mac OS. It is taken from the sample renderer example on the QuickDraw 3D SDK. In the windows version there is a switch statement for registration and unloading, whereas in the Mac OS version these processes are handled by two distinct functions.

Listing 3-3 Library registering and unregistering

```
/* Mac OS registration & termination */

OSErr SR_Initialize(
    const CFragInitBlock    *initBlock)
{
    TQ3XSharedLibraryInfo   sharedLibraryInfo;
    OSErr           err = noErr;

    sharedLibraryInfo.registerFunction      = SR_Register;
    sharedLibraryInfo.sharedLibrary
                                = (unsigned long)initBlock->connectionID;

    Q3XSharedLibrary_Register(&sharedLibraryInfo);

    SRgSharedLibrary = (unsigned long)initBlock->connectionID;

    err = SR_CreateAliasHandle(initBlock);
```

```
    return (err);
}

TQ3Status SR_Exit(
    void)
{
    if (SRgSharedLibrary != NULL) {
        Q3XSharedLibrary_Unregister(SRgSharedLibrary);
        SRgSharedLibrary = NULL;
    }

    SR_FreeAliasHandle();

    return (kQ3Success);
}

/* Win32 extension entry point*/

HINSTANCE   hinstMyDLL = NULL;

BOOL WINAPI DllMain(
    HINSTANCE   hinstDLL,
    DWORD       fdwReason,
    LPVOID      lpvReserved)
{
    TQ3XSharedLibraryInfo   sharedLibraryInfo;

    if (fdwReason == DLL_PROCESS_ATTACH) {
        hinstMyDLL = hinstDLL;

        sharedLibraryInfo.registerFunction = SR_Register;
        sharedLibraryInfo.sharedLibrary = (unsigned long)hinstDLL;
        if (Q3XSharedLibrary_Register(&sharedLibraryInfo) == kQ3Success)
{
            return TRUE;
        } else {
            return FALSE;
        }
    }
```

```
    if (fdwReason == DLL_PROCESS_DETACH) {
        Q3XSharedLibrary_Unregister((unsigned long)hinstDLL);
    }

    return (TRUE);
}
```

Q3XSharedLibrary_Register

You can use the `Q3XSharedLibrary_Register` function to notify QuickDraw 3D that there is a library to be loaded and provide the entry point to the library's registration function.

```
typedef struct TQ3XSharedLibraryInfo {
    TQ3XSharedLibraryRegister   registerFunction;
    unsigned long               sharedLibrary;
} TQ3XSharedLibraryInfo;

TQ3Status Q3XSharedLibrary_Register(
                    TQ3XSharedLibraryInfo    *sharedLibraryInfo);
```

`sharedLibraryInfo`
Pointer to a struct of type `TQ3XSharedLibraryInfo`.

`sharedLibrary`
Entry point to the shared library.

DESCRIPTION

The `Q3XSharedLibrary_Register` function registers a shared library with QuickDraw 3D.

Q3XSharedLibrary_Unregister

You can use the `Q3XSharedLibrary_Unregister` function to unregister a QuickDraw 3D shared library.

```
TQ3Status Q3XSharedLibrary_Unregister(
                    unsigned long    sharedLibrary);
```

`sharedLibrary`
Entry point to the shared library.

DESCRIPTION

The `Q3XSharedLibrary_Unregister` function unregisters a QuickDraw 3D shared library whose entry point is designated by `sharedLibrary`.

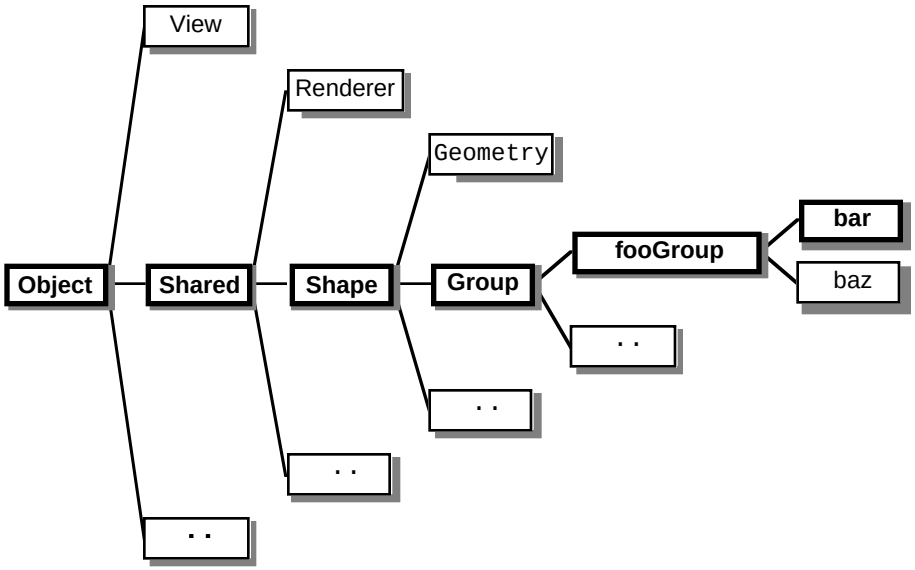
Creating a Hierarchy

Because you can make any class a subclass, you can create a hierarchy of classes. QuickDraw 3D represents the hierarchy as an acyclic graph with bidirectional links. This allows any child class to access its parent's class, and any parent class to track and access its child classes.

For example, the following three calls add the hierarchy shown in Figure 3-3 to the existing QuickDraw 3D hierarchy:

```
gFooGroupClass =
    Q3XObjectHierarchy_RegisterClass(
        kQ3ShapeTypeGroup,
        kGroupTypeFoo,
        ...);

gBarGroupClass =
    Q3XObjectHierarchy_RegisterClass(
        kGroupTypeFoo,
        kGroupTypeBar,
        ...);

gBazGroupClass =
    Q3XObjectHierarchy_RegisterClass(
        kGroupTypeFoo,
        kGroupTypeBaz,
        ...);
```

Figure 3-3 Sample object hierarchy



The depth of an object in a hierarchy is sometimes referred to as its *level.* In Figure 3-3, there are a total of 6 levels in the `bar` object class A single object class may contain several private data structures, one for each level in that particular class. Access to class data is restricted in that only the owner of the `TQX3ObjectClass` may access the class private data. A particular class may expose calls for another class to access the class data, or it may call the class methods. This is also true for each instance of each class.

Thus, in the example of Figure 3-3 the `bar` class contains private class data and private instance data for `Object`, `Shared`, `Shape`, `Group`, `fooGroup`, and `bar`. The `bar` class may access only the class methods and instance data at its level, and so on. Each class exposes calls such as the following to access the class or the instance data at its level:

`Q3Group_CountObjectsOfType`	(exposes a method)
`Q3Shape_AddElement`	(exposes a method)
`Q3Shared_Edited`	(exposes a method and alters the instance)
`Q3Object_GetType`	(exposes class data)

Object Methods

Every object in QuickDraw 3D contains object methods, which generally deal with name space information, allocation and deallocation, I/O processes, and submit routines. The `TQ3MethodType` type is declared as follows:

```
typedef unsigned long       TQ3MethodType;
```

The public methods are the following:

```
kQ3MethodTypeObjectClassVersion
kQ3MethodTypeObjectClassRegister
kQ3MethodTypeObjectClassReplace
kQ3MethodTypeObjectClassUnregister
kQ3MethodTypeObjectNew
kQ3MethodTypeObjectDelete
kQ3MethodTypeObjectCopy
kQ3MethodTypeObjectTraverseData
```

Multilevel Methods

A method which applies to every level in a hierarchy is called a **multilevel method.** Generally, only object methods are multilevel, though other aspects of QuickDraw 3D may use multilevel methods, if desired.

Many object methods apply to the private class structure or the private instance structure of a particular class. When an object class or object instance is created, a data structure for each level is allocated and initialized by each class. Creation occurs from root to leaf. If a failure occurs midway through creation, only those levels which were initialized are deleted.

The multi-level methods used in QuickDraw 3D are

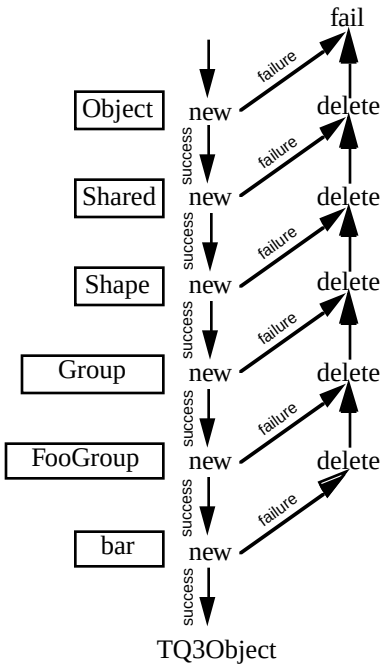
```
kQ3MethodTypeObjectClassRegister
kQ3MethodTypeObjectClassReplace
kQ3MethodTypeObjectClassUnregister
kQ3MethodTypeObjectNew
kQ3MethodTypeObjectDelete
kQ3MethodTypeObjectDuplicate
kQ3MethodTypeObjectTraverseData
```

IMPORTANT

These methods should never be returned in a virtual metahandler, because they always apply to a single level in a particular object class. ▲

Figure 3-4 illustrates how the multilevel methods `_New` and `_Delete` can be used to create the `bar` object hierarchy.

Figure 3-4 Object creation using multilevel methods



Class Routines

This section describes the QuickDraw 3D routines you can use with object classes.

Instantiating an Object

To instantiate an object of a class, the QuickDraw 3D object system calls `Q3XObjectHierarchy_NewObject`.

Q3XObjectHierarchy_NewObject

The `Q3XObjectHierarchy_NewObject` routine instantiates an object in a class.

```
TQ3XObject Q3XObjectHierarchy_NewObject(
                    TQX3ObjectClass    objectClass,
                    void               *parameters);
```

`objectClass` An object class.

`parameters` Pointer to parameters to be passed.

DESCRIPTION

The `Q3XObjectHierarchy_NewObject` routine begins the QuickDraw 3D object creation mechanism. The parameters pointed to by `parameters` are passed into the `TQ3ObjectNewMethod` method at each level.

EXAMPLE

To initialize multiple levels of data, organize the data structure into multiple levels as illustrated below:

```
typedef struct TFooGroupData {
    float               dummy1;
} TFooGroupData;

typedef struct TBarGroupData {
    TFooGroupData       fooData;
    float               dummy2;
} TBarGroupData;
```

This way, the new method for the `fooGroup` class receives a `TFooGroupData` parameter and subclassses receive initialization parameters.

Accessing Types in a Class

QuickDraw 3D provides routines for accessing various object types. The types are defined as follows:

```
typedef struct TQ3ObjectClassPrivate    *TQ3XObjectClass;
```

Q3XObjectClass_GetType

You can use the `Q3XObjectClass_GetType` function to get the type, given a reference to a class. This is most useful in the instance where you register an element or attribute and need to get the type. When you register an element, QuickDraw 3D will take the type you pass in and modify it, to avoid name conflicts. Many object system calls require an object type; this function lets you get the type from the class reference that you ordinarily store when you register a class.

```
TQ3Status Q3XObjectClass_GetType(
                    TQ3XObjectClass    objectClass,
                    TQ3ObjectType      *type);
```

`objectClass` A class.

`type` On return, an object type.

DESCRIPTION

The `Q3XObjectClass_GetType` function returns, in the `type` parameter, the type of the class referenced by `objectClass`.

Q3XObjectClass_GetLeafType

The `Q3XObjectClass_GetLeafType` function lets you determine the leaf type of a class.

```
TQ3ObjectType Q3XObjectClass_GetLeafType(
                    TQX3ObjectClass    objectClass);
```

`objectClass` An object class.

DESCRIPTION

The `Q3XObjectClass_GetLeafType` function returns the leaf type of a class. If an error occurs, it returns `kQ3ObjectTypeInvalid` and posts an error.

Q3XObjectClass_GetSubClassType

The `Q3XObjectClass_GetSubClassType` function lets you determine the subclass type of one object class relative to another object class.

```
TQ3ObjectType Q3XObjectClass_GetSubClassType(
                    TQX3ObjectClass    objectClass,
                    TQX3ObjectClass    targetObjectClass);
```

`objectClass` First object class.

`targetObjectClass`
Second object class.

DESCRIPTION

The `Q3XObjectClass_GetSubClassType` function is used for `_GetType` calls in a particular class. For example, `Q3Geometry_GetType` would be implemented

```
TQ3ObjectType Q3Geometry_GetType (TQ3GeometryObject object)
    {
    return Q3XObject_GetSubClassType (gGeometryClass, object);
    }
```

where `gGeometryClass` is the geometry object class, and `object` is a subclass of the geometry class. The type returned is the subclass type of the geometry.

If an error occurs, the `Q3XObjectClass_GetSubClassType` function returns `kQ3ObjectTypeInvalid` and posts an error.

Q3XObject_GetClass

You can use the `Q3XObject_GetClass` function to get the class of an object.

```
TQ3XObjectClass Q3XObject_GetClass(
                    TQ3Object    object);
```

`object` An object.

DESCRIPTION

The `Q3XObject_GetClass` function returns the class of the object designated by `object`.

Q3XObject_GetSubClassType

The `Q3XObject_GetSubClassType` function lets you determine the subclass type of an object relative to an object class.

```
TQ3ObjectType Q3XObject_GetSubClassType(
                    TQ3XObjectClass    objectClass,
                    TQ3XObject          targetObject);
```

`objectClass` An object class.

`targetObject` An object.

DESCRIPTION

Use of `Q3XObject_GetSubClassType` resembles `Q3XObjectClass_GetSubClassType` (page 217), except it is used for an object relative to an object class instead of for two object classes.

Version Checking

The `Q3XObjectHierarchy_GetClassVersion` function lets you check the version number of a custom class.

Q3XObjectHierarchy_GetClassVersion

You can use the `Q3XObjectHierarchy_GetClassVersion` function to get the version number of a class type.

```
TQ3Status Q3XObjectHierarchy_GetClassVersion(
                    TQ3ObjectType            objectClassType,
                    TQ3XObjectClassVersion   *version);
```

`objectClassType`
A class type.

`version` On return, a version number.

DESCRIPTION

The `Q3XObjectHierarchy_GetClassVersion` function returns, in the `version` parameter, the version number of the class type designated by `objectClassType`.

QuickDraw 3D includes two utility macros that let you obtain the version numbers of a class.

```
#define Q3_OBJECT_CLASS_GET_MAJOR_VERSION(version)
                                (unsigned long) ((version) >> 16)

#define Q3_OBJECT_CLASS_GET_MINOR_VERSION(version)
                                (unsigned long) ((version) & 0x0000FFFF)
```

These are convenience macros that unpack a version, accessing its major and minor version numbers.

SEE ALSO

"Q3XMethodTypeObjectClassVersion" (page 228)

Class Method Retrieval

A class should retrieve the methods passed in via the metahandler during the registration process, using the `Q3ObjectHierarchy_GetMethod` object system call.

Q3XObjectHierarchy_GetMethod

Repeated calls to the `Q3XObjectHierarchy_GetMethod` function return all the methods for a class hierarchy.

```
TQ3FunctionPointer Q3XObjectHierarchy_GetMethod(
                    TQX3ObjectClass    objectClass,
                    TQ3MethodType       methodType);
```

`objectClass` An object class.

`methodType` A method type.

DESCRIPTION

The `Q3XObjectHierarchy_GetMethod` routine searches for methods, starting from the leaf class and continuing with the parent classes, for a non-`NULL` method type from a class. If the leaf class returns `NULL` for the method, its virtual metahandler is called to retrieve a method. This continues with the parent class's virtual metahandler and on up the hierarchy. When no method is found, `Q3XObjectHierarchy_GetMethod` returns `NULL`.

Q3XObjectClass_GetMethod

You can use the `Q3XObjectClass_GetMethod` function to get the methods for a class.

```
TQ3XFunctionPointer Q3XObjectClass_GetMethod(
                    TQ3XObjectClass    objectClass,
                    TQ3XMethodType     methodType);
```

`objectClass` A class.

`methodType` On return, a method type.

DESCRIPTION

The `Q3XObjectClass_GetMethod` function returns, in the `methodType` parameter, a method type for the class `objectClass`. The `Q3XObjectClass_GetMethod` function

searches for non-NULL methods starting from the leaf class and continuing with the parent classes. If the leaf class returns NULL for the method, its virtual metahandler is called to retrieve a method. This continues with the parent class's virtual metahandler, and on up the hierarchy. If no method is found, Q3XObjectClass_GetMethod returns NULL.

Accessing Private Data

You may access private data at any time in a class or object with the routines described in this section.

Q3XObjectClass_GetPrivate

You can use the Q3XObjectClass_GetPrivate routine to get the private instance data of an object.

```
void *Q3XObjectClass_GetPrivate(
                    TQ3XObjectClass    objectClass,
                    TQ3Object          targetObject);
```

objectClass	A class.
targetObject	An object
return value	The class's private data block.

DESCRIPTION

The Q3XObjectClass_GetPrivate routine returns a pointer to a block of instanceSize bytes (where instanceSize is from the objectClass class's previous Q3XObjectHierarchy_RegisterClass call) that contains the private instance data of targetObject. Q3XObjectClass_GetPrivate returns NULL if instanceSize was 0.

The Q3XObjectClass_GetPrivate routine may return NULL if an invalid object or object of the wrong type is passed in, if instanceSize or classSize is 0 in the previous Q3ObjectHierarchy_Register call, or if an invalid target is passed in.

EXAMPLE

```
TQ3Status Q3FooGroup_SetDummy(
    TQ3GroupObject          group,
    float                   dummy)
{
    TFooGroupPrivate *gPriv;

    gPriv = Q3XObjectClass_GetPrivate(
            gFooGroupClass,
            group);
    if (gPriv == NULL)
        return kQ3Failure;
    gPriv->dummy = dummy;
    return Q3Shared_Edited(group);
}
```

Q3XObject_GetClassPrivate

You can use the `Q3XObject_GetClassPrivate` routine to get private class data from an object.

```
void *Q3XObject_GetClassPrivate(
                    TQ3XObjectClass    objectClass,
                    TQ3Object          targetObject);
```

`objectClass` A class.

`targetObject` An object

return value The class's private data block.

DESCRIPTION

The `Q3XObject_GetClassPrivate` routine returns a pointer to a block of `instanceSize` bytes (where `instanceSize` is from the `objectClass` class's previous `Q3XObjectHierarchy_RegisterClass` call) that contains the private class data of `targetObject`. If `instanceSize` was 0, `Q3XObject_GetClassPrivate` returns `NULL`.

The Q3XObject_GetClassPrivate routine may return NULL if an invalid object or object of the wrong type is passed in, if instanceSize or classSize is 0 in the previous Q3ObjectHierarchy_Register call, or if an invalid target is passed in.

Q3XObjectClass_GetClassPrivate

You can use the Q3XObjectClass_GetClassPrivate routine to get private class data from a class.

```
void *Q3XObjectClass_GetClassPrivate(
                    TQ3XObjectClass    objectClass,
                    TQ3XObjectClass    targetObjectClass);
```

`objectClass` A class.

`targetObjectClass`
A class.

return value The target class's private data block.

DESCRIPTION

The Q3XObjectClass_GetClassPrivate routine returns a pointer to a block of instanceSize bytes (where instanceSize is from the objectClass class's previous Q3XObjectHierarchy_RegisterClass call) that contains the private data of targetObject. If instanceSize was 0, Q3XObjectClass_GetClassPrivate returns NULL.

The Q3XObjectClass_GetClassPrivate routine may return NULL if an invalid object or object of the wrong type is passed in, if instanceSize or classSize is 0 in the previous Q3ObjectHierarchy_Register call, or if an invalid target is passed in.

Class Methods

This section describes the methods that custom QuickDraw 3D objects should contain.

Class Registration and Unregistration

Custom objects should provide methods for registering, unregistering, and replacing classes.

Listing 3-4 provides an example of how these methods are called. It is based on the example discussed in “Creating a Hierarchy” (page 211).

Listing 3-4 Sample of registering and unregistering classes

```
#define kMethodTypeFooGroupDoSomething          \
                            Q3_METHOD_TYPE(0xFE, 'f','g','r')

typedef TQ3Status (*TFooGroupDoSomethingMethod)(
    TQ3ObjectClass              objectClass,
    TQ3Object                   object,
    float                       *dummyArg);

typedef struct TFooGroupClass {
    TFooGroupDoSomethingMethod          doSomething;
} TFooGroupClass;

typedef struct TFooGroupData {
    float                           dummy1;
} TFooGroupData;

gFooGroupClass =
    Q3XObjectHierarchy_RegisterClass(
        kQ3ShapeTypeGroup,
        kGroupTypeFoo,
        "SomeCompany:FooGroup",
        FooGroupClass_MetaHandler,
        FooGroupClass_VirtualMetaHandler,
        sizeof(TFooGroupClass),
        sizeof(TFooGroupData));
```

The registration method for the foregoing would look like this:

```
static TQ3Status FooGroupClass_Register(
    TQ3ObjectClass                  objectClass,
    TFooGroupClass                  *gClass)
```

```
{
    gClass->doSomething =
        Q3XObjectHierarchy_GetMethod(
            objectClass,
            kMethodTypeFooGroupDoSomething);

    if ((Q3XObjectClass_GetLeafType(objectClass) !=
            kGroupTypeFoo) &&
            (gClass->doSomething == NULL)) {
        return kQ3Failure;
    }

    return kQ3Success;
}
```

When the parent class `FooGroup` is registered in this example, there is no need for the `kMethodTypeGroupFoo` method. In subclasses, however, this method is required. This type of strategy may be used to impose restrictions on subclasses only, especially when the parent class is never intended to be instantiated.

There is also no need for an `ObjectClassUnregister` method because the `ObjectClassRegister` method does not allocate any data.

TQ3XObjectClassRegisterMethod

The `TQ3XObjectClassRegisterMethod` function, which is returned by the `kQ3XMethodTypeObjectClassRegister` method, registers a class.

```
#define kQ3XMethodTypeObjectClassRegister Q3_METHOD_TYPE('r','g','s','t')

typedef TQ3Status (*TQ3XObjectClassRegisterMethod)(
                    TQ3XObjectClass    objectClass,
                    void               *classPrivate);
```

`objectClass` An object class.

`classPrivate` A pointer to the class's private data.

DESCRIPTION

The `TQ3XObjectClassRegisterMethod` method registers the class designated by `objectClass`, with private data pointed to by `classPrivate`. The size of the private data is equivalent to the `methodsSize` parameter used in the earlier `Q3XObjectHierarchy_RegisterClass` call; if `methodsSize` was 0, `classPrivate` is `NULL`.

`ObjectClassRegister` is called from `Q3ObjectHierarchy_RegisterClass` upon initial registration of an object class. It is also called when any subclass of an object class is registered. Registration occurs from root to leaf, as shown in Figure 3-4. The `ObjectClassRegister` method is called for the private class data on the way down the hierarchy, and the `ObjectClassUnregister` method is called on the way up in case of a failure.

`ObjectClassRegister` should initialize the data in `classPrivate` and collect any needed methods from the metahandler, using `Q3ObjectHierarchy_GetMethod`. A class may have no private class data (that is, its `methodsSize` may be 0), yet may still have an `ObjectClassRegister` method. In this case, `classPrivate` will be `NULL` (since there is no data), and a particular class could instead keep track of subclass states in global variables.

TQ3XObjectClassUnregisterMethod

The `TQ3XObjectClassUnregisterMethod` function, which is returned by the `kQ3XMethodTypeObjectClassUnregister` method, unregisters a class.

```
#define kQ3XMethodTypeObjectClassUnregister
                    Q3_METHOD_TYPE('u','n','r','g')

typedef void (*TQ3XObjectClassUnregisterMethod)(
                    TQ3XObjectClass    objectClass,
                    void               *classPrivate);
```

`objectClass` An object class.

`classPrivate` A pointer to the class's private data.

DESCRIPTION

The `TQ3XObjectClassUnregisterMethod` method unregisters the class designated by `objectClass`, which has private data pointed to by `classPrivate`. The size of the private data is equivalent to the `methodsSize` parameter used in the earlier `Q3XObjectHierarchy_RegisterClass` call; if `methodsSize` was 0, `classPrivate` is `NULL`.

The `ObjectClassUnregister` method should undo any operations performed in the `ObjectClassRegister` method, including removing the class from global tables and deallocating any memory used to store the class private data. If the `ObjectClassRegister` method performed no allocations, `ObjectClassUnregister` may be `NULL`.

TQ3XObjectClassReplaceMethod

The `TQ3XObjectClassReplaceMethod` function, which is returned by the `kQ3XMethodTypeObjectClassReplace` method, replaces one class with another. It is used only when a new version of an object class is registered, eliminating the old version.

```
#define kQ3XMethodTypeObjectClassReplace Q3_METHOD_TYPE('r','g','r','p')

typedef void (*TQ3XObjectClassReplaceMethod)(
                    TQ3XObjectClass    oldObjectClass,
                    void               *oldClassPrivate,
                    TQ3XObjectClass    newObjectClass,
                    void               *newClassPrivate);
```

`oldObjectClass`
The old object class to be replaced.

`oldClassPrivate`
A pointer to the old class's private data.

`newObjectClass`
A new object class.

`newClassPrivate`
A pointer to the new class's private data.

DESCRIPTION

The `TQ3XObjectClassReplaceMethod` method replaces the class designated by `oldObjectClass`, which has private data pointed to by `oldClassPrivate`, with the class designated by `newObjectClass`, which has private data pointed to by `newClassPrivate`. The sizes of the private data areas are equivalent to the `methodsSize` parameters used in the earlier `Q3XObjectHierarchy_RegisterClass` calls; if a `methodsSize` value was 0, its equivalent `classPrivate` value is `NULL`.

`TQ3XObjectClassReplaceMethod` is required only by classes that maintain or track their subclasses in a table. When object classes of the same type collide, use the replace method instead of calling `Unregister(oldClass, oldClassPrivate)` followed by `Register(newClass, newClassPrivate)`, which may cause an unexpected failure.

The replace method should register the new class and then unregister the old class, without failure. If a class's `TQ3XObjectClassRegisterMethod` method never fails, the replace method is not needed.

SEE ALSO

"Q3XMethodTypeObjectClassVersion" (page 228).

Class Version

The `kQ3XMethodTypeObjectClassVersion` method lets you publish the version number of a custom class.

Q3XMethodTypeObjectClassVersion

The `kQ3XMethodTypeObjectClassVersion` method returns the version of a class as a `TQ3XObjectClassVersion` type. This information may be used to determine when to invoke the `TQ3XObjectClassReplaceMethod` method.

```
#define kQ3XMethodTypeObjectClassVersion Q3_METHOD_TYPE('v','r','s','n')

typedef unsigned long       TQ3XObjectClassVersion;
```

`TQ3XObjectClassVersion`
Version of a class.

DESCRIPTION

QuickDraw 3D includes a utility macro that lets you provide the version number of a class. If there are two identical implementations of a class, the system will only load the latter, as determined by the version number.

```
#define Q3_OBJECT_CLASS_VERSION(major, minor)
                            (unsigned long) (((major) << 16) | (minor))
```

▲ **WARNING**
If you do not provide a version number the version is automatically set to 0.0. ▲

SEE ALSO

"Q3XObjectHierarchy_GetClassVersion" (page 219)

Object Creation and Deletion

Object creation and deletion is similar to object registration, except that the data being operated on is the instance data. The `TQ3XObjectNewMethod` method should initialize all data in the private data structure and allocate any memory needed to copy the data in. The `TQ3XObjectDeleteMethod` method should deallocate any data in the private data structure of the object.

TQ3XObjectNewMethod

The `TQ3XObjectNewMethod` function, returned by the `kQ3XMethodTypeObjectNew` method, initializes data in the object's private data structure and allocates the required memory.

```
#define kQ3XMethodTypeObjectNew Q3_METHOD_TYPE('n','e','w','o')
```

```
typedef TQ3Status (*TQ3XObjectNewMethod)(
                    TQ3Object    object,
                    void         *privateData,
                    void         *parameters);
```

`object` An object.

`privateData` Pointer to the object's private data.

`parameters` Pointer to parameters to be passed.

DESCRIPTION

The `TQ3XObjectNewMethod` method should initialize all data in the private data structure (possibly with parameters) and allocate any memory needed to copy the data in. If `instanceSize` in the previous `Q3ObjectHierarchy_RegisterClass` call was nonzero, a `TQ3ObjectNewMethod` is required. If `instanceSize` was 0, the `TQ3XObjectNewMethod` method is never called.

TQ3XObjectDeleteMethod

The `TQ3XObjectDeleteMethod` function, which is returned by the `kQ3XMethodTypeObjectDelete` method, deallocates data in the object's private data structure.

```
#define kQ3XMethodTypeObjectDelete Q3_METHOD_TYPE('d','l','t','e')

typedef void (*TQ3XObjectDeleteMethod)(
              TQ3Object    object,
              void         *privateData);
```

`object` An object.

`privateData` Pointer to the object's private data.

DESCRIPTION

The `TQ3XObjectDeleteMethod` method deallocates any data in the private data structure of the object. If `instanceSize` in the `Q3ObjectHierarchy_RegisterClass` call was nonzero, a `TQ3XObjectDeleteMethod` is required. If `instanceSize` was 0, the `TQ3XObjectDeleteMethod` method is never called.

TQ3XObjectDuplicateMethod

The `TQ3XObjectDuplicateMethod` function, which is returned by the `kQ3XMethodTypeObjectDuplicate` method, duplicates an object and copies its private instance data.

```
#define kQ3XMethodTypeObjectDuplicate Q3_METHOD_TYPE('d','u','p','l')

typedef TQ3Status (*TQ3XObjectDuplicateMethod)(
                    TQ3Object     fromObject,
                    const void    *fromPrivateData,
                    TQ3Object     toObject,
                    const void    *toPrivateData);
```

`fromObject` Object to be copied.

`fromPrivateData`
Pointer to private data of object to be copied.

`toObject` Object to be copied into.

`toPrivateData`
Pointer to private data of object to be copied into.

DESCRIPTION

The `TQ3XObjectDuplicateMethod` method should copy the private instance data from `fromPrivateData` to `toPrivateData` and return `kQ3Success` if sucessful. Otherwise, it should deallocate anything it has allocated, clean up its parent classes, and return `kQ3Failure`. `TQ3XObjectDuplicateMethod` is called in the same way as `TQ3XObjectNewMethod` and `TQ3XObjectDeleteMethod`.

EXAMPLE

```
TQ3Status Q3FooGroup_Duplicate(
    TQ3GroupObject              src,
    TFooGroupPrivate            *srcPriv,
    TQ3GroupObject              dst,
    TFooGroupPrivate            *dstPriv)
```

```
{
    *dstPriv = *srcPriv;
    return kQ3Success;
}
```

TQ3XObjectUnregisterMethod

The `TQ3XObjectUnregisterMethod` function, which is returned by the `kQ3MethodTypeObjectUnregister` method, removes a custom object class.

```
#define kQ3MethodTypeObjectUnregister Q3_METHOD_TYPE('u','n','r','g')

typedef TQ3Status (*TQ3XObjectUnregisterMethod)
                    (TQ3XObjectClass    objectClass);
```

`objectClass` An object class.

DESCRIPTION

The `TQ3XObjectUnregisterMethod` function unregisters the custom object class specified by the `objectClass` parameter.

Shared Objects

A custom class uses the `TQ3XSharedLibraryRegister` type for library sharing.

TQ3XSharedLibraryRegister

The `TQ3XSharedLibraryRegister` type defines the shared library registration function for a custom class.

```
typedef struct TQ3XSharedLibraryInfo {
    TQ3XSharedLibraryRegister   registerFunction;
    unsigned long               sharedLibrary;
} TQ3XSharedLibraryInfo;
```

```
typedef TQ3Status (*TQ3XSharedLibraryRegister) (void);
```

DESCRIPTION

See “Registering a Shared Library,” beginning on page 207.

I/O Methods

A custom object may include these methods for file access:

```
#define kQ3XMethodTypeObjectTraverse Q3_METHOD_TYPE('t','r','v','s')
#define kQ3XMethodTypeObjectTraverseData Q3_METHOD_TYPE('t','r','v','d')
#define kQ3XMethodTypeObjectWrite Q3_METHOD_TYPE('w','r','i','t')
#define kQ3XMethodTypeObjectReadData Q3_METHOD_TYPE('r','d','d','t')
#define kQ3XMethodTypeObjectRead Q3_METHOD_TYPE('r','e','a','d')
#define kQ3XMethodTypeObjectAttach Q3_METHOD_TYPE('a','t','t','c')
```

The operation of some of these methods is discussed in the chapter “File Objects” under the headings shown:

- `Q3XMethodTypeObjectTraverse` and `Q3XMethodTypeObjectTraverseData`: “Writing to Custom File Objects” (page 1090)
- `Q3XMethodTypeObjectWrite`: “Writing to Custom File Objects” (page 1090)
- `Q3XMethodTypeObjectReadData`: “Reading and Writing File Data” (page 1045)
- `Q3XMethodTypeObjectRead`: “Reading and Writing File Data” (page 1045)

The `Q3XMethodTypeObjectAttach` method is described below.

TQ3XObjectAttachMethod

The `TQ3XObjectAttachMethod` function, which is returned by the `kQ3XMethodTypeObjectAttach` method, attaches a child object to a parent object for traversal and other I/O operations.

```
#define kQ3XMethodTypeObjectAttach Q3_METHOD_TYPE('a','t','t','c')
```

```
typedef TQ3Status (*TQ3XObjectAttachMethod)(
                    TQ3Object    childObject,
                    TQ3Object    parentObject);
```

`childObject` An object that is to be attached as a child.

`parentObject` An object that is to be attached as a parent.

DESCRIPTION

The `TQ3XObjectAttachMethod` method attaches `childObject` to `parentObject` as child to parent.

Object Errors, Warnings, and Notices

The following is a list of errors, warnings, and notices that object routines can return. A list of general QuickDraw 3D errors is given in "QuickDraw 3D Errors, Warnings, and Notices" (page 87).

```
kQ3ErrorInvalidObject
kQ3ErrorInvalidObjectClass
kQ3ErrorInvalidObjectType
kQ3ErrorInvalidObjectName
kQ3ErrorObjectClassInUse
kQ3ErrorAccessRestricted
kQ3ErrorMetaHandlerRequired
kQ3ErrorNeedRequiredMethods
kQ3ErrorNoSubClassType
kQ3ErrorUnknownElementType
kQ3ErrorNotSupported
kQ3ErrorTypeAlreadyExistsAndHasSubclasses
kQ3ErrorTypeAlreadyExistsAndOtherClassesDependOnIt
kQ3ErrorTypeAlreadyExistsAndHasObjectInstances
kQ3WarningNoObjectSupportForDuplicateMethod
kQ3WarningNoObjectSupportForDrawMethod
kQ3WarningNoObjectSupportForWriteMethod
kQ3WarningNoObjectSupportForReadMethod
kQ3WarningUnknownElementType
kQ3WarningTypeAndMethodAlreadyDefined
```

```
kQ3WarningTypeIsOutOfRange
kQ3WarningTypeHasNotBeenRegistered
kQ3WarningTypeAlreadyRegistered
kQ3WarningTypeSameVersionAlreadyRegistered
kQ3WarningTypeNewerVersionAlreadyRegistered
kQ3WarningInvalidObjectInGroupMetafile
kQ3NoticeObjectAlreadySet
kQ3NoticeMethodNotSupported
```

CHAPTER 3

QuickDraw 3D Objects

CHAPTER 4

Geometric Objects

This chapter describes the QuickDraw 3D geometric objects and the functions you can use to manipulate them. Geometric objects form the basis of any three-dimensional model, so you need to know how to define (and perhaps also create and dispose of) geometric objects to render any image. QuickDraw 3D provides a rich set of geometric primitive objects, which you can group, copy, illuminate, texture, or otherwise modify as desired.

To use this chapter, you should already be familiar with the QuickDraw 3D class hierarchy, described in the chapter "QuickDraw 3D Objects." earlier in this book.

This chapter begins by describing the QuickDraw 3D geometric primitives. Then it shows how to create and manipulate instances of those primitives. The section "Geometric Objects Reference," beginning on page 275 provides a complete description of the geometric primitives and the routines you can use to create and manipulate them.

This chapter also provides definitions of the fundamental mathematical objects (points, vectors, matrices, quaternions, and so forth) that are used in defining QuickDraw 3D geometric objects. For routines that you can use to manipulate those basic mathematical objects, see the chapter "Mathematical Utilities." For routines that you can use to group geometric primitive objects into groups or collections, see the chapter "Group Objects" later in this book.

About Geometric Objects

A **geometric object** (or a **geometry**) is an instance of the `TQ3GeometryObject` class. The `TQ3GeometryObject` class is a subclass of the `TQ3ShapeObject`, which is itself a subclass of the `TQ3SharedObject` class. As a result, a geometric object is

associated with a reference count, which is incremented or decremented whenever you create or dispose of an instance of that type of object.

Currently, QuickDraw 3D provides many types of primitive geometric objects. A geometric object has one of these types:

```
kQ3GeometryTypeBox
kQ3GeometryTypeCone
kQ3GeometryTypeCylinder
kQ3GeometryTypeDisk
kQ3GeometryTypeEllipse
kQ3GeometryTypeEllipsoid
kQ3GeometryTypeGeneralPolygon
kQ3GeometryTypeLine
kQ3GeometryTypeMarker
kQ3GeometryTypeMesh
kQ3GeometryTypeNURBCurve
kQ3GeometryTypeNURBPatch
kQ3GeometryTypePixmapMarker
kQ3GeometryTypePoint
kQ3GeometryTypePolygon
kQ3GeometryTypePolyhedron
kQ3GeometryTypePolyLine
kQ3GeometryTypeTorus
kQ3GeometryTypeTriangle
kQ3GeometryTypeTriGrid
kQ3GeometryTypeTriMesh
```

These objects are described in detail later in this chapter, beginning on page 282. In most cases, the definitions of these objects are simple and obvious. For instance, a triangle is just a closed plane figure defined by three points, or vertices, in space. A simple polygon (object type `kQ3GeometryTypePolygon`) is a closed plane figure defined by a list of vertices. Only six of these types of geometric primitives—meshes, trimeshes, trigrids, polyhedra, NURB curves, and NURB patches—need special discussion. See "Polyhedral Primitives," beginning on page 240 for a description of meshes, trimeshes, trigrids, and polyhedra. See "NURB Curves and Patches," beginning on page 248 for a description of NURB curves and patches.

Note
You can determine a geometric object's type by calling the `Q3Geometry_GetType` function, described on page 331. ◆

QuickDraw 3D geometric objects are opaque. This means that you can edit the data associated with an object only by calling accessor functions provided by QuickDraw 3D. For instance, once you've created a triangle, you can alter its shape or position only indirectly, for example by calling the functions `Q3Triangle_GetVertexPosition` and `Q3Triangle_SetVertexPosition`.

Attributes of Geometric Objects

Every QuickDraw 3D geometric object can contain one or more optional sets of attributes, which define characteristics of all or part of the object, such as its color or other material properties. For example, QuickDraw 3D defines the data associated with a triangle like this:

```
typedef struct TQ3TriangleData {
    TQ3Vertex3D                 vertices[3];
    TQ3AttributeSet             triangleAttributeSet;
} TQ3TriangleData;
```

As you can see, the triangle data consists of three vertices that define the triangle's position, together with a set of attributes that specify characteristics of the planar area enclosed by the lines connecting those vertices. A set of attributes is simply a collection of attributes, each of which consists of an attribute type and its associated data. Some common attribute types are diffuse color, specular color, surface normal vector, transparency, and so forth. You can, if you wish, define your own custom types of attributes and include them in attribute sets like any other kind of attribute. See the chapter "Attribute Objects" for complete information on the types of attributes defined by QuickDraw 3D and on defining custom attribute types.

You can associate a set of attributes with most parts of a geometric object. For example, you can associate a set of attributes with the face of a triangle or with one or more of the triangle's vertices. Similarly, a box can have an attribute set for the entire box as well as an attributes set for each of the six faces of the box. In this way, you can assign different colors to each of the box faces. Accordingly, QuickDraw 3D defines the data associated with a box like this:

```
typedef struct TQ3BoxData {
    TQ3Point3D                  origin;
    TQ3Vector3D                 orientation;
    TQ3Vector3D                 majorAxis;
    TQ3Vector3D                 minorAxis;
    TQ3AttributeSet             *faceAttributeSet;
    TQ3AttributeSet             boxAttributeSet;
} TQ3BoxData;
```

The `boxAttributeSet` field is a set of attributes that apply to the entire box, and the `faceAttributeSet` field is a pointer to an array of attribute sets that apply to the six faces of the box.

Trimeshes do not use attribute sets. See "Trimeshes" (page 246) for information on specifying attributes for a trimesh and its parts.

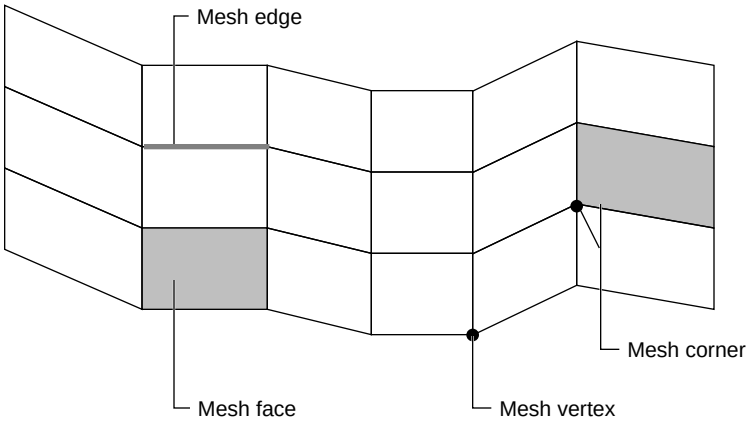
Polyhedral Primitives

QuickDraw 3D provides four basic **polyhedral primitives,** three-dimensional surfaces composed of polygonal faces that share edges and vertices with other faces. These are the mesh, the trimesh, the trigrid, and the polyhedron. Although you can use each of these primitives to represent the same sorts of shapes, there are important differences in their memory use, ease of definition, flexibility, and other features. This section describes the four polyhedral primitives individually. Then it compares their strengths and weaknesses (in "Comparison of the Polyhedral Primitives," beginning on page 247). See "Using Geometric Objects," beginning on page 257 for code samples that show how to construct several different polyhedral primitives.

Meshes

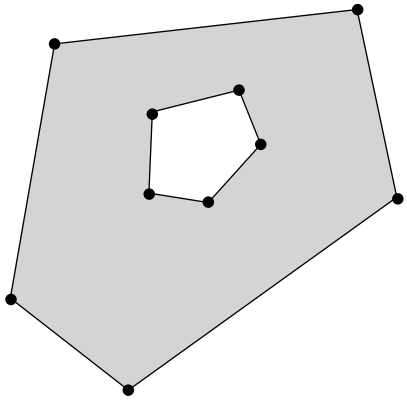
A **mesh** is a collection of vertices, faces, and edges that represents a topological polyhedron (that is, a solid figure composed of polygonal faces). The polyhedra represented by QuickDraw 3D meshes do not need to be closed, so that the meshes may have boundaries. Figure 4-1 illustrates a mesh.

Figure 4-1 A mesh



A **mesh face** is a polygonal figure that forms part of the surface of the mesh. QuickDraw 3D does not require mesh faces to be planar, but you can obtain unexpected results when rendering nonplanar mesh faces with a filled style. In addition, a mesh face can contain holes, as shown in Figure 4-2.

Figure 4-2 A mesh face with a hole



A mesh face is defined by a list of **mesh vertices.** The ordering of the vertices is unimportant; you can list the vertices of a mesh face in either clockwise or counterclockwise order. QuickDraw 3D internally attempts to maintain a consistent ordering of the vertices of all the faces of a mesh.

Because of their potential complexity, QuickDraw 3D treats meshes differently than it treats all other basic geometric objects. Usually, you create a basic geometric object by filling in a public data structure that completely specifies that object (for example, a structure of type `TQ3TriangleData`) and then by passing that structure to the appropriate object-creating routine (for example, `Q3Triangle_New`). To create a mesh, however, you first create a new empty mesh (by calling `Q3Mesh_New`), and then you explicitly add vertices and faces to the mesh (by calling `Q3Mesh_VertexNew` and `Q3Mesh_FaceNew`).

Note
Although you can manipulate an edge in a mesh (for instance, assign an attribute set to it), you cannot explicitly add an edge to a mesh. Mesh edges are implicitly created or destroyed when the faces containing them are created or destroyed. ◆

Because you can dynamically add or remove faces and vertices in a mesh, a mesh is always a retained object (that is, QuickDraw 3D maintains the mesh data internally) and never an immediate object. As a result, QuickDraw 3D does not supply routines to submit or write meshes in immediate mode. QuickDraw 3D builds an internal data structure that records the topology of a mesh (that is, the edge connections between all the faces and vertices in the mesh). For large models, this might require a large amount of memory. If your application does not need to use the topological information maintained by QuickDraw 3D (which you access by calling mesh iterator functions), you might want to use a trigrid or polyhedron (or a number of triangles, or a number of simple or general polygons) to represent a large number of interconnected polygons.

Note
See "Traversing Mesh Components, Vertices, Faces, and Edges," beginning on page 410, for information on the mesh iterator functions. ◆

As you've seen, a face of a mesh can contain one or more holes. A hole is defined by a **contour,** which is just a list of vertices. You create a contour in a mesh face by creating a face that contains the vertices in the contour (by calling

`Q3Mesh_FaceNew`) and then by converting the face into a contour (by calling `Q3Mesh_FaceToContour`). For optimal results, the face that contains the contour (called the **container face**) and the contour itself should be coplanar. In addition, the contour should lie entirely within the container face.

Note
See "Creating a Mesh," beginning on page 270 for sample code that creates a mesh. ◆

The geometric structure of a mesh is completely defined by its faces, vertices, edges, and contours. For purposes of shading and picking, QuickDraw 3D defines several other parts of a mesh: corners, mesh parts, and components. A **mesh corner** (or a **corner**) is specified by a mesh face together with one of its vertices. (A face with five vertices therefore has five corners.) You can associate a set of attributes with each corner. The attributes in a corner override any existing attributes of the associated vertex. For example, you can use corners to achieve special shading effects, such as hard edges when applying a smooth shading to a mesh. When a face is being shaded smoothly, the normals used to determine the amount of shading are the normals of the face's vertices. Because a vertex and its normal may be associated with several faces, the light intensity computed by a shading algorithm is the same for all points around that vertex. As a result, the edges between appear smooth. To get a hard edge, you can assign different normals to the corners on opposite sides of the edge.

A **mesh part object** (or, more briefly, a **mesh part**) is a single distinguishable part of a mesh. You can use mesh parts to handle user picking in a mesh. When, for example, the user clicks on a mesh, you can interpret the click as a click on the entire mesh, on a face of a mesh, on an edge of the mesh, or on a vertex of the mesh. QuickDraw 3D signals your application that the user clicked on a mesh part by putting a reference to that mesh part in the `shapePart` field of a hit data structure. (Mesh parts are currently the only types of shape part objects.) You can then call QuickDraw 3D routines to get the mesh face, edge, or vertex that corresponds to the selected mesh part. See the chapter "Pick Objects" for complete details about mesh parts.

A **mesh component** (or a **component**) is a collection of connected vertices. (Two vertices are considered to be **connected** if an unbroken path of edges exists linking one vertex to the other.) For each mesh, QuickDraw 3D maintains information about the components in the mesh and updates that information whenever a face or vertex is added to or removed from a mesh. You can use QuickDraw 3D routines to iterate through the components in a mesh, and you

can call `Q3MeshPart_GetComponent` to get the component in a mesh that was selected during picking. Mesh components cannot have attributes.

Mesh components are transient; that is, they are created and destroyed dynamically as the topology of the mesh changes. Whenever you change the topology (for example, by adding or deleting a vertex or face), QuickDraw 3D needs to update its internal list of mesh components. You can turn off this updating by calling the `Q3Mesh_DelayUpdates` function, and you can resume this updating by calling the `Q3Mesh_ResumeUpdates` function. For performance reasons, it's useful to delay updates while adding or deleting a large number of vertices or faces.

Note, however, that you cannot rely on some mesh functions to return accurate results if you call them while mesh updating is delayed. For instance, the `Q3Mesh_GetNumComponents` function is not guaranteed to return accurate results if mesh updating is delayed.

Note also that a vertex, edge, or face might be shifted from one component to another during a change in the topology of the mesh. To be safe, you should bracket all changes to the mesh topology by calls to `Q3Mesh_DelayUpdates` and `Q3Mesh_ResumeUpdates`, and you should not assume that mesh component functions will return reliable results until after you've called `Q3Mesh_ResumeUpdates`.

Note
You can duplicate a mesh by calling `Q3Object_Duplicate`. The duplicate mesh, however, might not preserve the ordering of components, faces, or vertices of the original mesh. ◆

Trigrids

A **trigrid** is a rectangular grid composed of triangular facets. A trigrid, like most other QuickDraw 3D primitives, is defined using a public data structure, the `TQ3TriGridData` data type:

```
typedef struct TQ3TriGridData {
    unsigned long                   numRows;
    unsigned long                   numColumns;
    TQ3Vertex3D                     *vertices;
    TQ3AttributeSet                 *facetAttributeSet;
    TQ3AttributeSet                 triGridAttributeSet;
} TQ3TriGridData;
```

Once it's defined, a trigrid has a fixed topology defined by the number of rows and columns. You can alter the position of any individual vertex, but you cannot add vertices to (or remove vertices from) a trigrid. In addition, a trigrid can model only rectangular objects, not arbitrary three-dimensional surfaces. Nevertheless, trigrids use memory extremely efficiently and are therefore good choices for modeling rectangular objects.

Polyhedra

A **polyhedron** is a polyhedral primitive, all of whose faces are triangular. (As you'll see below, however, it's possible to render non-triangular faces by selecting which edges of each triangular face are drawn.) The faces of a polyhedron are defined indirectly, using indices into an array of vertices. This indirection makes it easy for faces to share vertices and attribute sets, which thereby reduces both the memory required to define the polyhedron and the time required to render the polyhedron.

IMPORTANT

The polyhedron is the preferred polyhedral primitive for general-purpose modeling of three-dimensional surfaces. Unlike a trigrid, a polyhedron can represent any surface, not just rectangular ones. In addition, you can use both immediate and retained modes with polyhedra. ▲

To define a polyhedron, you first need to create an array of three-dimensional points (of type `TQ3Point3D`). Then you need to define an array of triangles, each of which specifies three of the points in the point array and some additional information about which edges of the triangle to draw and what attributes, if any, the triangle has.

You specify a point in the array of points using a vertex specified by its index into the array of three-dimensional points.

An individual triangular face of a polyhedron is defined by the `TQ3PolyhedronTriangleData` data type.

```
typedef struct TQ3PolyhedronTriangleData {
    unsigned long                   vertexIndices[3];
    TQ3PolyhedronEdge               edgeFlag;
    TQ3AttributeSet                 triangleAttributeSet;
} TQ3PolyhedronTriangleData;
```

The `edgeFlag` field specifies which edges of the triangle are to be drawn; see below for more details.

Finally, once you've created the array of points in the array and defined one or more triangular faces for the polyhedron, you can define a polyhedron using the `TQ3PolyhedronData` data type:

```
typedef struct TQ3PolyhedronData {
    unsigned long                       numPoints;
    TQ3Vertex3D                         *vertices;
    unsigned long                       numEdges;
    TQ3PolyhedronEdgeData               *edges;
    unsigned long                       numTriangles;
    TQ3PolyhedronTriangleData           *triangles;
    TQ3AttributeSet                     polyhedronAttributeSet;
} TQ3PolyhedronData;
```

This structure specifies the number of points in the polyhedron, the points array, the number of triangles in the polyhedron, and the triangles array. These fields contain the minimum data you need to define a polyhedron.

The polyhedron data structure also contains information about the edges in the polyhedron. You can specify edge information either using the `edgeFlag` field of each individual triangle, or you can do so using the `numEdges` and `edges` fields of the polyhedron data structure. See "Polyhedra" (page 311) for more information on specifying polyhedron edges.

Trimeshes

Trimeshes are similar to polyhedra in that they are defined indirectly, using indices into an array of points. In addition, a trimesh has an optional edge array that defines the edges that are to be drawn. However, trimeshes handle attributes quite differently from all other QuickDraw 3D geometric primitives. You do not store attributes for a trimesh (or for any part of a trimesh) in a set of type `TQ3AttributeSet`. Instead, you must use a structure of type `TQ3TriMeshAttributeData`, which stores attribute data contiguously in a single block of memory.

More importantly, attributes associated with a trimesh must conform to this restriction: if any single vertex (or edge, or face) has an attribute of a specific non-custom type, then *every* vertex (or edge, or face) in the trimesh must also

have an attribute of that type. (There are, therefore, no shared attributes.) This restriction can deleteriously affect the memory requirements of a large trimesh.

The trimesh is not suitable for general-purpose use representing polyhedral models. The restrictions on attribute storage can result in very large memory requirements, even though only a few faces might need attributes assigned to them. In addition, there are no functions provided by QuickDraw 3D that allow you to change the geometric or topological configuration of a trimesh object. Trimeshes are designed for immediate mode rendering, and are most suitable for surfaces in which all the component triangles have the same types of attributes.

Comparison of the Polyhedral Primitives

You can use the four polyhedral primitives—the polyhedron, trimesh, mesh, and trigrid—to create similar shapes. However, these primitives offer important differences in their generality, flexibility, style of programming, performance, and compliance with the overall design goal of treating retained and immediate mode programming as equivalent. Table 4-1 provides an overview of their chracteristics, which are discussed in greater detail in "Using Geometric Objects," beginning on page 257.

Table 4-1 Characteristics of polyhedral primitives

Characteristic	Polyhedron	Trimesh	Mesh	Trigrid
Memory usage	Very good	Fair to very good	Poor	Very good
File space usage	Very good	Fair to very good	Very good	Very good
Rendering speed	Good to very good	Good to very good	Fair to good	Good to very good
Geometric object editing	Very good	Impossible (no API calls)	Very good	Very good
Topological object editing	Poor	Impossible (no API calls)	Very good	Impossible (fixed topology)
Geometric data structure editing	Very good	Very good	Impossible (no data structure)	Very good

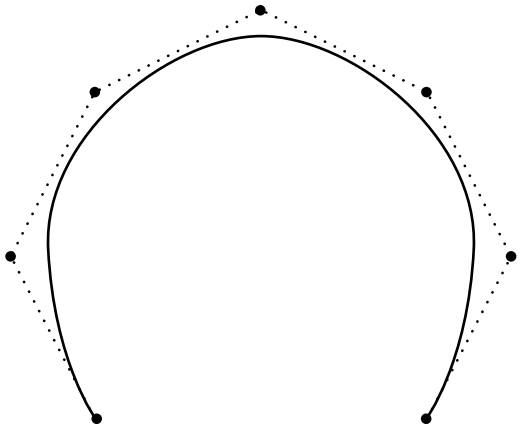
Table 4-1 Characteristics of polyhedral primitives (continued)

Characteristic	Polyhedron	Trimesh	Mesh	Trigrid
Topological data structure editing	Fair	Fair	Impossible (no data structure)	Impossible (fixed topology)
I/O speed	Good to very good	Fair to very good	Fair	Good to very good
Flexibility and generality	Good	Poor	Very good	Poor (fixed topology)
Suitability for general model representation and distribution	Very good	Fair	Fair	Poor

NURB Curves and Patches

QuickDraw 3D supports curves and surfaces that can be defined using **nonuniform rational B-splines (NURBs),** a class of equations defined by nonuniform parametric ratios of B-spline polynomials. A three-dimensional curve represented by a NURB equation is a **NURB curve,** and a three-dimensional surface represented by a NURB equation is a **NURB patch.** Figure 4-3 shows a sample NURB curve.

Figure 4-3 A NURB curve



NURBs can be used to define very complex curves and surfaces, as well as some common geometric objects (for instance, the conic sections). NURB curves and patches are especially useful in 3D imaging because they are invariant under scale, rotate, translation, and perspective transformations of their control points.

A **parametric curve** is any curve whose points are represented by one or more functions of a single parameter (usually denoted by the letter *t* or *u*). The Cartesian coordinates (*x*, *y*) of a two-dimensional parametric curve can be represented generally by these two equations:

$$x = x(u)$$

$$y = y(u)$$

The Cartesian coordinates (*x*, *y*, *z*) of a three-dimensional parametric curve can be represented generally by these three equations:

$$x = x(u)$$

$$y = y(u)$$

$$z = z(u)$$

For compactness, the two- or three-dimensional point is usually represented as a vector. A two-dimensional point has this vector:

$$P(u) = [x(u) \qquad y(u)]$$

For example, a circle can be defined parametrically by a pair of equations:

$$x = r\cos u$$
$$y = r\sin u$$

Alternatively, a circle can be defined parametrically by this vector equation:

$$P(u) = [r\cos u \qquad r\sin u]$$

A **B-spline polynomial** is a parametric equation of this form:

$$P(u) = \sum_{i=1}^{n+1} B_i N_{i,k}(u)$$

where

$$N_{i,1}(u) = \begin{cases} 1 & if\ x_i \le u < x_{i+1} \\ 0 & otherwise \end{cases}$$

$$N_{i,k}(u) = \frac{(u - x_i)N_{i,k-1}(u)}{x_{i+k-1} - x_i} + \frac{(x_{i+k} - u)N_{i+1,k-1}(u)}{x_{i+k} - x_{i+1}}$$

In these equations, the x_i are elements of an array of real numbers, known as the **knot vector,** where each element is greater than or equal to the previous (that is, they are nondecreasing). The B_i are, algebraically, the coefficients of the polynomial representing the curve. Geometrically, they are the (x, y) positions (in a two-dimensional curve) of **control points,** which (together with the knot vector) define the shape of the particular curve of which they are a part. The control points and the knots define the curve's shape in this way: a position of a point on the curve at some parametric value u is a weighted combination of the

positions of a subset of all the control points; the "weighting" is determined by the relative values of the knot vector.

Finally, a NURB curve is a curve defined by ratios of B-spline polynomials, where the values assigned to the parameter can be nonuniform. A NURB patch is a surface defined by ratios of B-spline surfaces, which are three-dimensional analogs of B-spline curves. A **B-spline surface** is a surface defined by a parametric equation of this form:

$$Q(u,v) = \frac{\sum_{i=1}^{n+1}\sum_{j=1}^{m+1} w_{i,j}B_{i,j}N_{i,k}(u)M_{j,l}(v)}{\sum_{i=1}^{n+1}\sum_{j=1}^{m+1} w_{i,j}N_{i,k}(u)M_{j,l}(v)}$$

where

$$N_{i,1}(u) = \begin{cases} 1 & if\ x_i \le u < x_{i+1} \\ 0 & otherwise \end{cases}$$

$$N_{i,k}(u) = \frac{(u - x_i)N_{i,k-1}(u)}{x_{i+k-1} - x_i} + \frac{(x_{i+k} - u)N_{i+1,k-1}(u)}{x_{i+k} - x_{i+1}}$$

and

$$M_{j,1}(v) = \begin{cases} 1 & if\ y_j \le v < y_{j+1} \\ 0 & otherwise \end{cases}$$

$$M_{j,k}(v) = \frac{(v - y_j)M_{j,l-1}(v)}{y_{j+l-1} - y_j} + \frac{(y_{j+l} - v)M_{j+1,l-1}(v)}{y_{j+l} - y_{j+1}}$$

In these equations, the factors $B_{i,j}$ are, algebraically, the coefficients of the polynomial representing the surface. Geometrically, they are the (x, y, z) coordinates of the control points that define the surface. The factors $w_{i,j}$ are the

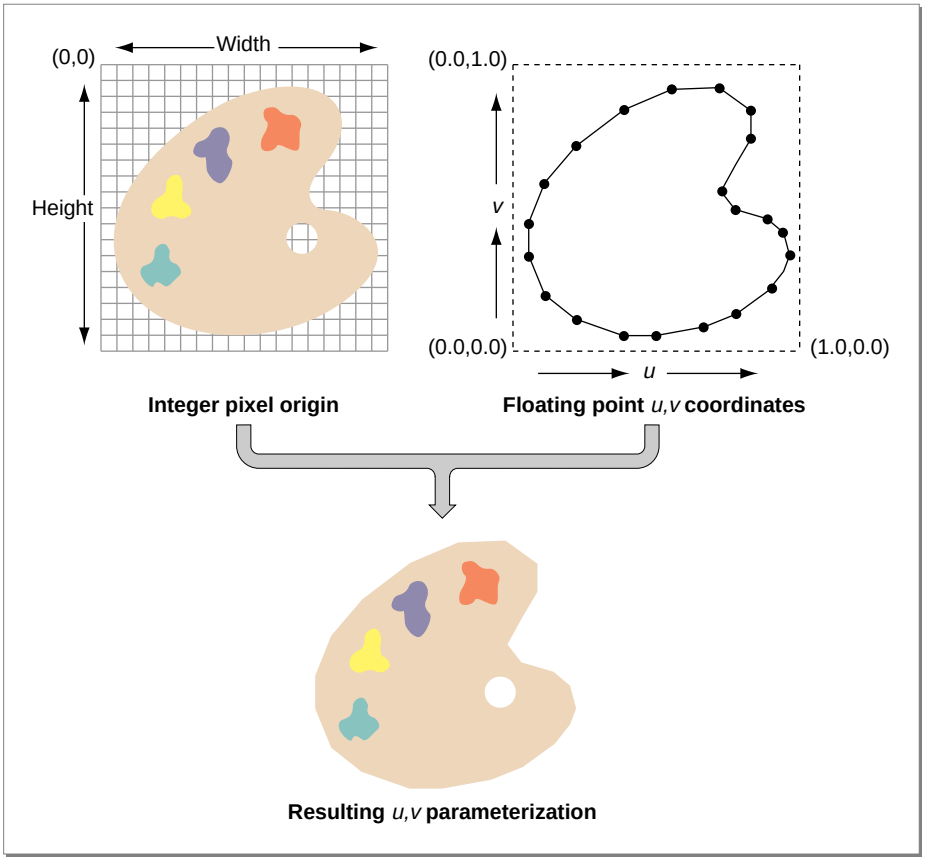
weights of those control points. The factors x_i and y_j are elements of arrays of real numbers, again called knot vectors. These vectors must be non-decreasing.

Surface Parameterizations

For some modeling operations—in particular, applying a texture to the surface of an object—QuickDraw 3D needs to perform a mapping between the texture and the surface. This mapping is usually specified using a pair of *uv* parametric spaces, one defined over the texture and one defined over the surface of the object. A *uv* parametric space is also called a **parameterization.** A *uv* parametric space applied to the surface of an object is a **surface parameterization.**

A texture is typically specified as a pixmap, that is, as a rectangular array of pixels. In that case, the texture has a simple *uv* parameterization (shown in Figure 4-4) that allows QuickDraw 3D to select pixels in the pixmap by varying *u* and *v* in the range 0 to 1. Figure 4-4 (page 253) shows the pixmap, with its origin in the upper-left corner; it also shows the standard pixmap parameterization, which maps the unit box from 0.0 to 1.0 along the *u* and *v* axes.

Figure 4-4 The standard *uv* parameterization for a pixmap



In addition to this texture parameterization, QuickDraw 3D uses another parameterization that picks out points on the surface of the object. For texture mapping, the most useful **standard surface parameterization** is any parameterization that results in the entire texture being mapped to the entire surface exactly once. QuickDraw 3D defines a standard surface parameterization for most of the primitive QuickDraw 3D geometric objects. In some cases, an object's standard surface parameterization is obtained from the object's **natural surface parameterization** (that is, a parameterization that defines the surface). For example, a NURB patch is naturally parameterized by its u and v knot vectors. (However, note that a texture will be mapped only into

the subregion of the patch that corresponds to the 1 by 1 subregion of domain space. You must take this into account when assigning values larger than 1 to a patch's knot vectors.)

In other cases, however, there is no natural surface parameterization for an object, and QuickDraw 3D must define an arbitrary standard surface parameterization for it. For example, for a box, which has no natural surface parameterization, QuickDraw 3D uses the standard surface parameterization shown in Figure 4-5.

Figure 4-5 The standard surface parameterization of a box

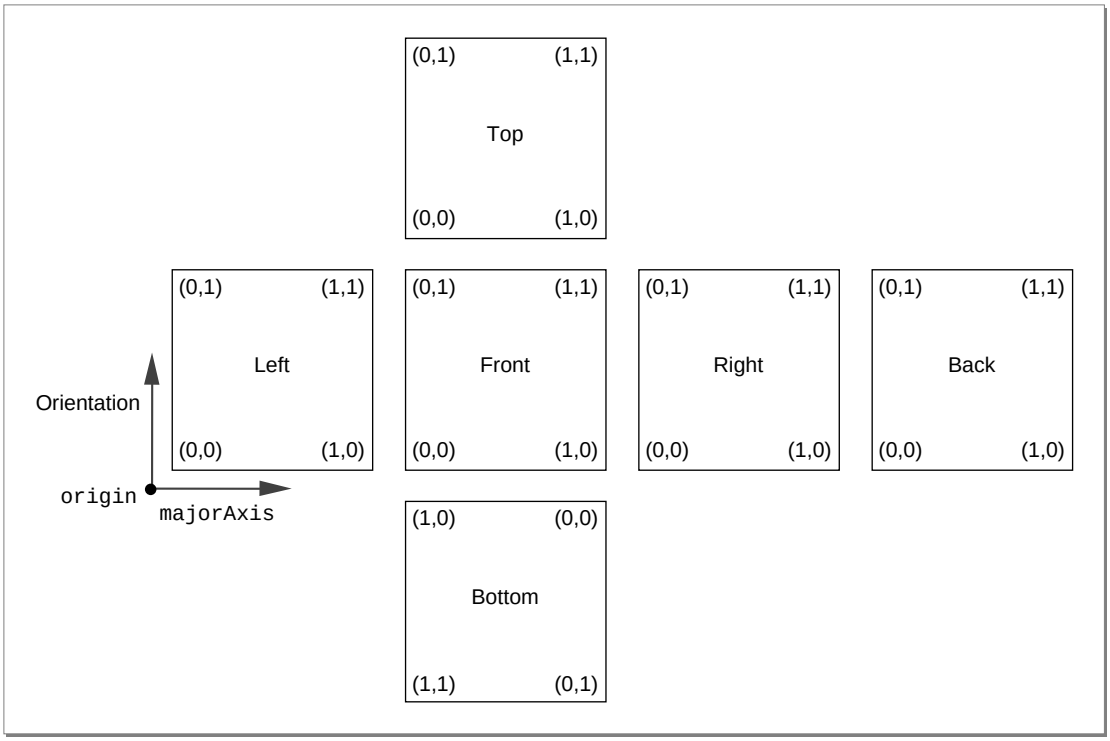
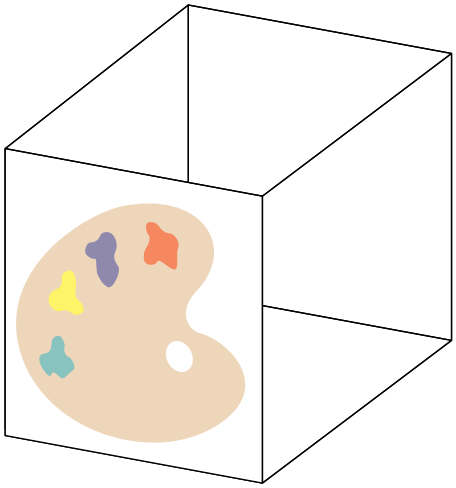


Figure 4-6 shows the result of mapping the texture shown in Figure 4-4 onto the front face of a box.

Figure 4-6 A texture mapped onto a box

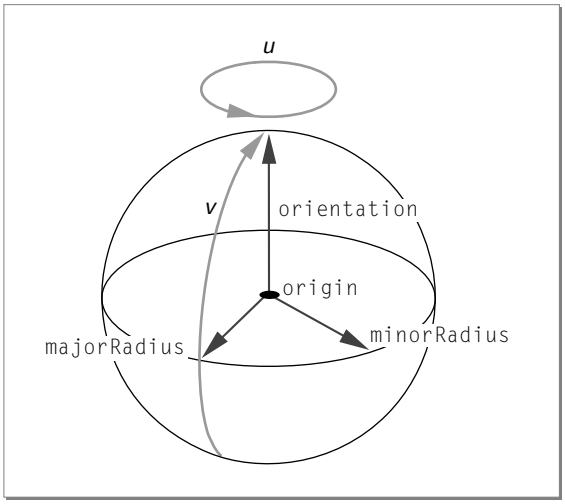


Similarly, an ellipsoid has the standard surface parameterization shown in Figure 4-7 (page 256).

In this case, the v parameter varies from 0 to 1 as it sweeps from the end of the orientation vector to the top of the ellipsoid, and the u parameter varies from 0 to 1 as it sweeps around in the plane defined by the major radius and minor radius. In the coordinate system defined by the orientation, the major axis, and the minor axis, the standard surface parameterization is given by these equations, where u and v are both defined on the interval [0, 1):

$$x_{major} = \cos(2\pi u) \cdot \sin(2\pi v)$$
$$x_{minor} = \sin(2\pi u) \cdot \sin(2\pi v)$$
$$x_{orient} = \cos(\pi v)$$

Figure 4-7 The standard surface parameterization for an ellipsoid.



Some objects have neither a natural surface parameterization nor a standard surface parameterization supplied by QuickDraw 3D. For example, the faces of a mesh have neither type of parameterization. To apply a texture to such an object, you need to define your own **custom surface parameterization.** You do this by adding attributes of type `kQ3AttributeTypeSurfaceUV` to the vertices of the object. See Listing 4-5 (page 271) for details.

It's possible to modify the mapping used in applying a texture to a surface, by changing the surface's *uv* shading transform. (For example, you can rotate the texture any desired amount by installing the appropriate transformation matrix.) See the chapter "Shader Objects" for information on setting the *uv* transform used by a surface shader.

Note
To override an object's standard surface parameterization, or to define a custom surface parameterization for an object that has no standard surface parameterization, you need to manipulate the surface *uv* attributes of the object. See the chapter "Attribute Objects" for details. ◆

The standard surface parameterizations of the QuickDraw 3D geometric objects are given in the section "Geometric Objects Reference."

Using Geometric Objects

QuickDraw 3D provides routines that you can use to create and edit geometric objects, get and set attributes for those objects, and perform other geometric operations. This section illustrates how to create and delete some geometric objects and how to traverse the parts of a mesh.

Creating and Deleting Geometric Objects

As you saw briefly in the chapter "Introduction to QuickDraw 3D," QuickDraw 3D supports both immediate and retained modes of defining and rendering a model. Which mode you employ in any particular instance depends on the needs of your application. As suggested earlier, if much of the model remains unchanged from frame to frame, you should use retained mode imaging to create and draw the model. If, however, many parts of the model do change from frame to frame, you should use immediate mode imaging, creating and rendering a model on a shape-by-shape basis.

Listing 4-1 illustrates how to create a retained box.

Listing 4-1 Creating a retained box

```
TQ3GeometryObject               myBox;
TQ3BoxData                      myBoxData;

Q3Point3D_Set(&myBoxData.origin, 1.0, 1.0, 1.0);
Q3Vector3D_Set(&myBoxData.orientation, 0, 2.0, 0);
Q3Vector3D_Set(&myBoxData.minorAxis, 2.0, 0, 0);
Q3Vector3D_Set(&myBoxData.majorAxis, 0, 0, 2.0);
myBox = Q3Box_New(&myBoxData);
```

Once the code in Listing 4-1 has been executed, the variable `myBox` contains a reference to the new box. You can then reuse or dispose of the `myBoxData` structure, because all subsequent operations on the retained box are performed using `myBox`. For example, to submit the box for drawing, picking, bounding, or

writing, you can execute the following line of code inside a rendering, picking, bounding, or writing loop:

```
myStatus = Q3Object_Submit(myBox, myView);
```

To dispose of the retained box, you can call the `Q3Object_Dispose` function, as follows:

```
myStatus = Q3Object_Dispose(myBox);
```

Listing 4-2 illustrates how to create an immediate box.

Listing 4-2 Creating an immediate box

```
TQ3BoxData                      myBoxData;

Q3Point3D_Set(&myBoxData.origin, 1.0, 1.0, 1.0);
Q3Vector3D_Set(&myBoxData.orientation, 0, 2.0, 0);
Q3Vector3D_Set(&myBoxData.minorAxis, 2.0, 0, 0);
Q3Vector3D_Set(&myBoxData.majorAxis, 0, 0, 2.0);
```

As you can see, you do not have to call any QuickDraw 3D routine to create an immediate box; instead, you simply define the box data in a structure of type `TQ3BoxData`. To draw an immediate box, you call the `Q3Box_Submit` function (inside a rendering loop), as follows:

```
myStatus = Q3Box_Submit(myBox, myView);
```

Because you didn't create any retained entity, you do not need to dispose of the immediate box.

Using Polyhedrons

The polyhedron is the primitive of choice for most programming situations, as well as for the creation and distribution of editable model files. Thus if your application requires the creation, conversion, or distribution of polyhedral models, you should produce them in polyhedron format instead of mesh or trimesh. User applications such as modelers and animation tools should also generally manipulate polyhedrons. Plug-in renderers are required to support

certain basic primitives (triangles, points, lines, and markers) and are strongly urged to support the polyhedron as well.

The polyhedron format gives you these features:

- It can easily represent many different polyhedral models in a space-efficient fashion.
- It's capable of fast rendering.
- It's highly consistent with the rest of the QuickDraw 3D API.
- Attributes can be attached in whatever combination is appropriate for the model.

Polyhedrons make geometric editing operations, which change the positions of existing vertices, easy and convenient. In immediate mode, you can simply alter a point's position in the array in the polyhedron data structure and render the shape again. In retained mode, several function calls let you change vertex locations, as well as providing the usual assortment of Get and Set calls for attributes, faces, face attributes, and so on.

You can use topological editing operations to change the relationships between vertices, faces, edges, and the whole polyhedron. However, the addition or deletion of vertices, faces, or edges may require reallocation of one or more of the polyhedron's arrays. Because the polyhedron has a public data structure, these operations are possible in both immediate mode and retained mode. If adding and deleting vertices, faces, or edges aren't the primary operations required for using the polyhedron, array reallocation will not be a problem; if they are, you should use the mesh primitive instead.

The polyhedron uses memory and disk space efficiently because shared locations and attributes are stored only once and only those parts that logically require attributes get them. This produces good I/O speed, although, as with all geometric primitives, the addition of textures can increase I/O time significantly. The polyhedron also features superior rendering speed because its vertices are shared.

Creating a Polyhedron

The normal way to make a polyhedron is to create an array of points and a list of triangular faces that organize the points. Each face consists of a list of indices into the list of vertices, forming a polygon with one level of array-based indirection. If there is more than one face, the vertices can be shared by reusing the same array indices in each face. This allows the graphics system to run

faster because the same point doesn't have to be transformed or shaded more than once, and it saves storage space. In addition, because two or more faces share only one real vertex, this format makes it easier to do interactive editing programming.

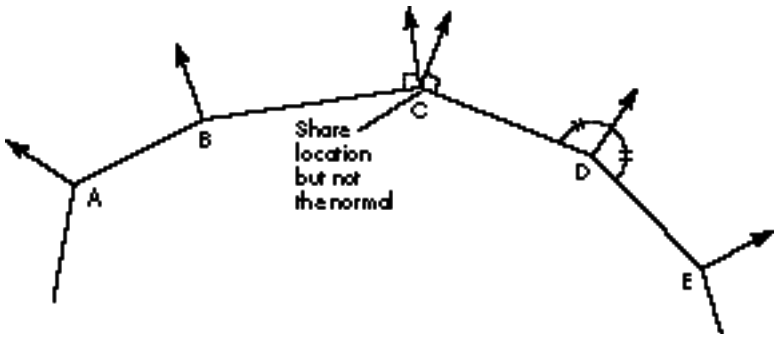
Polyhedrons—objects of type `kQ3GeometryTypePolyhedron`—implement this process in a way that is consistent with the other QuickDraw 3D primitives. Its basic component is the vertex of type `TQ3Vertex3D`, an {x, y, z} location with an attribute set. The vertices of adjacent triangular faces are shared simply by using the same vertex indices. Also, sets of attributes may be shared like other objects in QuickDraw 3D:

```
vertex->attributeSet = Q3Shared_GetReference(otherVertex->attributeSet);
```

Vertices can contain the same locations, but need not share attributes. This can be useful, for example, when creating a polyhedron that is generally smooth but has some edges or corners where you want a discontinuity. For example, consider the cross section of a polyhedron shown in Figure 4-8, which has vertices sharing locations but not attributes.

In Figure 4-8, each location is shared, and vertices at positions A, B, D, and E share normals, while the vertices at position C share the location but not the normal. So when smooth-shaded, the object has an edge or corner at position C but appears smooth elsewhere.

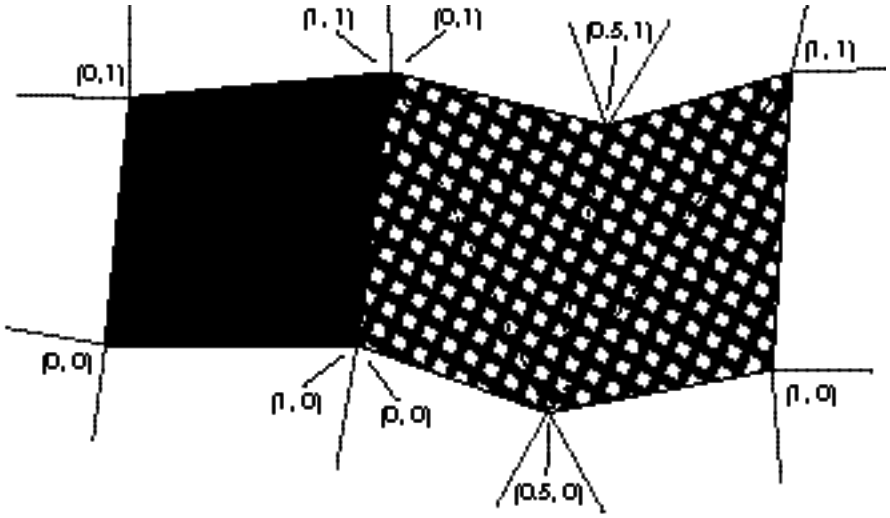
Figure 4-8 Cross-section of a polyhedron



Because values in an attribute set apply to all vertices or faces sharing that attribute set, operations on it will affect all these elements. For example, you can associate a single texture with a group of faces by simply giving each face a

shared reference to the texture-containing attribute set. For a single texture to span a number of faces, you need to make sure their shared vertices share texture coordinates. You can do this by making shared vertices of faces that are spanned by a single texture use the same attribute set, as shown in Figure 4-9.

Figure 4-9 Applying textures that span several faces

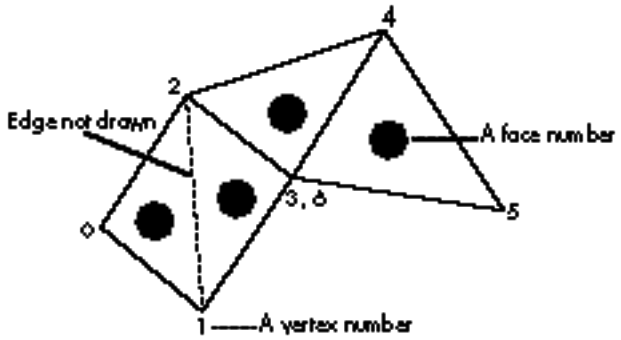


Besides an attribute set for the face, the three vertices defining a face of a polyhedron are in an array of size 3. The polyhedron also uses an enumerated type that defines which edges are drawn and which not:

```
typedef enum TQ3PolyhedronEdgeMasks {
    kQ3PolyhedronEdgeNone   = 0,
    kQ3PolyhedronEdge01     = 1 << 0,
    kQ3PolyhedronEdge12     = 1 << 1,
    kQ3PolyhedronEdge20     = 1 << 2,
    kQ3PolyhedronEdgeAll    = kQ3PolyhedronEdge01 |
                              kQ3PolyhedronEdge12 |
                              kQ3PolyhedronEdge20
} TQ3PolyhedronEdgeMasks;

typedef unsigned long TQ3PolyhedronEdge;
```

By OR-combining these flags you can select which edges of a particular triangle you want drawn. For example, if you're using a wireframe renderer to draw an object like the one shown in Figure 4-10, you wouldn't have to show the "internal" edges, just the edges that represent the true border of the face. For face 0 in Figure 4-10, you could specify that you want to display only the edges between vertices 0 and 1 and vertices 2 and 0, leaving undrawn the edge between vertices 1 and 2. You'd do this by specifying `(kQ3PolyhedronEdge01 | kQ3PolyhedronEdge20)` as the edge mask.

Figure 4-10 Wireframe polyhedron



All the information discussed so far is collected in this data structure:

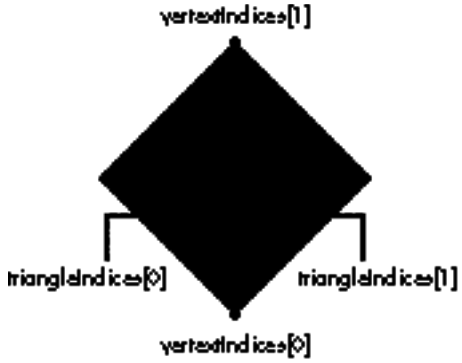
```
typedef struct TQ3PolyhedronTriangleData {
    unsigned long       vertexIndices[3];
    TQ3PolyhedronEdge   edgeFlag;
    TQ3AttributeSet     triangleAttributeSet;
} TQ3PolyhedronTriangleData;
```

An alternative to using a mask to specify the edges is to create a list of edges for the entire polyhedron. If the renderer draws the edges (or lines, in the case of a wireframe renderer) from an edge list, the renderer can transform the points just once each and draw each edge just once, resulting in much faster rendering. The renderer ignores the edge flags in the face data structure if an array of these edges is present:

```
typedef struct TQ3PolyhedronEdgeData {
    unsigned long       vertexIndices[2];
    unsigned long       triangleIndices[2];
    TQ3AttributeSet     edgeAttributeSet;
} TQ3PolyhedronEdgeData;
```

As shown in Figure 4-11, the `vertexIndices` field specifies indices into the vertex array, one for the vertex at each end of each edge.

Figure 4-11 Filling out a polyhedron's edge data structure



The `triangleIndices` field shown in Figure 4-11 specifies indices into the array of faces. You need to provide the indices to the faces that share an edge because performing correct backface removal requires that the edge be drawn only if at least one of the faces that it's part of is facing forward.

The `edgeAttributeSet` field allows the application to specify the color and other attributes of edges independently. If no attribute is set for an edge, the attributes are inherited from the geometric object, or from the view's state if that's not present. Every edge must have two points, but edges may have one or two faces adjacent to them; those with just one are on a boundary of the object. To represent a boundary in an array-based representation, you use the identifier `kQ3ArrayIndexNULL` as a face index for the side of an edge that has no face attached to it.

Note
When going from the vertex at index 0 to the vertex at index 1 in Figure 4-11, the 0th face is to the left. If possible, fill out your data structures to conform to this practice. Other code may want to traverse the edge list and be assured of knowing exactly which face is on which side of each edge. ◆

The following is the entire polyhedron data structure:

```
typedef struct TQ3PolyhedronData {
    unsigned long               numVertices;
    TQ3Vertex3D                 *vertices;
    unsigned long               numEdges;
    TQ3PolyhedronEdgeData       *edges;
    unsigned long               numTriangles;
    TQ3PolyhedronTriangleData   *triangles;
    TQ3AttributeSet             polyhedronAttributeSet;
} TQ3PolyhedronData;
```

Listing 4-3 shows the code that creates the four-faced polyhedron shown in Figure 4-10.

Listing 4-3 Creating a four-faced polyhedron

```
TQ3ColorRGB         color;
TQ3PolyhedronData   polyhedronData;
TQ3GeometryObject   polyhedron;
TQ3Vector3D         normal;

static TQ3Vertex3Dvertices[7] = {
    { { -1.0,  1.0,  0.0 }, NULL },
    { { -1.0, -1.0,  0.0 }, NULL },
    { {  0.0,  1.0,  1.0 }, NULL },
    { {  0.0, -1.0,  1.0 }, NULL },
    { {  2.0,  1.0,  1.0 }, NULL },
    { {  2.0, -1.0,  0.0 }, NULL },
    { {  0.0, -1.0,  1.0 }, NULL }
};
```

```
TQ3PolyhedronTriangleData   triangles[4] = {
    {  /* Face 0 */
        { 0, 1, 2 },                                /* vertexIndices */
        kQ3PolyhedronEdge01 | kQ3PolyhedronEdge20,  /* edgeFlag */
        NULL                                    /* triangleAttributeSet */
    },
    {  /* Face 1 */
        { 1, 3, 2 },
        kQ3PolyhedronEdge01 | kQ3PolyhedronEdge12,
        NULL
    },
    {  /* Face 2 */
        { 2, 3, 4 },
        kQ3PolyhedronEdgeAll,
        NULL
    },
    {../* Face 3 */
        { 6, 5, 4 },
        kQ3PolyhedronEdgeAll,
        NULL
    }
};

/* Set up vertices, edges, and triangular faces. */
polyhedronData.numVertices      = 7;
polyhedronData.vertices         = vertices;
polyhedronData.numEdges         = 0;
polyhedronData.edges            = NULL;
polyhedronData.numTriangles     = 4;
polyhedronData.triangles        = triangles;

/* Inherit the attribute set from the current state. */
polyhedronData.polyhedronAttributeSet = NULL;

/* Put a normal on the first vertex. */
Q3Vector3D_Set(&normal, -1, 0, 1);
Q3Vector3D_Normalize(&normal, &normal);
vertices[0].attributeSet = Q3AttributeSet_New();
Q3AttributeSet_Add(vertices[0].attributeSet, kQ3AttributeTypeNormal,
    &normal);
```

```
/* Same normal on the second. */
vertices[1].attributeSet =
    Q3Shared_GetReference(vertices[0].attributeSet);

/* Different normal on the third. */
Q3Vector3D_Set(&normal, -0.5, 0.0, 1.0);
Q3Vector3D_Normalize(&normal, &normal);
vertices[2].attributeSet = Q3AttributeSet_New();
Q3AttributeSet_Add(vertices[2].attributeSet, kQ3AttributeTypeNormal,
    &normal);

/* Same normal on the fourth. */
vertices[3].attributeSet =
    Q3Shared_GetReference(vertices[2].attributeSet);

/* Put a color on the third triangle. */
triangles[3].triangleAttributeSet = Q3AttributeSet_New();
Q3ColorRGB_Set(&polyhedronColor, 0, 0, 1);
Q3AttributeSet_Add(triangles[3].triangleAttributeSet,
    kQ3AttributeTypeDiffuseColor, &polyhedronColor);

/* Create the polyhedron object. */
polyhedron = Q3Polyhedron_New(&polyhedronData);

... /* Dispose of attributes created and referenced. */
```

Listing 4-4 shows code that specifies the edges of the polyhedron shown in Figure 4-10, but using the optional edge list. It is added to the code in Listing 4-3. When using an edge list, you would set the edge flags in the triangle data of Listing 4-3 to a legitimate value, such as `kQ3EdgeFlagAll`, that will be ignored.

Listing 4-4 Using an edge list to specify the edges of a polyhedron

```
polyhedronData.numEdges     = 8;
polyhedronData.edges        = malloc(8 * sizeof(TQ3PolyhedronEdgeData));
```

```
polyhedronData.edges[0].vertexIndices[0]    = 0;
polyhedronData.edges[0].vertexIndices[1]    = 1;
polyhedronData.edges[0].triangleIndices[0]  = 0;
polyhedronData.edges[0].triangleIndices[1]  = kQ3ArrayIndexNULL;
polyhedronData.edges[0].edgeAttributeSet    = NULL;

polyhedronData.edges[1].vertexIndices[0]    = 2;
polyhedronData.edges[1].vertexIndices[1]    = 0;
polyhedronData.edges[1].triangleIndices[0]  = 0;
polyhedronData.edges[1].triangleIndices[1]  = kQ3ArrayIndexNULL;
polyhedronData.edges[1].edgeAttributeSet    = NULL;

polyhedronData.edges[2].vertexIndices[0]    = 1;
polyhedronData.edges[2].vertexIndices[1]    = 3;
polyhedronData.edges[2].triangleIndices[0]  = 1;
polyhedronData.edges[2].triangleIndices[1]  = kQ3ArrayIndexNULL;
polyhedronData.edges[2].edgeAttributeSet    = NULL;

polyhedronData.edges[3].vertexIndices[0]    = 3;
polyhedronData.edges[3].vertexIndices[1]    = 2;
polyhedronData.edges[3].triangleIndices[0]  = 1;
polyhedronData.edges[3].triangleIndices[1]  = 2;
polyhedronData.edges[3].edgeAttributeSet    = NULL;

... /* Specify the rest of the edges. */
```

Using Trimeshes

Like the polyhedron, the trimesh uses a list of points and a list of triangular faces that contain indices into the list of points. It also has an optional edge list. However, it differs from the polyhedron in other ways. The trimesh primitive has several unique characteristics that significantly affect its applicability:

- All trimesh data values are stored in explicit arrays—vertex locations, vertex attributes, triangle attributes, and edge attributes.
- Trimeshes generally do not keep attributes in objects of type `TQ3AttributeSet`; instead, their attributes are kept as arrays of explicit data structures. However, the trimesh does maintain a `TQ3AttributeSet` object for its whole geometry, like other primitives.

- With the exception of custom attributes, every vertex, face, or edge of a trimesh must have exactly the same types of attributes. For example, you must put a color on every face of a trimesh if you want to put a color on just one face; similarly for vertices and edges. For some types of models, such as those in existing applications ported to QuickDraw 3D that already use uniform attributes, this may not be a problem. In such cases, the trimesh may be the natural choice, as well as being faster and more compact.

The uniform-attributes requirement just mentioned, and the use of arrays of explicit data for attributes, can make the trimesh format preferable to the polyhedron in some models and applications. However, these features make it hard to use trimeshes to represent arbitrary, nonuniform polyhedra. Many solid shapes have regions that are smoothly curved and regions that are flat or faceted, as well as sharp edges, corners, and creases. The vertices in the curved regions need normals that approximate the surface normal at that vertex, but vertices at corners or along edges or in flat regions need none. With the polyhedron, mesh, and trigrid formats, you must allocate storage for normals only for those vertices that actually require a normal. With the trimesh format, you must allocate vertex normals on all vertices, causing heavy memory usage.

This same problem applies to face attributes. Solid shapes often have regions that differ in color, transparency, or surface texture. For example, a soccer ball has black and white faces and a wine bottle may have a label on the front, a different label on the back, and another around the neck. The other polyhedral primitives would, in the case of the soccer ball, simply create two attribute sets (one for each color) and attach a reference to the appropriate attribute set to each face, thus sharing the color information. The trimesh format is forced to create an array of colors, using a lot of memory to represent the same data over and over. If you wanted to highlight one face of a soccer ball, you couldn't just attach a highlight switch attribute to that face, set to "on"—you'd need to attach it to the rest as well, set to "off." In the case of the wine bottle, you would want to attach label textures to the appropriate faces on the bottle by attaching texture parameters to the vertices of those faces. With a trimesh, this powerful approach is not possible.

When using the trimesh for large polyhedral models, these problems can result in heavy space usage, both on disk and in memory. Consider a 10,000-face model whose faces are either red or green. The other polyhedral primitives would use references to just two color attribute sets while the trimesh would need 10,000 * 12 = 120,000 bytes. If the red faces were to be transparent, a trimesh would use another 120,000 bytes. Highlighting just one face would require 40,000 bytes more, and the same sort of data explosion would happen

with vertex attributes as well. These problems don't occur with the other polyhedral primitives.

In spite of these features that limit the suitability of the trimesh for general-purpose polyhedral representation, the uniform-attributes requirement makes it ideal for models in which each vertex or face naturally has the same type of attributes as the other vertices (or faces), but with different values. For example, if your application uses Coons patches, it could subdivide the patch into a trimesh with normals on each vertex. Games often are written with objects such as walls, or even some stylized characters, that typically have just one texture and either no vertex attributes or normals on every vertex. Multimedia, some demo programs, and other "display-only" applications in which the user is unable to modify objects may find the trimesh useful, at least for shapes that don't evoke the memory usage problems described above.

Geometric editing operations in immediate mode for the trimesh are similar to those for the polyhedron: you simply alter a point's position in the array in the trimesh data structure and render the shape again. There are no retained-mode API calls for editing parts of a trimesh. Topological editing in immediate mode is also similar to that for the polyhedron. Because there are no suitable API calls, however, it is impossible to edit a trimesh object topologically in retained mode.

The uniform-attributes requirement for trimeshes causes generally good I/O performance. However, poor I/O speeds may result from the repeated transfer of multiple copies of the same data (for example, the same color on every face). Rendering speed for the trimesh is usually good.

Using Meshes

Like the polyhedron and trimesh, the mesh is designed for representing polyhedra. However, it is intended for the interactive topological creation and editing of polyhedra, so its architecture and API were designed to support both iterative construction and topological modification.

Iterative construction means that you can easily construct a mesh by building it face-by-face, instead of filling in a data structure and constructing it from the data structure all at once.

Topological modification means that you can easily add and delete vertices, faces, edges, and other components in a mesh. A mesh has no explicit public data structure; unlike the other geometric primitives, it also has no immediate-mode capability.

Meshes are not intended for representing large-scale polyhedral models with many vertices and faces. If employed this way, the mesh format causes poor I/O behavior, heavy memory usage, and suboptimal rendering speed. Hence modeling, animation, and design applications should use the polyhedron format for most model creation and storage.

On the other hand, in some applications the mesh format is superior to other geometric primitives. For example, it would be ideal in an application that used a 3D sampling peripheral, such as a Polhemus device, to digitize physical objects. You could use the mesh to construct the digitized model face-by-face, to merge or split faces, to add or delete vertices, and so forth. Doing these tasks with an array-based data structure would be awkward to program and force the program to make repeated array reallocations.

The faces of meshes, unlike those of the polyhedron and trimesh, may have more than three vertices, may be concave (though not self-intersecting), and may contain holes by defining faces with more than one list of vertices.

The mesh API supports a rich variety of geometric and topological editing operations, but only for retained mode; it has no immediate-mode public data structure. If your application needs immediate mode, you should use the polyhedron format.

In general, the rendering speed of meshes is relatively slow. They must be either traversed for rendering or decomposed into other primitives that yield faster rendering. Traversing usually results in the slow retransformation and reshading of shared vertices, while decomposition may require heavy memory usage as well as complex and slow bookkeeping code.

To summarize, you should use the mesh primitive for interactive construction and topological editing. Its rich set of geometric and topological editing calls, the ability to make nontriangular faces directly, the ability to make concave faces and faces with holes, and the consistent use of attribute sets make the mesh primitive ideal for many purposes. In addition, the 3D metafile representation of a mesh is quite space efficient. Because the mesh lacks an immediate mode, however, it requires a large amount of memory and may be inefficient for other uses.

Creating a Mesh

As explained in “Meshes,” beginning on page 240, you create a mesh by calling `Q3Mesh_New` to create a new empty mesh and then by calling `Q3Mesh_VertexNew` and `Q3Mesh_FaceNew` to explicitly add vertices and faces to the mesh. Listing 4-5

illustrates how to create a simple mesh using these functions. It also shows how to attach a custom surface parameterization to a mesh face, so that a texture can be mapped onto the face.

Listing 4-5 Creating a simple mesh

```
TQ3GroupObject MyBuildMesh (void)
{
    TQ3ColorRGB                         myMeshColor;
    TQ3GroupObject                      myModel;
    static TQ3Vertex3D                  vertices[9] = {
        { { -0.5,  0.5, 0.0 }, NULL },
        { { -0.5, -0.5, 0.0 }, NULL },
        { {  0.0, -0.5, 0.3 }, NULL },
        { {  0.5, -0.5, 0.0 }, NULL },
        { {  0.5,  0.5, 0.0 }, NULL },
        { {  0.0,  0.5, 0.3 }, NULL },
        { { -0.4,  0.2, 0.0 }, NULL },
        { {  0.0,  0.0, 0.0 }, NULL },
        { { -0.4, -0.2, 0.0 }, NULL }};
    static TQ3Param2D                   verticesUV[9] = {
        {0.0, 1.0}, {0.0, 0.0}, {0.5, 0.0}, {1.0, 0.0},
        {1.0, 1.0}, {0.5, 1.0}, {0.1, 0.8}, {0.5, 0.5},
        {0.1, 0.4}};

    TQ3MeshVertex                       myMeshVertices[9];
    TQ3GeometryObject                   myMesh;
    TQ3MeshFace                         myMeshFace;
    TQ3AttributeSet                     myFaceAttrs;
    unsigned long                       i;

    myMesh = Q3Mesh_New();              /*create new empty mesh*/

    Q3Mesh_DelayUpdates(myMesh);        /*turn off mesh updating*/

    /*Add vertices and surface parameterization to mesh.*/
    for (i = 0; i < 9; i++) {
        TQ3AttributeSet                 myVertAttrs;
```

```
        myMeshVertices[i] = Q3Mesh_VertexNew(myMesh, &vertices[i]);
        myVertAttrs = Q3AttributeSet_New();
        Q3AttributeSet_Add(myVertAttrs, kQ3AttributeTypeSurfaceUV, &verticesUV[i]);
        Q3Mesh_SetVertexAttributeSet(myMesh, myMeshVertices[i], myVertAttrs);
        Q3Object_Dispose(myVertAttrs);
    }

    myFaceAttrs = Q3AttributeSet_New();
    myMeshColor.r = 0.3;
    myMeshColor.g = 0.9;
    myMeshColor.b = 0.5;
    Q3AttributeSet_Add(myFaceAttrs, kQ3AttributeTypeDiffuseColor, &myMeshColor);

    myMeshFace = Q3Mesh_FaceNew(myMesh, 6, myMeshVertices, myFaceAttrs);

    Q3Mesh_FaceToContour(myMesh, myMeshFace,
                        Q3Mesh_FaceNew(myMesh, 3, &myMeshVertices[6], NULL));

    Q3Mesh_ResumeUpdates(myMesh);

    myModel = Q3OrderedDisplayGroup_New();
    Q3Group_AddObject(myModel, myMesh);
    Q3Object_Dispose(myFaceAttrs);
    Q3Object_Dispose(myMesh);
    return (myModel);
}
```

The new mesh created by `MyBuildMesh` is a retained object. Note that you need to call `Q3Mesh_New` before you call `Q3Mesh_VertexNew` and `Q3Mesh_FaceNew`. Also, the call to `Q3Mesh_FaceToContour` destroys any attributes associated with the mesh face that is turned into a contour.

Traversing a Mesh

QuickDraw 3D supplies functions that you can use to traverse a mesh by iterating through various parts of the it. For example, you can operate on each face of a mesh by calling the `Q3Mesh_FirstMeshFace` function to get the first face in the mesh and then `Q3Mesh_NextMeshFace` to get each successive face. When you call `Q3Mesh_FirstMeshFace`, you specify a mesh and a **mesh iterator structure,** which QuickDraw 3D fills in with information about its current position while traversing the mesh. You must pass that same mesh iterator

structure to `Q3Mesh_NextMeshFace` when you get successive faces in the mesh. Listing 4-6 illustrates how to use these routines to operate on all faces in a mesh.

Listing 4-6 Iterating through all faces in a mesh

```
TQ3Status MySetMeshFacesDiffuseColor (TQ3GeometryObject myMesh,
                                      TQ3ColorRGB color)
{
    TQ3MeshFace                 myFace;
    TQ3MeshIterator             myIter;
    TQ3Status                   myErr;
    TQ3AttributeSet             mySet;

    for (myFace = Q3Mesh_FirstMeshFace(myMesh, &myIter);
          myFace;
          myFace = Q3Mesh_NextMeshFace(&myIter)) {

        /*Get the current attribute set of the current face.*/
        myErr = Q3Mesh_GetFaceAttributeSet(myMesh, myFace, &mySet);
        if (myErr == kQ3Failure) return (kQ3Failure);

        /*Add the color attribute to the face attribute set.*/
        myErr = Q3AttributeSet_Add((TQ3AttributeSet)mySet,
                                   kQ3AttributeTypeDiffuseColor, &color);
        if (myErr == kQ3Failure) return (kQ3Failure);

        /*Set the attribute set of the current face.*/
        myErr = Q3Mesh_SetFaceAttributeSet(myMesh, myFace, mySet);
        if (myErr == kQ3Failure) return (kQ3Failure);
    }
    return (kQ3Success);
}
```

QuickDraw 3D also supplies a number of C language macros that you can use to simplify your source code when traversing a mesh. For example, you can use the following `Q3ForEachMeshFace` macro:

```
#define Q3ForEachMeshFace(m,f,i)
    for ( (f) = Q3Mesh_FirstMeshFace((m),(i));
        (f);
        (f) = Q3Mesh_NextMeshFace((i)) )
```

Listing 4-7 shows how to use two of these macros to attach a corner to each vertex or each face of a mesh.

Listing 4-7 Attaching corners to all vertices in all faces of a mesh

```
TQ3Status MyAddCornersToMesh (TQ3GeometryObject myMesh,
                              TQ3AttributeSet mySet)
{
    TQ3MeshFace                 myFace;
    TQ3MeshVertex               myVertex;
    TQ3MeshIterator             myIter1;
    TQ3MeshIterator             myIter2;
    TQ3Status                   myErr;

    Q3ForEachMeshFace(myMesh, myFace, &myIter1) {
        Q3ForEachFaceVertex(myFace, myVertex, &myIter2) {
            myErr = Q3Mesh_SetCornerAttributeSet
                        (myMesh, myFace, myVertex, mySet);
            if (myErr == kQ3Failure) return (kQ3Failure);
        }
    }
    return (kQ3Success);
}
```

Using Trigrids

The trigrid format has a fixed topology, defined by the numbers of its rows and columns. As a result, memory and file space is very efficiently used and I/O and rendering speeds are fast. However, the trigrid's fixed topology and the fact that shared locations must share attributes restrict its generality and flexibility.

The trigrid format is good for representing objects that are topologically rectangular—for example, surfaces of revolution, swept surfaces, and terrain models. It is also useful as an output primitive for applications that need to

decompose their own parametric or implicit surfaces. In such applications the trigrid can be an especially good choice because it's more space efficient than the other polyhedral primitives.

Geometric Objects Reference

This section describes the constants and data structures provided by QuickDraw 3D that you can use to define the QuickDraw 3D geometric objects. It also describes the routines you can use to create and manipulate those objects.

Constants

This section describes the constants that pertain to QuickDraw 3D geometric objects.

Geometric Object Types

Every QuickDraw 3D geometric object has an object type, which you can determine by calling the `Q3Geometry_GetType` function. `Q3Geometry_GetType` returns one of the following constants, or `kQ3ObjectTypeInvalid` if the type of an object cannot be determined or is invalid.

```
#define kQ3GeometryTypeBox              Q3_OBJECT_TYPE('b','o','x',' ')
#define kQ3GeometryTypeCone             Q3_OBJECT_TYPE('c','o','n','e')
#define kQ3GeometryTypeCylinder         Q3_OBJECT_TYPE('c','y','l','n')
#define kQ3GeometryTypeDisk             Q3_OBJECT_TYPE('d','i','s','k')
#define kQ3GeometryTypeEllipse          Q3_OBJECT_TYPE('e','l','p','s')
#define kQ3GeometryTypeEllipsoid        Q3_OBJECT_TYPE('e','l','p','d')
#define kQ3GeometryTypeGeneralPolygon   Q3_OBJECT_TYPE('g','p','g','n')
#define kQ3GeometryTypeLine             Q3_OBJECT_TYPE('l','i','n','e')
#define kQ3GeometryTypeMarker           Q3_OBJECT_TYPE('m','r','k','r')
#define kQ3GeometryTypeMesh             Q3_OBJECT_TYPE('m','e','s','h')
#define kQ3GeometryTypeNURBCurve        Q3_OBJECT_TYPE('n','r','b','c')
#define kQ3GeometryTypeNURBPatch        Q3_OBJECT_TYPE('n','r','b','p')
#define kQ3GeometryTypePixmapMarker     Q3_OBJECT_TYPE('m','r','k','p')
#define kQ3GeometryTypePoint            Q3_OBJECT_TYPE('p','n','t',' ')
#define kQ3GeometryTypePolygon          Q3_OBJECT_TYPE('p','l','y','g')
```

```
#define kQ3GeometryTypePolyhedron       Q3_OBJECT_TYPE('p','l','h','d')
#define kQ3GeometryTypePolyLine         Q3_OBJECT_TYPE('p','l','y','l')
#define kQ3GeometryTypeTorus            Q3_OBJECT_TYPE('t','o','r','s')
#define kQ3GeometryTypeTriangle         Q3_OBJECT_TYPE('t','r','n','g')
#define kQ3GeometryTypeTriGrid          Q3_OBJECT_TYPE('t','r','i','g')
#define kQ3GeometryTypeTriMesh          Q3_OBJECT_TYPE('t','m','s','h')
```

Constant descriptions

`kQ3GeometryTypeBox` A box. See “Boxes” (page 301) for information about boxes.

`kQ3GeometryTypeCone`
A cone. See “Cones” (page 325) for information about cones.

`kQ3GeometryTypeCylinder`
A cylinder. See “Cylinders” (page 322) for information about cylinders.

`kQ3GeometryTypeDisk`
A disk. See “Disks” (page 323) for information about disks.

`kQ3GeometryTypeEllipse`
An ellipse. See “Ellipses” (page 314) for information about ellipses.

`kQ3GeometryTypeEllipsoid`
An ellipsoid. See “Ellipsoids” (page 320) for information about ellipsoids.

`kQ3GeometryTypeGeneralPolygon`
A general polygon. See “General Polygons” (page 299) for information about general polygons.

`kQ3GeometryTypeLine`
A line. See “Lines” (page 295) for information about lines.

`kQ3GeometryTypeMarker`
A bitmap marker. See “Markers” (page 329) for information about bitmap markers.

`kQ3GeometryTypeMesh`
A mesh. See “Meshes” (page 305) for information about meshes.

`kQ3GeometryTypeNURBCurve`
A NURB curve. See “NURB Curves” (page 315) for information about NURB curves.

`kQ3GeometryTypeNURBPatch`
A NURB patch. See “NURB Patches” (page 317) for information about NURB patches.

`kQ3GeometryTypePixmapMarker`
A pixmap marker. See “Markers” (page 329) for information about pixmap markers.

`kQ3GeometryTypePoint`
A point. See “Point Objects” (page 295) for information about point.

`kQ3GeometryTypePolygon`
A simple polygon. See “Simple Polygons” (page 298) for information about simple polygons.

`kQ3GeometryTypePolyhedron`
A polyhedron. See “Polyhedra” (page 311) for information about polyhedra.

`kQ3GeometryTypePolyLine`
A polyline. See “Polylines” (page 296) for information about polylines.

`kQ3GeometryTypeTorus`
A torus. See “Tori” (page 326) for information about tori.

`kQ3GeometryTypeTriangle`
A triangle. See “Triangles” (page 297) for information about triangles.

`kQ3GeometryTypeTriGrid`
A trigrid. See “Trigrids” (page 304) for information about trigrids.

`kQ3GeometryTypeTriMesh`
A trimesh. See “Trimeshes” (page 307) for information about trimeshes.

Pixel Types

The `pixelType` field of a pixmap or a storage pixmap specifies the type of pixel in the pixmap. You can use these constants to specify a pixel type:

```
typedef enum TQ3PixelType {
    kQ3PixelTypeRGB32                   = 0,
    kQ3PixelTypeARGB32                  = 1,
```

```
    kQ3PixelTypeRGB16                   = 2,
    kQ3PixelTypeARGB16                  = 3.
    kQ3PixelTypeRGB16_565               = 4,
    kQ3PixelTypeRGB24                   = 5
} TQ3PixelType;
```

Constant descriptions

`kQ3PixelTypeRGB32` A pixel occupies 32 bits of memory, with the red component in bits 23 through 16, the green component in bits 15 through 8, and the blue component in bits 7 through 0. There is no per-pixel alpha channel value. As a result, the pixmap (perhaps defining a texture) is treated as opaque. (You can, however, apply transparency to the pixmap using the alpha channel values of a triangle vertex, for instance.)

`kQ3PixelTypeARGB32` A pixel occupies 32 bits of memory, with the red component in bits 23 through 16, the green component in bits 15 through 8, and the blue component in bits 7 through 0. In addition, the pixel's alpha channel value is in bits 31 through 24. When the alpha value is 255, the pixmap is opaque; when the alpha value is 0, the pixmap is completely transparent.

`kQ3PixelTypeRGB16` A pixel occupies 16 bits of memory, with the red component in bits 14 through 10, the green component in bits 9 through 5, and the blue component in bits 4 through 0. There is no per-pixel alpha channel value. As a result, the pixmap (perhaps defining a texture) is treated as opaque. (You can, however, apply transparency to the pixmap using the alpha channel values of a triangle vertex, for instance.)

`kQ3PixelTypeARGB16` A pixel occupies 16 bits of memory, with the red component in bits 14 through 10, the green component in bits 9 through 5, and the blue component in bits 4 through 0. In addition, the pixel's alpha channel value is in bit 15. When the alpha value is 1, the pixmap is opaque; when the alpha value is 0, the pixmap is completely transparent.

`kQ3PixelTypeRGB16_565` A pixel occupies 16 bits of memory, with the red component in bits 15 through 11, the green component in bits 10 through 5, and the blue component in bits 4 through 0. There is no per-pixel alpha channel value. This pixel type is currently defined only for Windows 32 devices.

`kQ3PixelTypeRGB24` A pixel occupies 24 bits of memory, with the red component in bits 23 through 16, the green component in bits 15 through 8, and the blue component in bits 7 through 0. There is no per-pixel alpha channel value. This pixel type is currently defined only for Windows 32 devices.

Endian Types

The `bitOrder` field of a bitmap, a pixmap, or a storage pixmap indicates the order in which the bits in a byte are addressed. This field must contain one of these constants:

```
typedef enum TQ3Endian {
    kQ3EndianBig,
    kQ3EndianLittle
} TQ3Endian;
```

Constant descriptions

`kQ3EndianBig` The bits are addressed in a big-endian manner (that is, each field is addressed by referring to its most significant bit).

`kQ3EndianLittle` The bits are addressed in a little-endian manner (that is, each field is addressed by referring to its least significant bit).

General Polygon Shape Hints

A general polygon has a **shape hint** associated with it that specifies the shape of the general polygon. A general polygon's shape hint may be used by a renderer to optimize drawing the polygon.

```
typedef enum TQ3GeneralPolygonShapeHint {
    kQ3GeneralPolygonShapeHintComplex,
    kQ3GeneralPolygonShapeHintConcave,
    kQ3GeneralPolygonShapeHintConvex
} TQ3GeneralPolygonShapeHint;
```

Constant descriptions

`kQ3GeneralPolygonShapeHintComplex`
The general polygon consists of more than one contour, is

self-intersecting, or is not known to be either concave or convex.

`kQ3GeneralPolygonShapeHintConcave`
The general polygon has exactly one contour, which is concave.

`kQ3GeneralPolygonShapeHintConvex`
The general polygon has exactly one contour, which is convex.

End Caps Masks

Some geometric objects (for example, cones and cylinders) have boundaries that delimit the object and that are distinct from the surface of the object itself. These boundaries are the object's **end caps,** of type `TQ3EndCap`.

Note
The term *end caps* is potentially confusing, because it applies also to the interior portions of partial solids (that is, a solid object whose `uMin` field is greater than 0.0 or whose `uMax` field is less than 1.0). ◆

When defining these geometric objects, you specify the kind of end caps by setting the `caps` field to some combination of these constants:

```
typedef enum TQ3EndCapMasks {
    kQ3EndCapNone                       = 0,
    kQ3EndCapMaskTop                    = 1 << 0,
    kQ3EndCapMaskBottom                 = 1 << 1,
    kQ3EndCapMaskInterior               = 1 << 2
} TQ3EndCapMasks;
```

Constant descriptions

`kQ3EndCapNone` The specified geometric object has no end caps.

`kQ3EndCapMaskTop` The specified geometric object has an end cap at the top.

`kQ3EndCapMaskBottom`
The specified geometric object has an end cap at the bottom.

`kQ3EndCapMaskInterior`
The specified geometric object has an end cap at the interior.

IMPORTANT
Some of these constants are not applicable to some geometric objects. For instance, the mask `kQ3EndCapMaskTop`, if set in the `caps` field of a cone, is ignored. ▲

Polyhedron Edge Masks

The `edgeFlag` field of the polyhedron triangle data structure contains a value that indicates which edges of a polyhedral triangle are to be rendered. You specify a value for that field using a combination of these edge masks:

```
typedef enum TQ3PolyhedronEdgeMasks {
    kQ3PolyhedronEdgeNone               = 0,
    kQ3PolyhedronEdge01                 = 1 << 0,
    kQ3PolyhedronEdge12                 = 1 << 1,
    kQ3PolyhedronEdge20                 = 1 << 2,
    kQ3PolyhedronEdgeAll                = kQ3PolyhedronEdge01 |
                                          kQ3PolyhedronEdge12 |
                                          kQ3PolyhedronEdge20
} TQ3PolyhedronEdgeMasks;
```

Constant descriptions

`kQ3PolyhedronEdgeNone`
Render none of the edges of the triangle.

`kQ3PolyhedronEdge01`
Render the edge between the first and second vertices (that is, between `vertices[0]` and `vertices[1]`).

`kQ3PolyhedronEdge12`
Render the edge between the second and third vertices (that is, between `vertices[1]` and `vertices[2]`).

`kQ3PolyhedronEdge20`
Render the edge between the third and first vertices (that is, between `vertices[2]` and `vertices[0]`).

`kQ3PolyhedronEdgeAll`
Render all of the edges of the triangle.

Data Structures

This section describes the data structures that define the QuickDraw 3D geometric objects. QuickDraw 3D defines the following primitive objects:

- points
- lines
- polylines
- triangles
- simple and general polygons
- boxes
- trigrids
- meshes
- trimeshes
- polyhedra
- ellipses
- NURB curves
- NURB patches
- ellipsoids
- cylinders
- disks
- cones
- tori
- markers

Each of these QuickDraw 3D geometric objects has a set of attributes associated with it. The set of attributes specifies information about the appearance of the objects (for example, its color and transparency). You can edit an object's attributes by calling the functions `Q3Geometry_GetAttributeSet` and `Q3Geometry_SetAttributeSet`.

Note
Don't confuse a QuickDraw 3D geometric object (which contains attribute information) with some corresponding standard geometric object (which doesn't contain attribute information). For example, the `TQ3Point3D` data type defines the standard three-dimensional Cartesian point. The associated QuickDraw 3D geometric object is defined by the `TQ3PointData` data type. For simplicity, the QuickDraw 3D types are usually referred to by their usual geometric names. When it is necessary to distinguish QuickDraw 3D types from standard mathematical types, the QuickDraw 3D type will be referred to as an *object*. For example, the `TQ3Point3D` data type defines a point and the `TQ3PointData` data type defines a point object. ◆

Points

QuickDraw 3D defines two- and three-dimensional points in the usual way, as pairs and triples of floating-point numbers. You'll use the `TQ3Point3D` data type throughout the QuickDraw 3D application programming interfaces. You'll use the `TQ3Point2D` data type for defining two-dimensional points.

```
typedef struct TQ3Point2D {
    float           x;
    float           y;
} TQ3Point2D;

typedef struct TQ3Point3D {
    float           x;
    float           y;
    float           z;
} TQ3Point3D;
```

Field descriptions

`x`	The *x* coordinate (abscissa) of a point.
`y`	The *y* coordinate (ordinate) of a point.
`z`	The *z* coordinate of a point.

Rational Points

QuickDraw 3D defines three- and four-dimensional rational points as pairs and triples of floating-point numbers, together with a floating-point weight. You'll use the `TQ3RationalPoint4D` data type for defining control points of rational surfaces and solids. The `TQ3RationalPoint4D` data type represents homogeneous points in four-dimensional space. To get the equivalent three-dimensional point, divide the point's `x`, `y`, and `z` components by the `w` component. You'll use the `TQ3RationalPoint3D` data type to define control points of NURB trim curves.

```
typedef struct TQ3RationalPoint3D {
    float           x;
    float           y;
    float           w;
} TQ3RationalPoint3D;

typedef struct TQ3RationalPoint4D {
    float           x;
    float           y;
    float           z;
    float           w;
} TQ3RationalPoint4D;
```

Field descriptions

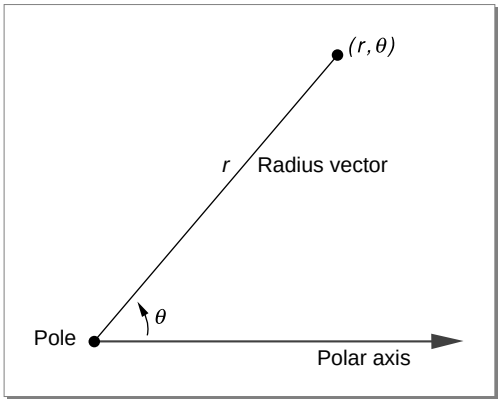
`x`	The *x* coordinate (abscissa) of a rational point.
`y`	The *y* coordinate (ordinate) of a rational point.
`z`	The *z* coordinate of a rational point.
`w`	The weight of a rational point.

Polar and Spherical Points

QuickDraw 3D defines polar and spherical points in the usual way. A **polar point** is a point in a plane described using polar coordinates. As illustrated in Figure 4-12, a polar point is uniquely determined by a distance *r* along a ray (the **radius vector**) that forms a given angle θ with a polar axis. Polar points are defined by the `TQ3PolarPoint` data type.

Note
Given a fixed polar origin and polar axis, a polar point can be described by infinitely many polar coordinates. For example, the polar point (5, π) is the same as the polar point (5, 3π). ◆

Figure 4-12 A planar point described with polar coordinates



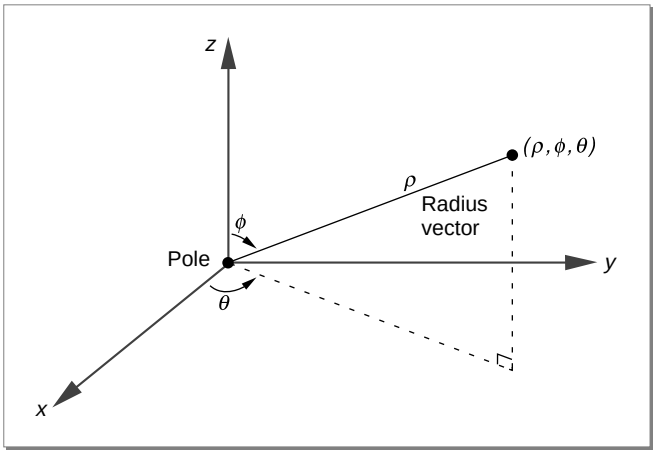

```
typedef struct TQ3PolarPoint {
    float           r;
    float           theta;
} TQ3PolarPoint;
```

Field descriptions

`r`	The distance along the radius vector from the polar origin to the polar point.
`theta`	The angle, in radians, between the polar axis and the radius vector.

A **spherical point** is a point in space described using spherical coordinates. As illustrated in Figure 4-13, a spherical point is uniquely determined by a distance ρ along a ray (the **radius vector**) that forms a given angle θ with the *x* axis and another given angle ϕ with the *z* axis. Spherical points are defined by the `TQ3SphericalPoint` data type.

Figure 4-13 A spatial point described with spherical coordinates




```
typedef struct TQ3SphericalPoint {
    float           rho;
    float           theta;
    float           phi;
} TQ3SphericalPoint;
```

Field descriptions

`rho`	The distance along the radius vector from the polar origin to the spherical point.
`theta`	The angle, in radians, between the *x* axis and the projection of the radius vector onto the *xy* plane.
`phi`	The angle, in radians, between the *z* axis and the radius vector.

Vectors

QuickDraw 3D defines two- and three-dimensional vectors in the usual way, as pairs and triples of floating-point numbers. Vectors are defined by data types distinct from those that define points primarily for conceptual clarity and for enforcing the correct usage of vectors in mathematical routines. Vectors are defined by the `TQ3Vector2D` and `TQ3Vector3D` data types.

```
typedef struct TQ3Vector2D {
    float           x;
    float           y;
} TQ3Vector2D;

typedef struct TQ3Vector3D {
    float           x;
    float           y;
    float           z;
} TQ3Vector3D;
```

Field descriptions

`x`	The *x* scalar component of a vector.
`y`	The *y* scalar component of a vector.
`z`	The *z* scalar component of a vector.

Quaternions

QuickDraw 3D defines quaternions as quadruples of floating-point numbers. A quaternion is defined by the `TQ3Quaternion` data type.

Note
For a description of quaternions and their use in computer graphics, see the article by Hart, Francis, and Kaufman listed in the bibliography. ◆

```
typedef struct TQ3Quaternion {
    float           w;
    float           x;
    float           y;
    float           z;
} TQ3Quaternion;
```

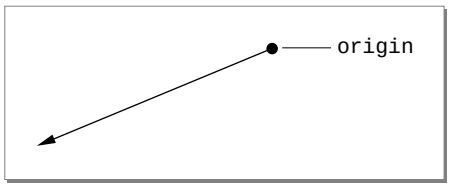
Field descriptions

`w`	The *w* component of a quaternion.
`x`	The *x* component of a quaternion.
`y`	The *y* component of a quaternion.
`z`	The *z* component of a quaternion.

Rays

QuickDraw 3D defines a ray as a point of origin and a direction. A ray is defined by the `TQ3Ray3D` data type. Figure 4-14 shows a ray.

Figure 4-14 A ray




```
typedef struct TQ3Ray3D {
    TQ3Point3D                      origin;
    TQ3Vector3D                     direction;
} TQ3Ray3D;
```

Field descriptions

`origin`	The origin of the ray.
`direction`	The direction of the ray.

Parametric Points

QuickDraw 3D defines the `TQ3Param2D` and `TQ3Param3D` data structures to represent two- and three-dimensional parametric points.

```
typedef struct TQ3Param2D {
    float           u;
    float           v;
} TQ3Param2D;

typedef struct TQ3Param3D {
    float           u;
    float           v;
    float           w;
} TQ3Param3D;
```

Field descriptions

`u`	The *u* component of a parametric point.
`v`	The *v* component of a parametric point.
`w`	The *w* component of a parametric point.

Note
The *u*, *v*, and *w* components are sometimes represented by the letters *s*, *t*, and *u*, respectively. This book always uses *u*, *v*, and *w*. ◆

Tangents

QuickDraw 3D defines the `TQ3Tangent2D` and `TQ3Tangent3D` data structures to represent two- and three-dimensional parametric surface tangents. A **surface tangent** indicates the directions of changing *u*, *v*, and *w* parameters on a surface.

```
typedef struct TQ3Tangent2D {
    TQ3Vector3D                     uTangent;
    TQ3Vector3D                     vTangent;
} TQ3Tangent2D;

typedef struct TQ3Tangent3D {
    TQ3Vector3D                     uTangent;
    TQ3Vector3D                     vTangent;
    TQ3Vector3D                     wTangent;
} TQ3Tangent3D;
```

Field descriptions

`uTangent`	The tangent in the *u* direction.
`vTangent`	The tangent in the *v* direction.
`wTangent`	The tangent in the *w* direction.

Vertices

A vertex is a dimensionless position in three-dimensional space at which two or more lines (for instance, edges) intersect, with an optional set of vertex attributes. Vertices are defined by the `TQ3Vertex3D` data type.

```
typedef struct TQ3Vertex3D {
    TQ3Point3D                      point;
    TQ3AttributeSet                 attributeSet;
} TQ3Vertex3D;
```

Field descriptions

`point`	A three-dimensional point.
`attributeSet`	A set of attributes for the vertex. The value in this field is `NULL` if no vertex attributes are defined.

Matrices

QuickDraw 3D defines 3-by-3 and 4-by-4 matrices as structures containing two-dimensional arrays of floating-point numbers as the single field in the structure. This convention allows for easy structure copying and for passing matrix parameters either by value or by reference. In a C language two-dimensional array, the second index varies fastest; accordingly, you can think of the first index as representing the matrix row and the second index as representing the matrix column. For example, consider the 3-by-3 matrix *A* defined like this:

$$A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

Here, *A*[0][0] is the matrix element *a*, and *A*[2][1] is the matrix element *h*.

Matrices are defined by the `TQ3Matrix3x3` and `TQ3Matrix4x4` data types.

Note
Remember that arrays in C are indexed starting with 0. ◆

```
typedef struct TQ3Matrix3x3 {
    float                           value[3][3];
} TQ3Matrix3x3;

typedef struct TQ3Matrix4x4 {
    float                           value[4][4];
} TQ3Matrix4x4;
```

Field descriptions

`value` An array of floating-point values that define the matrix.

Bitmaps and Pixel Maps

QuickDraw 3D defines bitmaps and pixmaps to specify the images used to define markers, textures, and other objects. A **bitmap** is a two-dimensional array of values, each of which represents the state of one pixel. A bitmap is defined by the `TQ3Bitmap` data type.

```
typedef struct TQ3Bitmap {
    unsigned char                   *image;
    unsigned long                   width;
    unsigned long                   height;
    unsigned long                   rowBytes;
    TQ3Endian                       bitOrder;
} TQ3Bitmap;
```

Field descriptions

`image` The address of a two-dimensional block of memory that contains the bitmap image. The size, in bytes, of this block must be exactly the product of the values in the `height` and `rowBytes` fields.

`width` The width, in bits, of the bitmap.

`height` The height of the bitmap.

`rowBytes` The distance, in bytes, from the beginning of one row of the image data to the beginning of the next row of the image data. Each new row in the image begins at an unsigned character that follows (but not necessarily immediately follows) the last unsigned character of the previous row. The minimum value of this field is the size of the image (as returned, for example, by the `Q3Bitmap_GetImageSize` function) divided by the value of the `height` field.

`bitOrder` The order in which the bits in a byte are addressed. See "Endian Types" (page 279) for a description of the available bit orders.

A **pixel map** (or, more briefly, a **pixmap**) is a two-dimensional array of values, each of which represents the color of one pixel. A pixmap is defined by the `TQ3Pixmap` data type.

```
typedef struct TQ3Pixmap {
    void                                *image;
    unsigned long                       width;
    unsigned long                       height;
    unsigned long                       rowBytes;
    unsigned long                       pixelSize;
    TQ3PixelType                        pixelType;
    TQ3Endian                           bitOrder;
    TQ3Endian                           byteOrder;
} TQ3Pixmap;
```

Field descriptions

`image` The address of a two-dimensional block of memory that contains the pixmap image. The size, in bytes, of this block must be exactly the product of the values in the `height` and `rowBytes` fields.

`width` The width, in pixels, of the pixmap.

`height` The height, in pixels, of the pixmap.

`rowBytes` The distance, in bytes, from the beginning of one row of the image data to the beginning of the next row of the image data. The minimum value of this field depends on the values of the `width` and `pixelSize` fields. You can use the following C language macro to determine a value for this field:

```
#define Pixmap_GetRowBytes(width, pixelSize) \
        ((pixelSize) < 8) \
        ? (((width) / (8 / (pixelSize))) + \
        ((width) % (8 / (pixelSize)) > 0)) \
        : (width * ((pixelSize) / 8))
```

`pixelSize` The size, in bits, of a pixel.

`pixelType` The type of a pixel. See “Pixel Types” (page 277) for a description of the available pixel types. This field must match the size specified in the `pixelSize` field.

`bitOrder` The order in which the bits in a byte are addressed. See “Endian Types” (page 279) for a description of the available bit orders.

`byteOrder` The order in which the bytes in a word are addressed. See "Endian Types" (page 279) for a description of the available byte orders.

A **storage pixel map** (or, more briefly, a **storage pixmap**) is a pixmap whose data is contained in a storage object. A storage pixmap is defined by the `TQ3StoragePixmap` data type.

```
typedef struct TQ3StoragePixmap {
    TQ3StorageObject                    image;
    unsigned long                       width;
    unsigned long                       height;
    unsigned long                       rowBytes;
    unsigned long                       pixelSize;
    TQ3PixelType                        pixelType;
    TQ3Endian                           bitOrder;
    TQ3Endian                           byteOrder;
} TQ3StoragePixmap;
```

Field descriptions

`image` A storage object that contains the pixmap image. The size, in bytes, of this file must be exactly the product of the values in the `height` and `rowBytes` fields.

`width` The width, in pixels, of the pixmap.

`height` The height, in pixels, of the pixmap.

`rowBytes` The distance, in bytes, from the beginning of one row of the image data to the beginning of the next row of the image data. The minimum value of this field depends on the values of the `width` and `pixelSize` fields. You can use the following C language macro to determine a value for this field:

```
#define Pixmap_GetRowBytes(width, pixelSize) \
        ((pixelSize) < 8) \
        ? (((width) / (8 / (pixelSize))) + \
        ((width) % (8 / (pixelSize)) > 0)) \
        : (width * ((pixelSize) / 8))
```

`pixelSize` The size, in bits, of a pixel.

`pixelType`	The type of a pixel. See "Pixel Types" (page 277) for a description of the available pixel types. This field must match the size specified in the `pixelSize` field.
`bitOrder`	The order in which the bits in a byte are addressed. See "Endian Types" (page 279) for a description of the available bit orders.
`byteOrder`	The order in which the bytes in a word are addressed. See "Endian Types" (page 279) for a description of the available byte orders.

Areas and Plane Equations

A two-dimensional area is defined by the `TQ3Area` data type.

```
typedef struct TQ3Area {
    TQ3Point2D                      min;
    TQ3Point2D                      max;
} TQ3Area;
```

Field descriptions

`min`	A two-dimensional point.
`max`	A two-dimensional point.

A plane equation is defined by the `TQ3PlaneEquation` data type.

```
typedef struct TQ3PlaneEquation {
    TQ3Vector3D                     normal;
    float                           constant;
} TQ3PlaneEquation;
```

Field descriptions

`normal`	The vector that is normal (perpendicular) to the plane.
`constant`	The plane constant. A **plane constant** is the value d in the plane equation $ax+by+cz+d = 0$. The coefficients a, b, and c are the x, y, and z components of the normal vector.

Point Objects

A point object is simply a dimensionless position in three-dimensional space, with an optional set of attributes. A point object is defined by the `TQ3PointData` data type. See "Creating and Editing Points," beginning on page 334 for a description of the routines you can use to create and edit point objects.

```
typedef struct TQ3PointData {
    TQ3Point3D                  point;
    TQ3AttributeSet             pointAttributeSet;
} TQ3PointData;
```

Field descriptions

`point` A three-dimensional point.

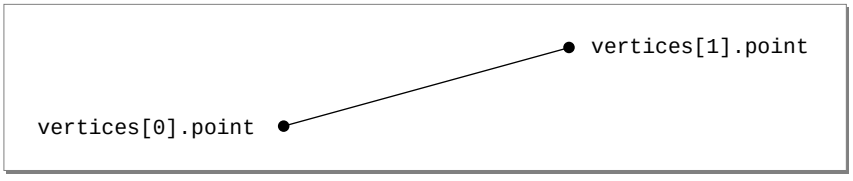
`pointAttributeSet`

A set of attributes for the point. The value in this field is `NULL` if no point attributes are defined.

Lines

A line is a straight segment in three-dimensional space defined by its two endpoints, with an optional set of attributes. (In addition, each vertex can have a set of attributes.) A line is defined by the `TQ3LineData` data type. See "Creating and Editing Lines," beginning on page 337 for a description of the routines you can use to create and edit lines. Figure 4-15 shows a line.

Figure 4-15 A line




```
typedef struct TQ3LineData {
    TQ3Vertex3D                 vertices[2];
    TQ3AttributeSet             lineAttributeSet;
} TQ3LineData;
```

Field descriptions

`vertices` An array of two vertices.

`lineAttributeSet`

A set of attributes for the line. The value in this field is `NULL` if no line attributes are defined.

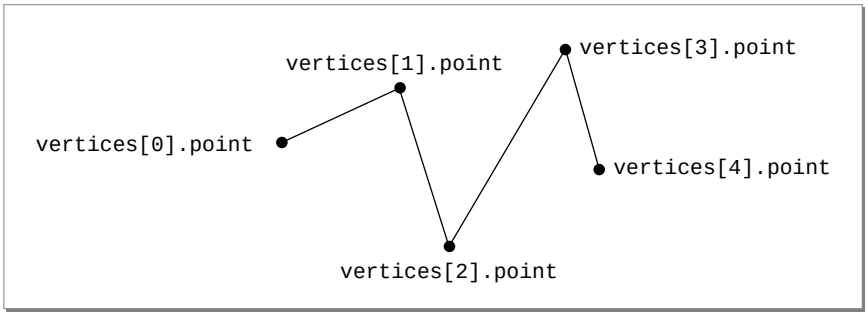
Polylines

A polyline is a collection of *n* lines defined by the *n*+1 points that define the endpoints of each line segment. The entire polyline can have a set of attributes, and each line segment in the polyline also can have a set of attributes. (In addition, each vertex can have a set of attributes.) A polyline is defined by the `TQ3PolyLineData` data type. See "Creating and Editing Polylines," beginning on page 342 for a description of the routines you can use to create and edit polylines. Figure 4-16 shows a polyline.

IMPORTANT

A polyline is not closed. The last point should not be connected to the first. ▲

Figure 4-16 A polyline




```
typedef struct TQ3PolyLineData {
    unsigned long                   numVertices;
    TQ3Vertex3D                     *vertices;
```

```
    TQ3AttributeSet                 *segmentAttributeSet;
    TQ3AttributeSet                 polyLineAttributeSet;
} TQ3PolyLineData;
```

Field descriptions

`numVertices` The number of vertices in the polyline. The value of this field must be at least 2.

`vertices` A pointer to an array of vertices which define the polyline.

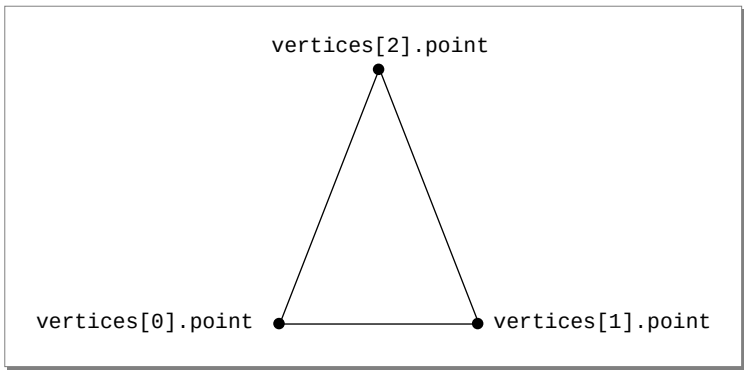
`segmentAttributeSet`
A pointer to an array of segment attribute sets. If no segments in the polyline are to have attributes, this field should contain the value `NULL`. If any of the segments have attributes, this field should contain a pointer to an array (containing `numVertices` – 1 elements) of attributes sets; the array element for segments with no attributes should be set to `NULL`.

`polyLineAttributeSet`
A set of attributes for the polyline. The value in this field is `NULL` if no polyline attributes are defined.

Triangles

A triangle is a closed plane figure defined by the three edges that connect three vertices. The entire triangle can have a set of attributes, and any or all of the three vertices can also have a set of attributes. A triangle is defined by the `TQ3TriangleData` data type. See "Creating and Editing Triangles," beginning on page 349 for a description of the routines you can use to create and edit triangles. Figure 4-17 shows a triangle.

Figure 4-17 A triangle




```
typedef struct TQ3TriangleData {
    TQ3Vertex3D                     vertices[3];
    TQ3AttributeSet                 triangleAttributeSet;
} TQ3TriangleData;
```

Field descriptions

`vertices` The three vertices that define the three sides of the triangle.

`triangleAttributeSet`
A set of attributes for the triangle. The value in this field is `NULL` if no triangle attributes are defined.

Simple Polygons

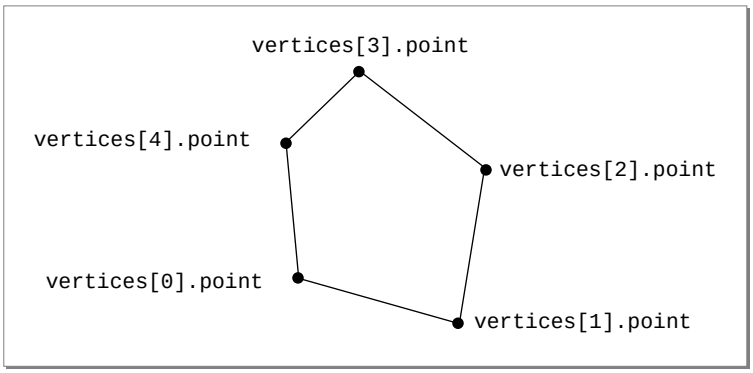
A simple polygon is a closed plane figure defined by a list of vertices. (In other words, a simple polygon is a polygon defined by a single contour.) The edges of a simple polygon should not intersect themselves or you will get unpredictable results when operating on the polygon. In addition, a simple polygon must be convex.

The entire simple polygon can have a set of attributes, and any or all of the vertices defining the polygon can have a set of attributes.

A simple polygon is defined by the `TQ3PolygonData` data type. See "Creating and Editing Simple Polygons," beginning on page 354 for a description of the

routines you can use to create and edit simple polygons. Figure 4-18 shows a simple polygon.

Figure 4-18 A simple polygon




```
typedef struct TQ3PolygonData {
    unsigned long                   numVertices;
    TQ3Vertex3D                     *vertices;
    TQ3AttributeSet                 polygonAttributeSet;
} TQ3PolygonData;
```

Field descriptions

`numVertices` The number of vertices in the simple polygon. The value of this field must be at least 3.

`vertices` A pointer to an array of vertices that define the simple polygon.

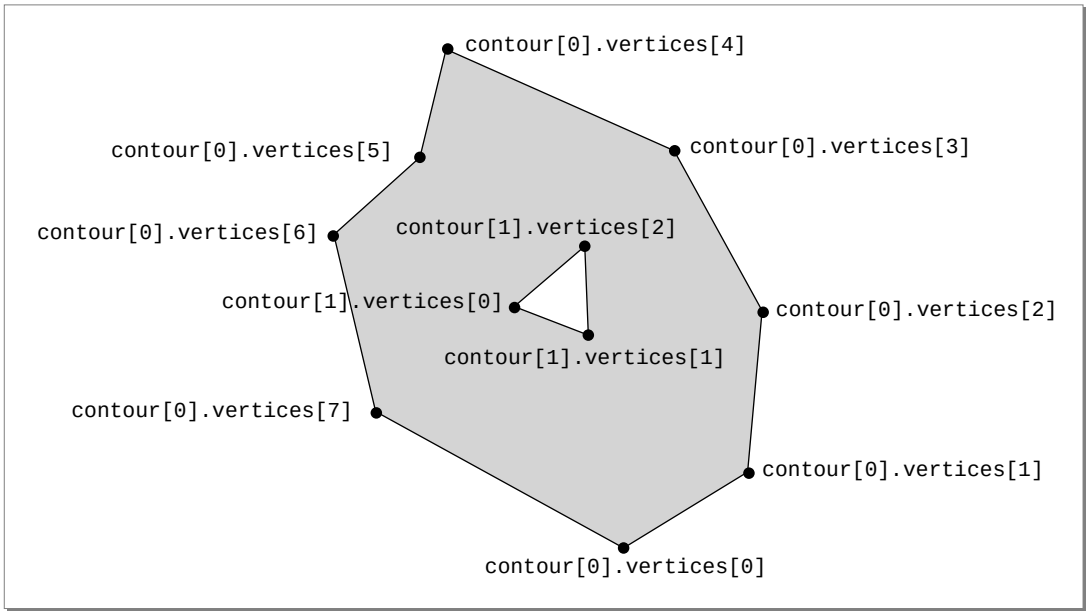
`polygonAttributeSet` A set of attributes for the simple polygon. The value in this field is `NULL` if no polygon attributes are defined.

General Polygons

A general polygon is a closed plane figure defined by one or more lists of vertices. (In other words, a general polygon is a polygon defined by one or more contours.) Each contour may be concave or convex, and contours may be

nested. In addition, a general polygon's contours may overlap or be disjoint. All contours, however, must be coplanar. A general polygon can have holes in it; if it does, the **even-odd rule** is used to determine which parts are inside the polygon. Figure 4-19 shows a general polygon.

Figure 4-19 A general polygon



The entire general polygon can have a set of attributes, and any or all of the vertices of any contour can have a set of attributes.

The orientation of a general polygon is determined by the order of the first three noncolinear and noncoincident vertices in the first contour of the general polygon and by the current orientation style of the model containing the polygon. See the chapter "Style Objects" for more information on orientation styles.

A general polygon is defined by the `TQ3GeneralPolygonData` data type. See "Creating and Editing General Polygons," beginning on page 360 for a description of the routines you can use to create and edit general polygons.

```
typedef struct TQ3GeneralPolygonData {
    unsigned long                           numContours;
    TQ3GeneralPolygonContourData            *contours;
    TQ3GeneralPolygonShapeHint              shapeHint;
    TQ3AttributeSet                         generalPolygonAttributeSet;
} TQ3GeneralPolygonData;
```

Field descriptions

`numContours` The number of contours in the general polygon. The value of this field must be at least 1.

`contours` A pointer to an array of contours that define the general polygon.

`shapeHint` A constant that specifies the shape of the general polygon. A general polygon's shape hint may be used by a renderer to optimize drawing the polygon. See "General Polygon Shape Hints" (page 279) for information about general polygon shape hints.

`generalPolygonAttributeSet`
A set of attributes for the general polygon. The value in this field is `NULL` if no general polygon attributes are defined.

The elements of the array of contours pointed to by the `contours` field are of type `TQ3GeneralPolygonContourData`, defined as follows:

```
typedef struct TQ3GeneralPolygonContourData {
    unsigned long                       numVertices;
    TQ3Vertex3D                         *vertices;
} TQ3GeneralPolygonContourData;
```

Field descriptions

`numVertices` The number of vertices in the contour. The value of this field must be at least 3.

`vertices` A pointer to an array of vertices that define the contour.

Boxes

A box is a three-dimensional object defined by an origin (that is, a corner of the box) and three vectors that define the edges of the box that meet in that corner. A box defined by three mutually orthogonal vectors is a regular rectangular prism. A box defined by nonorthogonal vectors is a general parallelepiped.

The entire box can have a set of attributes. In addition, you may specify an array of attributes to be applied to each face of the box. (In this way, for example, you can give each face of the box a different color.)

A box is defined by the `TQ3BoxData` data type. See "Creating and Editing Boxes," beginning on page 367 for a description of the routines you can use to create and edit boxes. Figure 4-20 shows a box.

Figure 4-20 A box

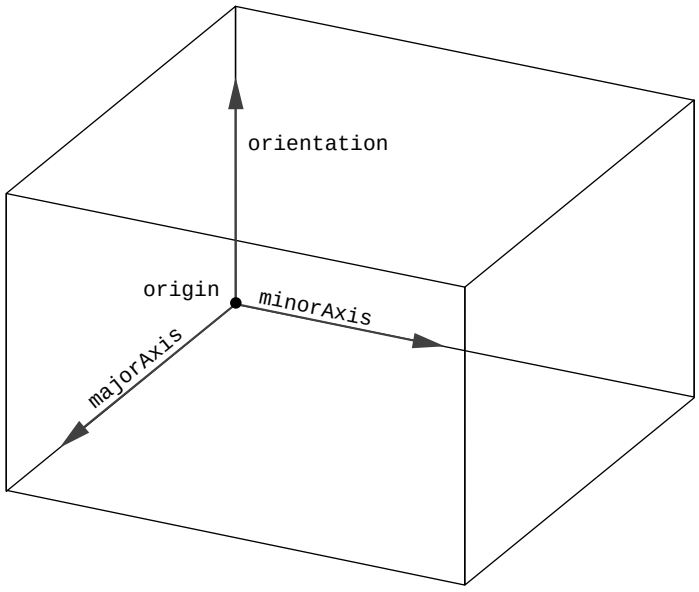
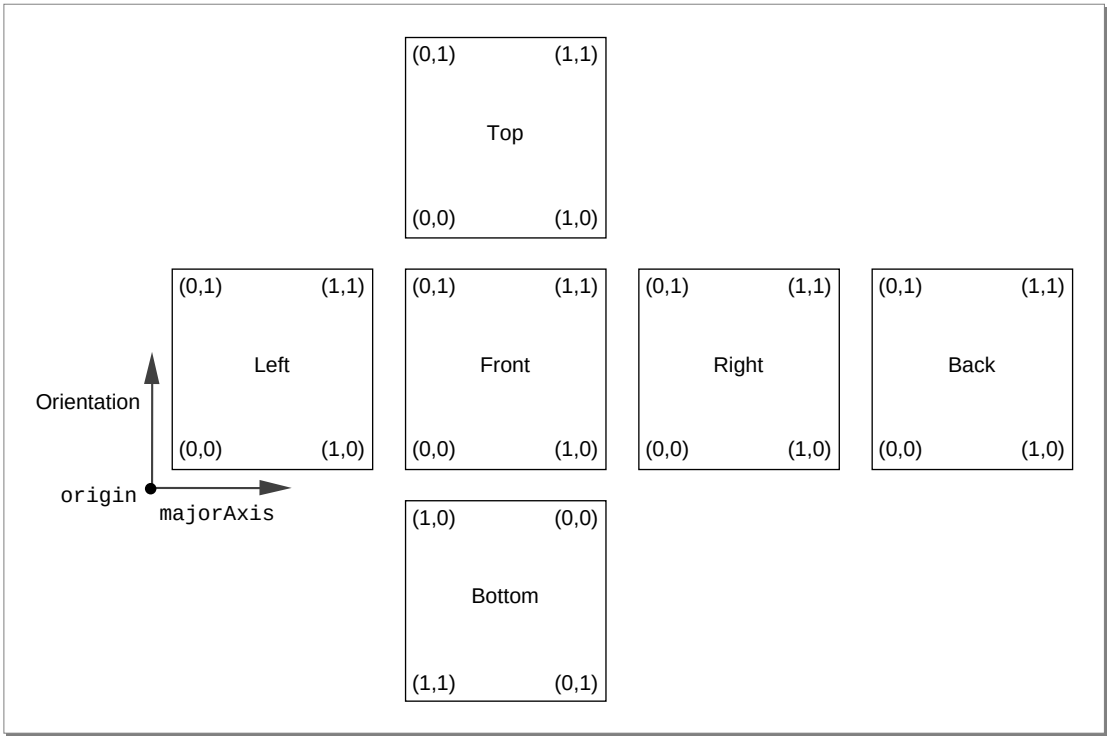


Figure 4-21 (page 303) shows the standard surface parameterization of a box.

Figure 4-21 The standard surface parameterization of a box




```
typedef struct TQ3BoxData {
    TQ3Point3D                  origin;
    TQ3Vector3D                 orientation;
    TQ3Vector3D                 majorAxis;
    TQ3Vector3D                 minorAxis;
    TQ3AttributeSet             *faceAttributeSet;
    TQ3AttributeSet             boxAttributeSet;
} TQ3BoxData;
```

Field descriptions

`origin` The origin of the box.

`orientation` The orientation of the box.

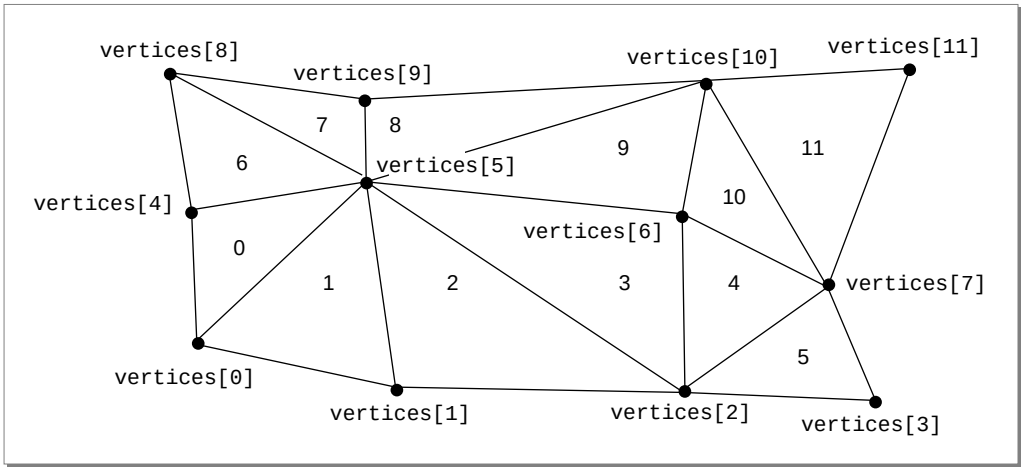
`majorAxis` The major axis of the box.

`minorAxis`	The minor axis of the box.
`faceAttributeSet`	A pointer to a six-element array of face attributes. The attributes apply to the faces of the box specified in the following order: left, right, front, back, top, bottom.
`boxAttributeSet`	A set of attributes for the box. The value in this field is `NULL` if no box attributes are defined.

Trigrids

A trigrid is a rectangular grid composed of triangular facets. The triangulation should be **serpentine** (that is, quadrilaterals are divided into triangles in an alternating fashion) to reduce shading artifacts when using Gouraud or Phong shading. Figure 4-22 shows a trigrid.

Figure 4-22 A trigrid



The entire trigrid can have a set of attributes. You may specify an array of attributes that apply to each facet of the trigrid. In this way, for example, you can give each facet of the trigrid a different color. In addition, any or all of the vertices can have a set of attributes.

A trigrid is defined by the `TQ3TriGridData` data type. See "Creating and Editing Trigrids," beginning on page 375 for a description of the routines you can use to create and edit trigrids.

```
typedef struct TQ3TriGridData {
    unsigned long                   numRows;
    unsigned long                   numColumns;
    TQ3Vertex3D                     *vertices;
    TQ3AttributeSet                 *facetAttributeSet;
    TQ3AttributeSet                 triGridAttributeSet;
} TQ3TriGridData;
```

Field descriptions

`numRows` The number of rows of vertices.

`numColumns` The number of columns of vertices.

`vertices` A pointer to an array of vertices. The first vertex in the array is the lower-left corner of the trigrid. The vertices are listed in a rectangular order, first in the direction of increasing column and then in the direction of increasing row. The number of vertices is the product of the values in the `numRows` and `numColumns` fields.

`facetAttributeSet` A pointer to an array of facet attribute sets. If this value is not `NULL`, the array should contain $2 \times ((\texttt{numRows} - 1) \times (\texttt{numColumns} - 1))$ elements.

`triGridAttributeSet` A set of attributes for the trigrid. The value in this field is `NULL` if no trigrid attributes are defined.

Meshes

A mesh is a collection of vertices and faces that represent a topological polyhedron. The polyhedron does not need to be closed (that is, a mesh may have a boundary). The structure of a mesh is maintained privately by QuickDraw 3D, using the following structure types:

```
typedef struct TQ3MeshComponentPrivate *TQ3MeshComponent;
typedef struct TQ3MeshContourPrivate *TQ3MeshContour;
typedef struct TQ3MeshEdgeRepPrivate *TQ3MeshEdge;
typedef struct TQ3MeshVertexPrivate *TQ3MeshFace;
```

You create a new mesh by calling `Q3Mesh_New`. When first created, a mesh is empty—it contains no vertices, faces, or edges. These must be added by calling `Q3Mesh_VertexNew` and `Q3Mesh_FaceNew`.

Once a mesh has been created and populated with vertices and faces, you can access the data associated with it by using the following routines:

```
TQ3Status Q3Mesh_GetNumVertices
TQ3Status Q3Mesh_GetNumFaces
TQ3Status Q3Mesh_GetNumEdges
TQ3Status Q3Mesh_GetNumComponents
TQ3Status Q3Mesh_GetNumCorners
TQ3Status Q3Mesh_GetOrientable
```

`Q3Mesh_GetOrientable` returns the Boolean variable `orientable`, which is true only if the faces of a mesh can be consistently oriented. A tessellated Möbius strip and the surface of a Klein bottle are two classic examples of nonorientable meshes.

A mesh's attribute set can be accessed and set by the routines

```
TQ3Status Q3Geometry_GetAttributeSet
TQ3Status Q3Geometry_SetAttributeSet
```

In addition, each mesh vertex, face, edge, and corner of a mesh can have a set of attributes attached to it.

The routines you can use to create and alter meshes are described in "Creating and Editing Meshes," beginning on page 382.

IMPORTANT

QuickDraw 3D supports meshes primarily for interactive rendering of polygonal models, not for representing large polygonal databases. A mesh is always a retained object, never an immediate object. As a result, QuickDraw 3D does not supply routines to draw or write meshes. ▲

There is only one public data structure defined for meshes, the mesh iterator structure. You use the **mesh iterator structure** when you call any one of a large number of mesh iterators. The mesh iterator structure is defined by the `TQ3MeshIterator` data type.

```
typedef struct TQ3MeshIterator {
    void                                *var1;
    void                                *var2;
    void                                *var3;
    struct {
        void                            *field1;
        char                            field2[4];
    } var4;
} TQ3MeshIterator;
```

Field descriptions

`var1`	Reserved for use by Apple Computer, Inc.
`var2`	Reserved for use by Apple Computer, Inc.
`var3`	Reserved for use by Apple Computer, Inc.
`var4`	Reserved for use by Apple Computer, Inc.
`field1`	Reserved for use by Apple Computer, Inc.
`field2`	Reserved for use by Apple Computer, Inc.

Trimeshes

A **trimesh** is a collection of vertices, edges, and faces in which all faces are triangular. In other words, a trimesh is simply a mesh composed entirely of triangles. A trimesh is like a polyhedron in that its faces are defined indirectly, using indices into an array of vertices. Similarly, the edges of a trimesh are defined by an optional array of edge vertices.

The main difference between trimeshes and all other QuickDraw 3D primitives (including polyhedra) is that attributes for trimesh vertices, edges, and faces are not stored as objects of type `TQ3AttributeSet`; rather, those attributes are stored in arrays of explicit attribute data structures. Moreover, if any single vertex (or edge, or face) has an attribute of a specific non-custom type, then every vertex (or edge, or face) in the trimesh must also have an attribute of that type. This restriction can deleteriously affect the memory requirements of a large trimesh.

Note

See "Comparison of the Polyhedral Primitives" (page 247) for more information on the advantages and disadvantages of using trimeshes to model a polyhedral surface. ◆

A trimesh triangle data structure specifies information about a triangular face of a trimesh. A trimesh triangle data structure is defined by the `TQ3TriMeshTriangleData` data type.

```
typedef struct TQ3TriMeshTriangleData {
    unsigned long                       pointIndices[3];
} TQ3TriMeshTriangleData;
```

Field descriptions

`pointIndices` Three indices into an array of three-dimensional points. (The array is specified by the `points` field of the `TQ3TriMeshData` data structure.) These three points define the vertices of the face.

A trimesh edge data structure specifies information about an edge of a trimesh; it is defined by the `TQ3TriMeshEdgeData` data structure.

```
typedef struct TQ3TriMeshEdgeData {
    unsigned long                       pointIndices[2];
    unsigned long                       triangleIndices[2];
} TQ3TriMeshEdgeData;
```

Field descriptions

`pointIndices` Two indices into an array of three-dimensional points. (The array is specified by the `points` field of the `TQ3TriMeshData` data structure.) These two points define the endpoints of the edge.

`triangleIndices` Two indices into an array of trimesh triangle data structures, which contain information about the faces in the trimesh. (The array is specified by the `triangles` field of the `TQ3TriMeshData` data structure.) These two triangles define the two faces that contain the edge. When an edge abuts only one face (that is, when the edge is on a boundary of the trimesh), you can use the constant `kQ3ArrayIndexNULL` as the face index for the side of the edge that has no face attached to it.

Attributes for the parts of a trimesh are defined by a **trimesh attributes data structure,** of type `TQ3TriMeshAttributeData`.

```
typedef struct TQ3TriMeshAttributeData {
    TQ3AttributeType                    attributeType;
    void                                *data;
    char                                *attributeUseArray
} TQ3TriMeshAttributeData;
```

Field descriptions

`attributeType` The type of the attribute.

`data` A pointer to the attribute data.

`attributeUseArray` A pointer to an array of 0's and 1's that defines which vertices (or edges, or faces) have custom attributes.

When the `attributeType` field is set to a pre-defined (that is, non-custom) attribute type, the `attributeUseArray` field must contain `NULL`. When the `attributeType` field is set to a custom attribute type, it should contain a pointer to an array containing only the values 0 and 1. If an element in `attributeUseArray` has the value 1, then the corresponding vertex (or edge, or face) has a custom attribute; otherwise, if an element in this array has the value 0, the corresponding vertex (or edge, or face) doesn't have a custom attribute.

Finally, a trimesh is defined by the `TQ3TriMeshData` data type.

```
typedef struct TQ3TriMeshData {
    TQ3AttributeSet                     triMeshAttributeSet;
    unsigned long                       numTriangles;
    TQ3TriMeshTriangleData              *triangles;
    unsigned long                       numTriangleAttributeTypes;
    TQ3TriMeshAttributeData             *triangleAttributeTypes;
    unsigned long                       numEdges;
    TQ3TriMeshEdgeData                  *edges;
    unsigned long                       numEdgeAttributeTypes;
    TQ3TriMeshAttributeData             *edgeAttributeTypes;
    unsigned long                       numPoints;
    TQ3Point3D                          *points;
    unsigned long                       numVertexAttributeTypes;
    TQ3TriMeshAttributeData             *vertexAttributeTypes;
    TQ3BoundingBox                      bBox;
} TQ3TriMeshData;
```

Field descriptions

`triMeshAttributeSet`
A pointer to a trimesh attributes data structure that

contains information about the attributes of the entire trimesh. If the trimesh has no attributes, this field should be set to `NULL`.

`numTriangles` The number of triangles (that is, faces) in the trimesh.

`triangles` A pointer to an array of trimesh triangle data structures, which contain information about the faces in the trimesh.

`numTriangleAttributeTypes`
The number of types of attributes that are associated with each triangle in the trimesh.

`triangleAttributeTypes`
A pointer to a trimesh attributes data structure that contains information about the face attributes of the trimesh. If no attributes are to be assigned to individual faces of the trimesh, this field should be set to `NULL`.

`numEdges` The number of edges in the trimesh. Set this field to 0 if you do not want to specify any edges.

`edges` A pointer to an array of trimesh edge data structures, which contain information about the edges in the trimesh. Set this field to `NULL` if you do not want to specify any edges.

`numEdgeAttributeTypes`
The number of types of attributes that are associated with each edge in the trimesh.

`edgeAttributeTypes` A pointer to a trimesh attributes data structure that contains information about the edge attributes of the trimesh. If no attributes are to be assigned to individual edges of the trimesh, this field should be set to `NULL`.

`numPoints` The number of points in the trimesh.

`points` A pointer to the array of points in the trimesh.

`numVertexAttributeTypes`
The number of types of attributes that are associated with each vertex in the trimesh.

`vertexAttributeTypes`
A pointer to a trimesh attributes data structure that contains information about the vertex attributes of the trimesh. If no attributes are to be assigned to individual vertices of the trimesh, this field should be set to `NULL`.

`bBox` The bounding box of the trimesh.

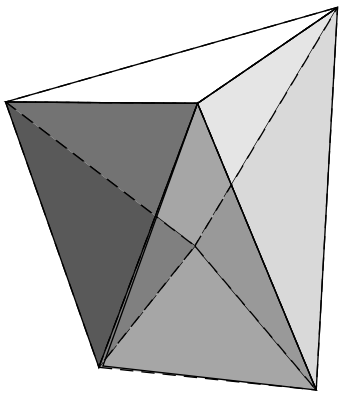
IMPORTANT

You can accelerate trimesh rendering by specifying the trimesh as a strip or fan. A **strip** is a trimesh whose triangles are ordered sequentially (that is, each triangle has one edge in common with the previous neighboring triangle, a second edge in common with the next neighboring triangle, and the remaining edge in common with no other triangle). A **fan** is a strip in which all the triangles share a common vertex. ▲

Polyhedra

A **polyhedron** is a polyhedral primitive, all of whose faces are triangular. The faces of a polyhedron are defined indirectly, using indices into an array of vertices. This indirection makes it easy for faces to share vertices and attribute sets, thereby reducing the memory required to define the polyhedron and reducing the time required to render the polyhedron. Figure 4-23 shows a polyhedron.

Figure 4-23 A polyhedron



Note
It's possible to render non-triangular faces by controlling which edges are drawn. For example, you can make a quadrilateral face by defining two triangular faces with a common edge that is not rendered. ◆

You define an individual face of a polyhedron in part by specifying three indexed vertices. An indexed vertex is a three-dimensional vertex specified by its index into an array of three-dimensional points, together with an attribute set. An indexed vertex is defined using the `TQ3IndexedVertex3D` data type.

```
typedef struct TQ3IndexedVertex3D {
    unsigned long                   pointIndex;
    TQ3AttributeSet                 attributeSet;
} TQ3IndexedVertex3D;
```

Field descriptions

`pointIndex` An index into an array of three-dimensional points. (The array is specified by the `points` field of the `TQ3PolyhedronData` data structure.)

`attributeSet` A set of attributes for the vertex. The value in this field is `NULL` if no vertex attributes are defined.

A polyhedron edge data structure specifies information about an edge of a polyhedron. A polyhedron edge data structure is defined by the `TQ3PolyhedronEdgeData` data structure.

```
typedef struct TQ3PolyhedronEdgeData {
    unsigned long                   pointIndices[2];
    unsigned long                   triangleIndices[2];
    TQ3AttributeSet                 edgeAttributeSet;
} TQ3PolyhedronEdgeData;
```

Field descriptions

`pointIndices` Two indices into an array of three-dimensional points. (The array is specified by the `points` field of the `TQ3PolyhedronData` data structure.) These two points define the endpoints of the edge.

`triangleIndices` Two indices into an array of polyhedron triangle data structures, which contain information about the faces in the polyhedron. (The array is specified by the `triangles` field of

the `TQ3PolyhedronData` data structure.) These two triangles define the two faces that contain the edge. When an edge abuts only one face (that is, when the edge is on a boundary of the polyhedron), you can use the constant `kQ3ArrayIndexNULL` as the face index for the side of the edge that has no face attached to it.

`edgeAttributeSet` A set of attributes for the edge. The value in this field is `NULL` if no edge attributes are defined.

A polyhedron triangle data structure specifies information about a triangular face of a polyhedron. A polyhedron triangle data structure is defined by the `TQ3PolyhedronTriangleData` data type.

```
typedef struct TQ3PolyhedronTriangleData {
    TQ3IndexedVertex3D                  vertices[3];
    TQ3PolyhedronEdge                   edgeFlag;
    TQ3AttributeSet                     triangleAttributeSet;
} TQ3PolyhedronTriangleData;
```

Field descriptions

`vertices` An array specifying the three indexed vertices that define the triangle.

`edgeFlag` A triangle edge flag. The bits in this field indicate which edges of a polyhedral triangle are to be rendered. See "Polyhedron Edge Masks" (page 281) for a list of the available edge flags. Note that a renderer ignores edge flags if an explicit list of polyhedron edges is available.

`triangleAttributeSet` A set of attributes for the triangle. The value in this field is `NULL` if no triangle attributes are defined.

Finally, a polyhedron is defined by the `TQ3PolyhedronData` data type.

```
typedef struct TQ3PolyhedronData {
    unsigned long                       numPoints;
    TQ3Point3D                          *points;
    unsigned long                       numEdges;
    TQ3PolyhedronEdgeData               *edges;
    unsigned long                       numTriangles;
    TQ3PolyhedronTriangleData           *triangles;
    TQ3AttributeSet                     polyhedronAttributeSet;
} TQ3PolyhedronData;
```

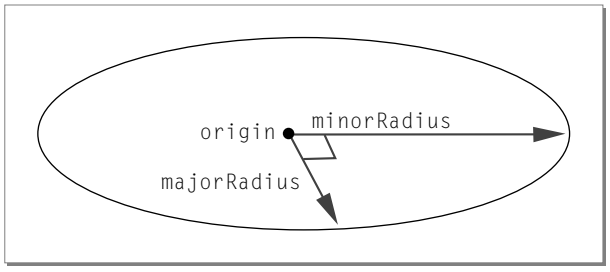
Field descriptions

`numPoints`	The number of points in the polyhedron.
`points`	A pointer to the array of points in the polyhedron.
`numEdges`	The number of edges in the polyhedron. Set this field to 0 if you do not want to specify any edges.
`edges`	A pointer to an array of polyhedron edge data structures, which contain information about the edges in the polyhedron. Set this field to `NULL` if you do not want to specify any edges.
`numTriangles`	The number of triangles (that is, faces) in the polyhedron.
`triangles`	A pointer to an array of polyhedron triangle data structures, which contain information about the faces in the polyhedron.
`polyhedronAttributeSet`	A set of attributes for the entire polyhedron. The value in this field is `NULL` if no polyhedron attributes are defined.

Ellipses

An ellipse is a two-dimensional curve defined by an origin (that is, the center of the ellipse) and two vectors that define the major and minor radii of the ellipse. The origin and the two points at the end of the major and minor radii define the plane in which the ellipse lies. An ellipse is defined by the `TQ3EllipseData` data type. See "Creating and Editing Ellipses," beginning on page 440 for a description of the routines you can use to create and edit ellipses. Figure 4-24 shows an ellipse.

Figure 4-24 An ellipse



```
typedef struct TQ3EllipseData {
    TQ3Point3D                      origin;
    TQ3Vector3D                     majorRadius;
    TQ3Vector3D                     minorRadius;
    float                           uMin, uMax;
    TQ3AttributeSet                 ellipseAttributeSet;
} TQ3EllipseData;
```

Field descriptions

`origin` The origin (that is, the center) of the ellipse.

`majorRadius` The major radius of the ellipse.

`minorRadius` The minor radius of the ellipse.

`uMin` The minimum value in the *u* parametric direction of the ellipse. This value should be greater than or equal to 0.0 and less than or equal to 1.0.

`uMax` The maximum value in the *u* parametric direction of the ellipse. This value should be greater than or equal to 0.0 and less than or equal to 1.0.

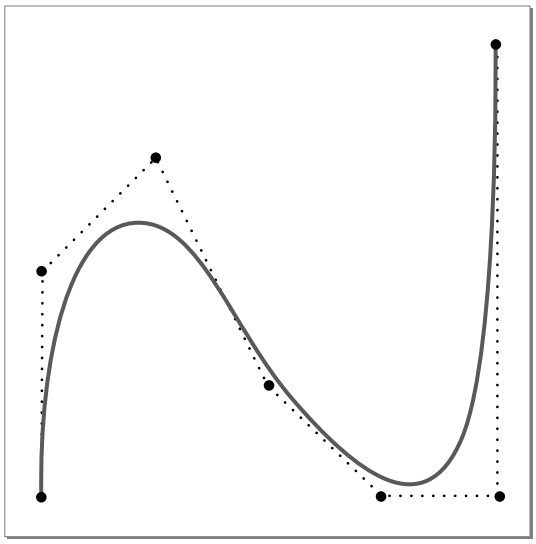
`ellipseAttributeSet` A set of attributes for the ellipse. The value in this field is `NULL` if no ellipse attributes are defined.

NURB Curves

A nonuniform rational B-spline (NURB) curve is a three-dimensional projection of a four-dimensional curve, with an optional set of attributes. A NURB curve is defined by the `TQ3NURBCurveData` data type. See "Creating and Editing NURB Curves," beginning on page 446 for a description of the routines you can use to create and edit NURB curves. Figure 4-25 shows a NURB curve.

```
typedef struct TQ3NURBCurveData {
    unsigned long                   order;
    unsigned long                   numPoints;
    TQ3RationalPoint4D              *controlPoints;
    float                           *knots;
    TQ3AttributeSet                 curveAttributeSet;
} TQ3NURBCurveData;
```

Figure 4-25 A NURB curve



Field descriptions

`order`	The order of the NURB curve. For NURB curves defined by ratios of cubic B-spline polynomials, the order is 4. In general, the order of a NURB curve defined by polynomial equations of degree *n* is *n*+1. The value in this field must be greater than 1.
`numPoints`	The number of control points that define the NURB curve. The value in this field must be greater than or equal to the order of the NURB curve.
`controlPoints`	A pointer to an array of rational four-dimensional control points that define the NURB curve.
`knots`	A pointer to an array of knots that define the NURB curve. The number of knots in a NURB curve is the sum of the values in the `order` and `numPoints` fields. The values in this array must be nondecreasing (but successive values may be equal, up to a multiplicity equivalent to the order of the curve).

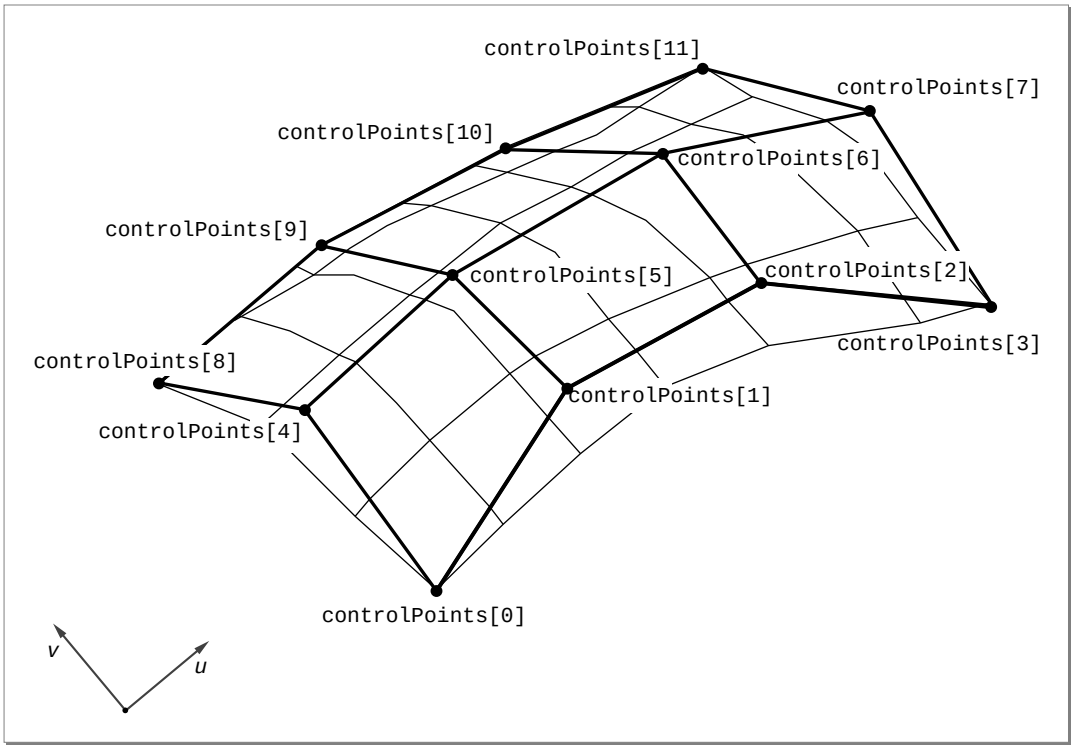
`curveAttributeSet`

A set of attributes for the NURB curve. The value in this field is `NULL` if no NURB curve attributes are defined.

NURB Patches

A NURB patch is a surface defined by ratios of B-spline surfaces, which are three-dimensional analogs of B-spline curves. A NURB patch is defined by the `TQ3NURBPatchData` data type. See "Creating and Editing NURB Patches," beginning on page 451 for a description of the routines you can use to create and edit NURB patches. Figure 4-26 shows a NURB patch.

Figure 4-26 A NURB patch



```
typedef struct TQ3NURBPatchData {
    unsigned long                       uOrder;
    unsigned long                       vOrder;
    unsigned long                       numRows;
    unsigned long                       numColumns;
    TQ3RationalPoint4D                  *controlPoints;
    float                               *uKnots;
    float                               *vKnots;
    unsigned long                       numTrimLoops;
    TQ3NURBPatchTrimLoopData            *trimLoops;
    TQ3AttributeSet                     patchAttributeSet;
} TQ3NURBPatchData;
```

Field descriptions

`uOrder` The order of the NURB patch in the *u* parametric direction. For NURB patches defined by ratios of B-spline polynomials that are cubic in *u*, the order is 4. In general, the order of a NURB patch defined by polynomial equations in which *u* is of degree *n* is *n*+1. The value in this field must be greater than 1.

`vOrder` The order of the NURB patch in the *v* parametric direction. For NURB patches defined by ratios of B-spline polynomials that are cubic in *v*, the order is 4. In general, the order of a NURB patch defined by polynomial equations in which *v* is of degree *n* is *n*+1. The value in this field must be greater than 1.

`numRows` The number of control points in the *u* parametric direction. The value of this field must be greater than 1.

`numColumns` The number of control points in the *v* parametric direction. The value of this field must be greater than 1.

`controlPoints` A pointer to an array of rational four-dimensional control points that define the NURB patch. The first control point in the array is the lower-left corner of the NURB patch. The control points are listed in a rectangular order, first in the direction of increasing *u* and then in the direction of increasing *v*. The number of elements in this array is the product of the values in the `numRows` and `numColumns` fields.

`uKnots` A pointer to an array of knots in the *u* parametric direction that define the NURB patch. The number of *u* knots in a NURB patch is the sum of the values in the `uOrder` and

	`numColumns` fields. The values in this array must be nondecreasing (but successive values may be equal).
`vKnots`	A pointer to an array of knots in the *v* parametric direction that define the NURB patch. The number of *v* knots in a NURB patch is the sum of the values in the `vOrder` and `numRows` fields. The values in this array must be nondecreasing (but successive values may be equal).
`numTrimLoops`	The number of trim loops in the array pointed to by the `trimLoops` field. Currently this field should contain 0.
`trimLoops`	A pointer to an array of trim loop data structures that define the loops used to trim a NURB patch. See below for the structure of the trim loop data structure. Currently this field should contain the value `NULL`.
`patchAttributeSet`	A set of attributes for the NURB patch. The value in this field is `NULL` if no NURB patch attributes are defined.

A **trim loop data structure** is defined by the `TQ3NURBPatchTrimLoopData` data type.

```
typedef struct TQ3NURBPatchTrimLoopData {
    unsigned long                               numTrimCurves;
    TQ3NURBPatchTrimCurveData                   *trimCurves;
} TQ3NURBPatchTrimLoopData;
```

Field descriptions

`numTrimCurves`	The number of trim curves in the array pointed to by the `trimCurves` field.
`trimCurves`	A pointer to an array of trim curve data structures that define the curves used to trim a NURB patch. See below for the structure of the trim curve data structure.

A **trim curve data structure** is defined by the `TQ3NURBPatchTrimCurveData` data type.

```
typedef struct TQ3NURBPatchTrimCurveData {
    unsigned long                               order;
    unsigned long                               numPoints;
    TQ3RationalPoint3D                          *controlPoints;
    float                                       *knots;
} TQ3NURBPatchTrimCurveData;
```

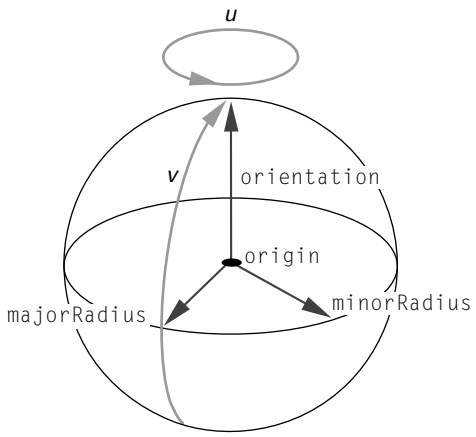
Field descriptions

`order`	The order of the NURB trim curve. In general, the order of a NURB trim curve defined by polynomial equations of degree *n* is *n*+1. The value in this field must be more than 1.
`numPoints`	The number of control points that define the NURB trim curve. The value in this field must be greater than 2.
`controlPoints`	A pointer to an array of three-dimensional rational control points that define the NURB trim curve.
`knots`	A pointer to an array of knots that define the NURB trim curve. The number of knots in a NURB trim curve is the sum of the values in the `order` and `numPoints` fields. The values in this array must be nondecreasing (but successive values may be equal).

Ellipsoids

An ellipsoid is a three-dimensional surface defined by an origin (that is, the center of the ellipsoid) and three mutually perpendicular vectors that define the orientation and the major and minor radii of the ellipsoid. An ellipsoid is defined by the `TQ3EllipsoidData` data type. See "Creating and Editing Ellipsoids," beginning on page 458 for a description of the routines you can use to create and edit ellipsoids. Figure 4-27 shows an ellipsoid.

Figure 4-27 An ellipsoid



```
typedef struct TQ3EllipsoidData {
    TQ3Point3D                      origin;
    TQ3Vector3D                     orientation;
    TQ3Vector3D                     majorRadius;
    TQ3Vector3D                     minorRadius;
    float                           uMin, uMax, vMin, vMax;
    TQ3EndCap                       caps;
    TQ3AttributeSet                 interiorAttributeSet;
    TQ3AttributeSet                 ellipsoidAttributeSet;
} TQ3EllipsoidData;
```

Field descriptions

`origin` The origin (that is, the center) of the ellipsoid.

`orientation` The orientation of the ellipsoid.

`majorRadius` The major radius of the ellipsoid.

`minorRadius` The minor radius of the ellipsoid.

`uMin` The minimum value in the *u* parametric direction of the ellipsoid. This value should be greater than or equal to 0.0 and less than or equal to 1.0.

`uMax` The maximum value in the *u* parametric direction of the ellipsoid. This value should be greater than or equal to 0.0 and less than or equal to 1.0.

`vMin` The minimum value in the *v* parametric direction of the ellipsoid. This value should be greater than or equal to 0.0 and less than or equal to 1.0.

`vMax` The maximum value in the *v* parametric direction of the ellipsoid. This value should be greater than or equal to 0.0 and less than or equal to 1.0.

`caps` The style of caps to be used on the ellipsoid ends. See "End Caps Masks" (page 280) for a description of the masks you can use to specify a value for this field.

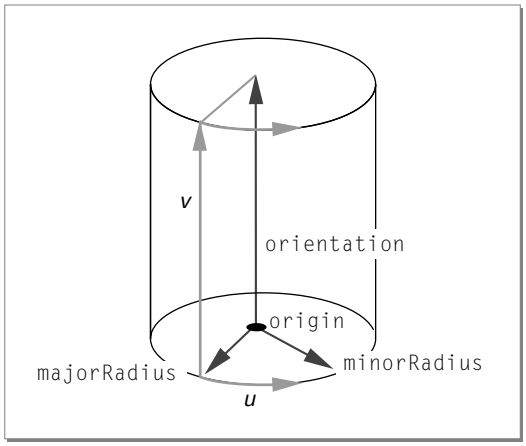
`interiorAttributeSet`
A set of ellipsoid interior attributes.

`ellipsoidAttributeSet`
A set of attributes for the ellipsoid. The value in this field is `NULL` if no ellipsoid attributes are defined.

Cylinders

A cylinder is a three-dimensional object defined by an origin (that is, the center of the base) and three mutually perpendicular vectors that define the orientation and the major and minor radii of the cylinder. A cylinder is defined by the `TQ3CylinderData` data type. See "Creating and Editing Cylinders," beginning on page 465 for a description of the routines you can use to create and edit cylinders. Figure 4-28 shows a cylinder.

Figure 4-28 A cylinder




```
typedef struct TQ3CylinderData {
    TQ3Point3D                  origin;
    TQ3Vector3D                 orientation;
    TQ3Vector3D                 majorRadius;
    TQ3Vector3D                 minorRadius;
    float                       uMin, uMax, vMin, vMax;
    TQ3EndCap                   caps;
    TQ3AttributeSet             interiorAttributeSet;
    TQ3AttributeSet             topAttributeSet;
    TQ3AttributeSet             faceAttributeSet;
    TQ3AttributeSet             bottomAttributeSet;
    TQ3AttributeSet             cylinderAttributeSet;
} TQ3CylinderData;
```

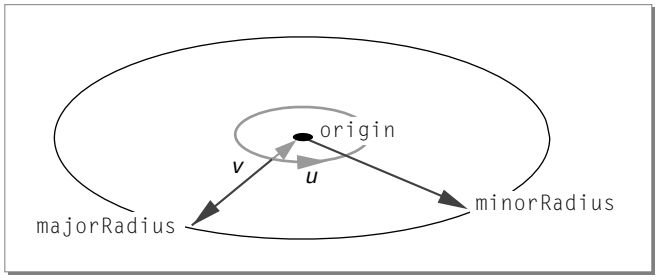
Field descriptions

`origin`	The origin (that is, the center of the base) of the cylinder.
`orientation`	The orientation of the cylinder.
`majorRadius`	The major radius of the cylinder.
`minorRadius`	The minor radius of the cylinder.
`uMin`	The minimum value in the *u* parametric direction of the cylinder. This value should be greater than or equal to 0.0 and less than or equal to 1.0.
`uMax`	The maximum value in the *u* parametric direction of the cylinder. This value should be greater than or equal to 0.0 and less than or equal to 1.0.
`vMin`	The minimum value in the *v* parametric direction of the cylinder. This value should be greater than or equal to 0.0 and less than or equal to 1.0.
`vMax`	The maximum value in the *v* parametric direction of the cylinder. This value should be greater than or equal to 0.0 and less than or equal to 1.0.
`caps`	The style of caps to be used on the cylinder ends. See "End Caps Masks" (page 280) for a description of the masks you can use to specify a value for this field.
`interiorAttributeSet`	A set of cylinder interior attributes.
`topAttributeSet`	A set of cylinder top attributes.
`faceAttributeSet`	A set of cylinder face attributes.
`bottomAttributeSet`	A set of cylinder bottom attributes.
`cylinderAttributeSet`	A set of attributes for the cylinder. The value in this field is `NULL` if no cylinder attributes are defined.

Disks

A disk is a two-dimensional surface defined by an origin (that is, the center of the disk) and two vectors that define the major and minor radii of the disk. A disk is defined by the `TQ3DiskData` data type. See "Creating and Editing Disks," beginning on page 476 for a description of the routines you can use to create and edit disks. Figure 4-29 shows a disk.

Figure 4-29 A disk




```
typedef struct TQ3DiskData {
    TQ3Point3D                  origin;
    TQ3Vector3D                 majorRadius;
    TQ3Vector3D                 minorRadius;
    float                       uMin, uMax, vMin, vMax;
    TQ3AttributeSet             diskAttributeSet;
} TQ3DiskData;
```

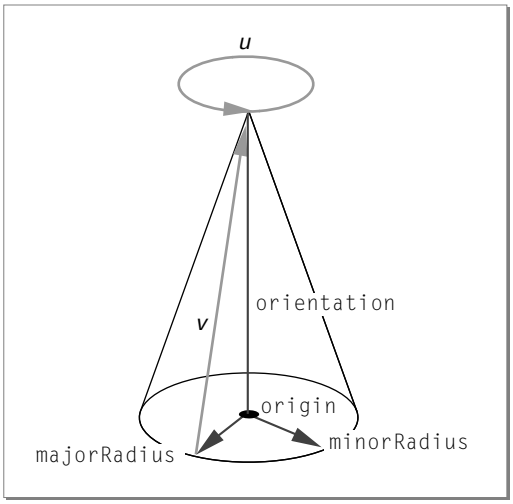
Field descriptions

`origin`	The origin (that is, the center) of the disk.
`majorRadius`	The major radius of the disk.
`minorRadius`	The minor radius of the disk.
`uMin`	The minimum value in the *u* parametric direction of the disk. This value should be greater than or equal to 0.0 and less than or equal to 1.0.
`uMax`	The maximum value in the *u* parametric direction of the disk. This value should be greater than or equal to 0.0 and less than or equal to 1.0.
`vMin`	The minimum value in the *v* parametric direction of the disk. This value should be greater than or equal to 0.0 and less than or equal to 1.0.
`vMax`	The maximum value in the *v* parametric direction of the disk. This value should be greater than or equal to 0.0 and less than or equal to 1.0.
`diskAttributeSet`	A set of attributes for the disk. The value in this field is `NULL` if no disk attributes are defined.

Cones

A cone is a three-dimensional object defined by an origin (that is, the center of the base) and three vectors that define the orientation and the major and minor radii of the cone. A cone is defined by the `TQ3ConeData` data type. See "Creating and Editing Cones," beginning on page 482 for a description of the routines you can use to create and edit cones. Figure 4-30 shows a cone.

Figure 4-30 A cone




```
typedef struct TQ3ConeData {
    TQ3Point3D                  origin;
    TQ3Vector3D                 orientation;
    TQ3Vector3D                 majorRadius;
    TQ3Vector3D                 minorRadius;
    float                       uMin, uMax, vMin, vMax;
    TQ3EndCap                   caps;
    TQ3AttributeSet             interiorAttributeSet;
    TQ3AttributeSet             faceAttributeSet;
    TQ3AttributeSet             bottomAttributeSet;
    TQ3AttributeSet             coneAttributeSet;
} TQ3ConeData;
```

Field descriptions

`origin`	The origin (that is, the center of the base) of the cone.
`orientation`	The orientation of the cone. This vector also specifies the height of the cone.
`majorRadius`	The major radius of the cone.
`minorRadius`	The minor radius of the cone.
`uMin`	The minimum value in the *u* parametric direction of the cone. This value should be greater than or equal to 0.0 and less than or equal to 1.0.
`uMax`	The maximum value in the *u* parametric direction of the cone. This value should be greater than or equal to 0.0 and less than or equal to 1.0.
`vMin`	The minimum value in the *v* parametric direction of the cone. This value should be greater than or equal to 0.0 and less than or equal to 1.0.
`vMax`	The maximum value in the *v* parametric direction of the cone. This value should be greater than or equal to 0.0 and less than or equal to 1.0.
`caps`	The style of cap to be used on the cone base. See "End Caps Masks" (page 280) for a description of the masks you can use to specify a value for this field. For a cone, the value `kQ3EndCapMaskTop` is ignored.
`interiorAttributeSet`	A set of cone interior attributes.
`faceAttributeSet`	A set of cone face attributes.
`bottomAttributeSet`	A set of cone bottom attributes.
`coneAttributeSet`	A set of attributes for the cone. The value in this field is `NULL` if no cone attributes are defined.

Tori

A torus is a three-dimensional object formed by the rotation of an ellipse about an axis in the plane of the ellipse that does not cut the ellipse. The major radius is the distance of the center of the ellipse from that axis. A torus is defined by the `TQ3TorusData` data type. See "Creating and Editing Tori," beginning on page 491 for a description of the routines you can use to create and edit tori. Figure 4-31 shows a torus.

Figure 4-31 A torus

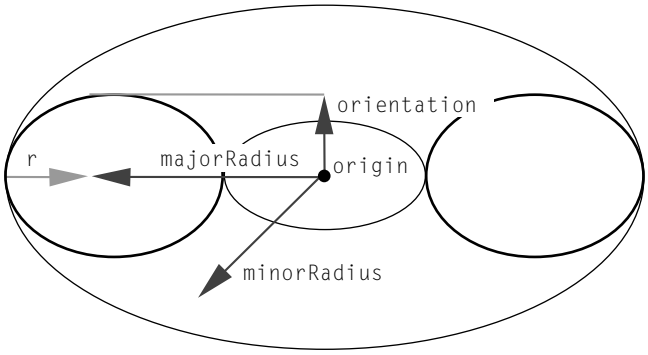
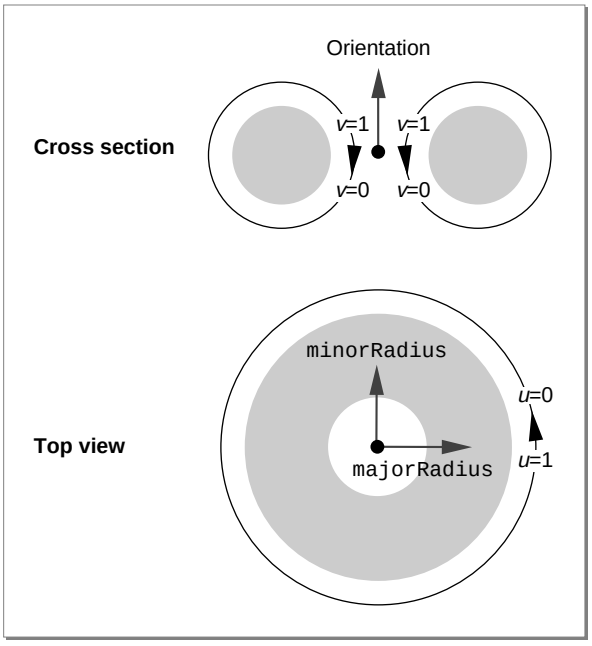


Figure 4-32 shows the standard surface parameterization of a torus.

Figure 4-32 The standard surface parameterization of a torus



```
typedef struct TQ3TorusData {
    TQ3Point3D                  origin;
    TQ3Vector3D                 orientation;
    TQ3Vector3D                 majorRadius;
    TQ3Vector3D                 minorRadius;
    float                       ratio;
    float                       uMin, uMax, vMin, vMax;
    TQ3EndCap                   caps;
    TQ3AttributeSet             interiorAttributeSet;
    TQ3AttributeSet             torusAttributeSet;
} TQ3TorusData;
```

Field descriptions

`origin` The center of the torus. This is the closest point on the axis of rotation to the rotated ellipse.

`orientation` The orientation of the torus. This field specifies the axis of rotation and the half-thickness of the torus. The orientation must be orthogonal to both the major and minor radii.

`majorRadius` The major radius of the torus.

`minorRadius` The minor radius of the torus.

`ratio` The ratio of the major radius of the rotated ellipse to the length of the orientation vector. In Figure 4-31, this is *r*/ length(*orientation*).

`uMin` The minimum value in the *u* parametric direction of the torus. This value should be greater than or equal to 0.0 and less than or equal to 1.0.

`uMax` The maximum value in the *u* parametric direction of the torus. This value should be greater than or equal to 0.0 and less than or equal to 1.0.

`vMin` The minimum value in the *v* parametric direction of the torus. This value should be greater than or equal to 0.0 and less than or equal to 1.0.

`vMax` The maximum value in the *v* parametric direction of the torus. This value should be greater than or equal to 0.0 and less than or equal to 1.0.

`caps` The style of cap to be used on the torus. See "End Caps Masks" (page 280) for a description of the masks you can use to specify a value for this field.

`interiorAttributeSet`
A set of torus interior attributes.

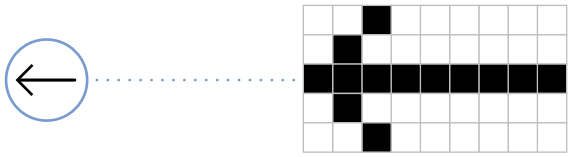
`torusAttributeSet` A set of attributes for the torus. The value in this field is `NULL` if no torus attributes are defined.

Markers

A **marker** is a two-dimensional object typically used to indicate the position of an object (or part of an object) in a window. QuickDraw 3D provides two types of markers, bitmap markers and pixmap markers. A **bitmap marker** is defined by the `TQ3MarkerData` data type, which contains a bitmap and a location, together with an optional set of attributes. A **pixmap marker** is defined by the `TQ3PixmapMarkerData` data type, which contains a pixmap and a location, together with an optional set of attributes.

The bitmap or pixmap specifies the image that is to be drawn on top of a rendered scene at the specified location. The marker is drawn perpendicular to the viewing vector, aligned with the window, with its origin located at the specified location. A marker is always drawn with the same size, no matter which rotations, scalings, or other transformations might be active. Figure 4-33 shows a bitmap marker.

Figure 4-33 A marker



```
typedef struct TQ3MarkerData {
    TQ3Point3D                  location;
    long                        xOffset;
    long                        yOffset;
    TQ3Bitmap                   bitmap;
    TQ3AttributeSet             markerAttributeSet;
} TQ3MarkerData;
```

Field descriptions

`location` The origin of the marker.

`xOffset` The number of pixels, in the horizontal direction, by which to offset the upper-left corner of the marker from the origin specified in the `location` field.

`yOffset` The number of pixels, in the vertical direction, by which to offset the upper-left corner of the marker from the origin specified in the `location` field.

`bitmap` A bitmap. Each bit of this bitmap corresponds to a pixel in the rendered image.

`markerAttributeSet` A set of attributes for the marker. You can use these attributes to specify the color, transparency, or other attributes of the bits in `bitmap` that are set to 1. The value in this field is `NULL` if no marker attributes are defined.

```
typedef struct TQ3PixmapMarkerData {
    TQ3Point3D                  position;
    long                        xOffset;
    long                        yOffset;
    TQ3StoragePixmap            pixmap;
    TQ3AttributeSet             pixmapMarkerAttributeSet;
} TQ3PixmapMarkerData;
```

Field descriptions

`position` The origin of the marker.

`xOffset` The number of pixels, in the horizontal direction, by which to offset the upper-left corner of the marker from the origin specified in the `location` field.

`yOffset` The number of pixels, in the vertical direction, by which to offset the upper-left corner of the marker from the origin specified in the `location` field.

`pixmap` A storage pixmap. Each bit of this pixmap corresponds to a pixel in the rendered image.

`pixmapMarkerAttributeSet`
A set of attributes for the marker. The value in this field is `NULL` if no marker attributes are defined.

Geometric Objects Routines

This section describes the QuickDraw 3D routines that you can use to create and edit the geometric primitive objects.

Managing Geometric Objects

QuickDraw 3D provides a number of general routines for manipulating its primitive geometric objects.

Q3Geometry_GetType

You can use the `Q3Geometry_GetType` function to get the type of a geometric object.

```
TQ3ObjectType Q3Geometry_GetType (TQ3GeometryObject geometry);
```

`geometry` A geometric object.

DESCRIPTION

The `Q3Geometry_GetType` function returns, as its function result, the type of the geometric object specified by the `geometry` parameter. The types of geometric objects currently supported by QuickDraw 3D are defined by these constants:

```
kQ3GeometryTypeBox
kQ3GeometryTypeCone
kQ3GeometryTypeCylinder
kQ3GeometryTypeDisk
kQ3GeometryTypeEllipse
kQ3GeometryTypeEllipsoid
kQ3GeometryTypeGeneralPolygon
kQ3GeometryTypeLine
kQ3GeometryTypeMarker
kQ3GeometryTypeMesh
kQ3GeometryTypeNURBCurve
kQ3GeometryTypeNURBPatch
kQ3GeometryTypePixmapMarker
```

```
kQ3GeometryTypePoint
kQ3GeometryTypePolygon
kQ3GeometryTypePolyhedron
kQ3GeometryTypePolyLine
kQ3GeometryTypeTorus
kQ3GeometryTypeTriangle
kQ3GeometryTypeTriGrid
kQ3GeometryTypeTriMesh
```

If the specified geometric object is invalid or is not one of these types, `Q3Geometry_GetType` returns the value `kQ3ObjectTypeInvalid`.

Q3Geometry_GetAttributeSet

You can use the `Q3Geometry_GetAttributeSet` function to get the attribute set associated with an entire geometric object.

```
TQ3Status Q3Geometry_GetAttributeSet (
                    TQ3GeometryObject geometry,
                    TQ3AttributeSet *attributeSet);
```

`geometry` A geometric object.

`attributeSet` On exit, the set of attributes of the specified geometric object.

DESCRIPTION

The `Q3Geometry_GetAttributeSet` function returns, in the `attributeSet` parameter, the set of attributes currently associated with the geometric object specified by the `geometry` parameter. The reference count of the set is incremented.

Q3Geometry_SetAttributeSet

You can use the `Q3Geometry_SetAttributeSet` function to set the attribute set associated with a geometric object.

```
TQ3Status Q3Geometry_SetAttributeSet (
                    TQ3GeometryObject geometry,
                    TQ3AttributeSet attributeSet);
```

`geometry` A geometric object.

`attributeSet` A set of attributes.

DESCRIPTION

The `Q3Geometry_SetAttributeSet` function sets the attribute set of the geometric object specified by the `geometry` parameter to the set specified by the `attributeSet` parameter.

Q3Geometry_Submit

You can use the `Q3Geometry_Submit` function to submit a retained geometric object for drawing, picking, bounding, or writing.

```
TQ3Status Q3Geometry_Submit (
                    TQ3GeometryObject geometry,
                    TQ3ViewObject view);
```

`geometry` A geometric object.

`view` A view.

DESCRIPTION

The `Q3Geometry_Submit` function submits the geometric object specified by the `geometry` parameter for drawing, picking, bounding, or writing according to the view characteristics specified in the `view` parameter. The geometric object must have been created by a call that creates a retained object (for example, `Q3Point_New`).

SPECIAL CONSIDERATIONS

You should call this function only in a submitting loop.

Creating and Editing Points

QuickDraw 3D provides routines that you can use to create and manipulate points. See "Point Objects" (page 295) for the definition of the point object.

Q3Point_New

You can use the `Q3Point_New` function to create a new point.

```
TQ3GeometryObject Q3Point_New (const TQ3PointData *pointData);
```

`pointData` A pointer to a `TQ3PointData` structure.

DESCRIPTION

The `Q3Point_New` function returns, as its function result, a new point object having the location and attributes passed in the fields of the `TQ3PointData` structure pointed to by the `pointData` parameter. If a new point object could not be created, `Q3Point_New` returns the value `NULL`.

Q3Point_Submit

You can use the `Q3Point_Submit` function to submit an immediate point for drawing, picking, bounding, or writing.

```
TQ3Status Q3Point_Submit (
                    const TQ3PointData *pointData,
                    TQ3ViewObject view);
```

`pointData` A pointer to a `TQ3PointData` structure.

`view` A view.

DESCRIPTION

The `Q3Point_Submit` function submits for drawing, picking, bounding, or writing the immediate point whose location and attribute set are passed in the fields of the `TQ3PointData` structure pointed to by the `pointData` parameter. The point is drawn, picked, bounded, or written according to the view characteristics specified in the `view` parameter.

SPECIAL CONSIDERATIONS

You should call this function only in a submitting loop.

Q3Point_GetData

You can use the `Q3Point_GetData` function to get the data that defines a point object and its attributes.

```
TQ3Status Q3Point_GetData (
                    TQ3GeometryObject point,
                    TQ3PointData *pointData);
```

`point` A point.

`pointData` On exit, a pointer to a `TQ3PointData` structure that contains information about the point specified by the `point` parameter.

DESCRIPTION

The `Q3Point_GetData` function returns, through the `pointData` parameter, information about the position and attribute set of the point specified by the `point` parameter. QuickDraw 3D allocates memory for the `TQ3PointData` structure internally; you must call `Q3Point_EmptyData` to dispose of that memory.

Q3Point_SetData

You can use the `Q3Point_SetData` function to set the data that defines a point object and its attributes.

```
TQ3Status Q3Point_SetData (
                    TQ3GeometryObject point,
                    const TQ3PointData *pointData);
```

`point` A point.

`pointData` A pointer to a `TQ3PointData` structure.

DESCRIPTION

The `Q3Point_SetData` function sets the data associated with the point specified by the `point` parameter to the data specified by the `pointData` parameter.

Q3Point_EmptyData

You can use the `Q3Point_EmptyData` function to release the memory occupied by the data structure returned by a previous call to `Q3Point_GetData`.

```
TQ3Status Q3Point_EmptyData (TQ3PointData *pointData);
```

`pointData` A pointer to a `TQ3PointData` structure.

DESCRIPTION

The `Q3Point_EmptyData` function releases the memory occupied by the `TQ3PointData` structure pointed to by the `pointData` parameter; that memory was allocated by a previous call to `Q3Point_GetData`.

Q3Point_GetPosition

You can use the `Q3Point_GetPosition` function to get the position of a point.

```
TQ3Status Q3Point_GetPosition (
                    TQ3GeometryObject point,
                    TQ3Point3D *position);
```

`point` A point.

`position` On exit, the position of the specified point.

DESCRIPTION

The `Q3Point_GetPosition` function returns, in the `position` parameter, the position of the point specified by the `point` parameter.

Q3Point_SetPosition

You can use the `Q3Point_SetPosition` function to set the position of a point.

```
TQ3Status Q3Point_SetPosition (
                    TQ3GeometryObject point,
                    const TQ3Point3D *position);
```

`point` A point.

`position` The desired position of the specified point.

DESCRIPTION

The `Q3Point_SetPosition` function sets the position of the point specified by the `point` parameter to that specified in the `position` parameter.

Creating and Editing Lines

QuickDraw 3D provides routines that you can use to create and manipulate lines. See “Lines” (page 295) for the definition of a line.

Q3Line_New

You can use the `Q3Line_New` function to create a new line.

```
TQ3GeometryObject Q3Line_New (const TQ3LineData *lineData);
```

`lineData` A pointer to a `TQ3LineData` structure.

DESCRIPTION

The `Q3Line_New` function returns, as its function result, a new line having the endpoints and attributes specified by the `lineData` parameter. If a new line could not be created, `Q3Line_New` returns the value `NULL`.

Q3Line_Submit

You can use the `Q3Line_Submit` function to submit an immediate line for drawing, picking, bounding, or writing.

```
TQ3Status Q3Line_Submit (
                    const TQ3LineData *lineData,
                    TQ3ViewObject view);
```

`lineData` A pointer to a `TQ3LineData` structure.

`view` A view.

DESCRIPTION

The `Q3Line_Submit` function submits for drawing, picking, bounding, or writing the immediate line whose location and attribute set are specified by the `lineData` parameter. The line is drawn, picked, bounded, or written according to the view characteristics specified in the `view` parameter.

SPECIAL CONSIDERATIONS

You should call this function only in a submitting loop.

Q3Line_GetData

You can use the `Q3Line_GetData` function to get the data that defines a line and its attributes.

```
TQ3Status Q3Line_GetData (
                    TQ3GeometryObject line,
                    TQ3LineData *lineData);
```

`line` A line.

`lineData` On exit, a pointer to a `TQ3LineData` structure that contains information about the line specified by the `line` parameter.

DESCRIPTION

The `Q3Line_GetData` function returns, through the `lineData` parameter, information about the line specified by the `line` parameter. QuickDraw 3D allocates memory for the `TQ3LineData` structure internally; you must call `Q3Line_EmptyData` to dispose of that memory.

Q3Line_SetData

You can use the `Q3Line_SetData` function to set the data that defines a line and its attributes.

```
TQ3Status Q3Line_SetData (
                    TQ3GeometryObject line,
                    const TQ3LineData *lineData);
```

`line` A line.

`lineData` A pointer to a `TQ3LineData` structure.

DESCRIPTION

The `Q3Line_SetData` function sets the data associated with the line specified by the `line` parameter to the data specified by the `lineData` parameter.

Q3Line_GetVertexPosition

You can use the `Q3Line_GetVertexPosition` function to get the position of a vertex of a line.

```
TQ3Status Q3Line_GetVertexPosition (
                    TQ3GeometryObject line,
                    unsigned long index,
                    TQ3Point3D *position);
```

`line` A line.

`index` An index into the `vertices` array of the specified line. This parameter should have the value 0 or 1.

`position` On exit, the position of the specified vertex.

DESCRIPTION

The `Q3Line_GetVertexPosition` function returns, in the `position` parameter, the position of the vertex having the index specified by the `index` parameter in the `vertices` array of the line specified by the `line` parameter.

Q3Line_SetVertexPosition

You can use the `Q3Line_SetVertexPosition` function to set the position of a vertex of a line.

```
TQ3Status Q3Line_SetVertexPosition (
                    TQ3GeometryObject line,
                    unsigned long index,
                    const TQ3Point3D *position);
```

`line` A line.

`index` An index into the `vertices` array of the specified line. This parameter should have the value 0 or 1.

`position` The desired position of the specified vertex.

DESCRIPTION

The `Q3Line_SetVertexPosition` function sets the position of the vertex having the index specified by the `index` parameter in the `vertices` array of the line specified by the `line` parameter to that specified in the `position` parameter.

Q3Line_GetVertexAttributeSet

You can use the `Q3Line_GetVertexAttributeSet` function to get the attribute set of a vertex of a line.

```
TQ3Status Q3Line_GetVertexAttributeSet (
                    TQ3GeometryObject line,
                    unsigned long index,
                    TQ3AttributeSet *attributeSet);
```

`line` A line.

`index` An index into the `vertices` array of the specified line. This parameter should have the value 0 or 1.

`attributeSet`
On exit, a pointer to a vertex attribute set for the specified vertex.

DESCRIPTION

The `Q3Line_GetVertexAttributeSet` function returns, in the `attributeSet` parameter, the set of attributes for the vertex having the index specified by the `index` parameter in the `vertices` array of the line specified by the `line` parameter. The reference count of the set is incremented.

Q3Line_SetVertexAttributeSet

You can use the `Q3Line_SetVertexAttributeSet` function to set the attribute set of a vertex of a line.

```
TQ3Status Q3Line_SetVertexAttributeSet (
                    TQ3GeometryObject line,
                    unsigned long index,
                    TQ3AttributeSet attributeSet);
```

`line` A line.

`index` An index into the `vertices` array of the specified line. This parameter should have the value 0 or 1.

`attributeSet`
The desired set of attributes for the specified vertex.

DESCRIPTION

The `Q3Line_SetVertexAttributeSet` function sets the attribute set of a vertex to the set specified in the `attributeSet` parameter. The vertex is identified by the specified index into the `vertices` array of the line specified by the `line` parameter.

Q3Line_EmptyData

You can use the `Q3Line_EmptyData` function to release the memory occupied by the data structure returned by a previous call to `Q3Line_GetData`.

```
TQ3Status Q3Line_EmptyData (TQ3LineData *lineData);
```

`lineData` A pointer to a `TQ3LineData` structure.

DESCRIPTION

The `Q3Line_EmptyData` function releases the memory occupied by the `TQ3LineData` structure pointed to by the `lineData` parameter; that memory was allocated by a previous call to `Q3Line_GetData`.

Creating and Editing Polylines

QuickDraw 3D provides routines that you can use to create and manipulate polylines. See “Polylines” (page 296) for the definition of a polyline.

Q3PolyLine_New

You can use the `Q3PolyLine_New` function to create a new polyline.

```
TQ3GeometryObject Q3PolyLine_New (
                    const TQ3PolyLineData *polyLineData);
```

`polyLineData`
A pointer to a `TQ3PolyLineData` structure.

DESCRIPTION

The `Q3PolyLine_New` function returns, as its function result, a new polyline having the vertices and attributes specified by the `polyLineData` parameter. If a new polyline could not be created, `Q3PolyLine_New` returns the value `NULL`.

Q3PolyLine_Submit

You can use the `Q3PolyLine_Submit` function to submit an immediate polyline for drawing, picking, bounding, or writing.

```
TQ3Status Q3PolyLine_Submit (
                    const TQ3PolyLineData *polyLineData,
                    TQ3ViewObject view);
```

`polyLineData`
A pointer to a `TQ3PolyLineData` structure.

`view` A view.

DESCRIPTION

The `Q3PolyLine_Submit` function submits for drawing, picking, bounding, or writing the immediate polyline whose shape and attribute sets are specified by the `polyLineData` parameter. The polyline is drawn, picked, bounded, or written according to the view characteristics specified in the `view` parameter.

SPECIAL CONSIDERATIONS

You should call this function only in a submitting loop.

Q3PolyLine_GetData

You can use the `Q3PolyLine_GetData` function to get the data that defines a polyline and its attributes.

```
TQ3Status Q3PolyLine_GetData (
                    TQ3GeometryObject polyLine,
                    TQ3PolyLineData *polyLineData);
```

`polyLine` A polyline.

`polyLineData` On exit, a pointer to a `TQ3PolyLineData` structure that contains information about the polyline specified by the `polyLine` parameter.

DESCRIPTION

The `Q3PolyLine_GetData` function returns, through the `polyLineData` parameter, information about the polyline specified by the `polyLine` parameter. QuickDraw 3D allocates memory for the `TQ3PolyLineData` structure internally; you must call `Q3PolyLine_EmptyData` to dispose of that memory.

Q3PolyLine_SetData

You can use the `Q3PolyLine_SetData` function to set the data that defines a polyline and its attributes.

```
TQ3Status Q3PolyLine_SetData (
                    TQ3GeometryObject polyLine,
                    const TQ3PolyLineData *polyLineData);
```

`polyLine` A polyline.

`polyLineData` A pointer to a `TQ3PolyLineData` structure.

DESCRIPTION

The `Q3PolyLine_SetData` function sets the data associated with the polyline specified by the `polyLine` parameter to the data specified by the `polyLineData` parameter.

Q3PolyLine_EmptyData

You can use the `Q3PolyLine_EmptyData` function to release the memory occupied by the data structure returned by a previous call to `Q3PolyLine_GetData`.

```
TQ3Status Q3PolyLine_EmptyData (TQ3PolyLineData *polyLineData);
```

`polyLineData` A pointer to a `TQ3PolyLineData` structure.

DESCRIPTION

The `Q3PolyLine_EmptyData` function releases the memory occupied by the `TQ3PolyLineData` structure pointed to by the `polyLineData` parameter; that memory was allocated by a previous call to `Q3PolyLine_GetData`.

Q3PolyLine_GetVertexPosition

You can use the `Q3PolyLine_GetVertexPosition` function to get the position of a vertex of a polyline.

```
TQ3Status Q3PolyLine_GetVertexPosition (
                    TQ3GeometryObject polyLine,
                    unsigned long index,
                    TQ3Point3D *position);
```

`polyLine` A polyline.

`index` An index into the `vertices` array of the specified polyline. This index should be greater than or equal to 0 and less than the number of vertices in the array.

`position` On exit, the position of the specified vertex.

DESCRIPTION

The `Q3PolyLine_GetVertexPosition` function returns, in the `position` parameter, the position of the vertex having the index specified by the `index` parameter in the `vertices` array of the polyline specified by the `polyLine` parameter.

Q3PolyLine_SetVertexPosition

You can use the `Q3PolyLine_SetVertexPosition` function to set the position of a vertex of a polyline.

```
TQ3Status Q3PolyLine_SetVertexPosition (
                    TQ3GeometryObject polyLine,
                    unsigned long index,
                    const TQ3Point3D *position);
```

`polyLine`	A polyline.
`index`	An index into the `vertices` array of the specified polyline.
`position`	The desired position of the specified vertex.

DESCRIPTION

The `Q3PolyLine_SetVertexPosition` function sets the position of a vertex to that specified in the `position` parameter. The vertex has the index specified by the `index` parameter into the `vertices` array of the polyline specified by the `polyLine` parameter.

Q3PolyLine_GetVertexAttributeSet

You can use the `Q3PolyLine_GetVertexAttributeSet` function to get the attribute set of a vertex of a polyline.

```
TQ3Status Q3PolyLine_GetVertexAttributeSet (
                    TQ3GeometryObject polyLine,
                    unsigned long index,
                    TQ3AttributeSet *attributeSet);
```

`polyLine`	A polyline.
`index`	An index into the `vertices` array of the specified polyline.
`attributeSet`	On exit, a pointer to a vertex attribute set for the specified vertex.

DESCRIPTION

The `Q3PolyLine_GetVertexAttributeSet` function returns, in the `attributeSet` parameter, the set of attributes for the vertex having the index specified by the `index` parameter in the `vertices` array of the polyline specified by the `polyLine` parameter. The reference count of the set is incremented.

Q3PolyLine_SetVertexAttributeSet

You can use the `Q3PolyLine_SetVertexAttributeSet` function to set the attribute set of a vertex of a polyline.

```
TQ3Status Q3PolyLine_SetVertexAttributeSet (
                    TQ3GeometryObject polyLine,
                    unsigned long index,
                    TQ3AttributeSet attributeSet);
```

`polyLine`	A polyline.
`index`	An index into the `vertices` array of the specified polyline.
`attributeSet`	The desired set of attributes for the specified vertex.

DESCRIPTION

The `Q3PolyLine_SetVertexAttributeSet` function sets the attribute set of the vertex having the index specified by the `index` parameter in the `vertices` array of the polyline specified by the `polyLine` parameter to the set specified in the `attributeSet` parameter.

Q3PolyLine_GetSegmentAttributeSet

You can use the `Q3PolyLine_GetSegmentAttributeSet` function to get the attribute set of a segment of a polyline.

```
TQ3Status Q3PolyLine_GetSegmentAttributeSet (
                    TQ3GeometryObject polyLine,
                    unsigned long index,
                    TQ3AttributeSet *attributeSet);
```

`polyLine` A polyline.

`index` An index into the `vertices` array of the specified polyline.

`attributeSet`
On exit, a pointer to an attribute set for the specified segment.

DESCRIPTION

The `Q3PolyLine_GetSegmentAttributeSet` function returns, in the `attributeSet` parameter, the set of attributes for a segment of a polyline. The segment is defined by the two vertices having indices `index` and `index`+1 in the `vertices` array of the polyline specified by the `polyLine` parameter. The reference count of the set is incremented.

Q3PolyLine_SetSegmentAttributeSet

You can use the `Q3PolyLine_SetSegmentAttributeSet` function to set the attribute set of a segment of a polyline.

```
TQ3Status Q3PolyLine_SetSegmentAttributeSet (
                    TQ3GeometryObject polyLine,
                    unsigned long index,
                    TQ3AttributeSet attributeSet);
```

`polyLine` A polyline.

`index` An index into the `vertices` array of the specified polyline.

`attributeSet` The desired set of attributes for the specified segment.

DESCRIPTION

The `Q3PolyLine_SetSegmentAttributeSet` function sets the attribute set of a segment of a polyline to the set specified in the `attributeSet` parameter. The segment is defined by the two vertices having indices `index` and `index`+1 in the `vertices` array of the polyline specified by the `polyLine` parameter.

Creating and Editing Triangles

QuickDraw 3D provides routines that you can use to create and manipulate triangles. See "Triangles" (page 297) for the definition of a triangle.

Q3Triangle_New

You can use the `Q3Triangle_New` function to create a new triangle.

```
TQ3GeometryObject Q3Triangle_New (
                    const TQ3TriangleData *triangleData);
```

`triangleData` A pointer to a `TQ3TriangleData` structure.

DESCRIPTION

The `Q3Triangle_New` function returns, as its function result, a new triangle having the vertices and attributes specified by the `triangleData` parameter. If a new triangle could not be created, `Q3Triangle_New` returns the value `NULL`.

Q3Triangle_Submit

You can use the `Q3Triangle_Submit` function to submit an immediate triangle for drawing, picking, bounding, or writing.

```
TQ3Status Q3Triangle_Submit (
                    const TQ3TriangleData *triangleData,
                    TQ3ViewObject view);
```

`triangleData` A pointer to a `TQ3TriangleData` structure.

`view` A view.

DESCRIPTION

The `Q3Triangle_Submit` function submits for drawing, picking, bounding, or writing the immediate triangle whose shape and attribute set are specified by the `triangleData` parameter. The triangle is drawn, picked, bounded, or written according to the view characteristics specified in the `view` parameter.

SPECIAL CONSIDERATIONS

You should call this function only in a submitting loop.

Q3Triangle_GetData

You can use the `Q3Triangle_GetData` function to get the data that defines a triangle and its attributes.

```
TQ3Status Q3Triangle_GetData (
                    TQ3GeometryObject triangle,
                    TQ3TriangleData *triangleData);
```

`triangle` A triangle.

`triangleData` On exit, a pointer to a `TQ3TriangleData` structure that contains information about the triangle specified by the `triangle` parameter.

DESCRIPTION

The `Q3Triangle_GetData` function returns, through the `triangleData` parameter, information about the triangle specified by the `triangle` parameter. QuickDraw 3D allocates memory for the `TQ3TriangleData` structure internally; you must call `Q3Triangle_EmptyData` to dispose of that memory.

Q3Triangle_SetData

You can use the `Q3Triangle_SetData` function to set the data that defines a triangle and its attributes.

```
TQ3Status Q3Triangle_SetData (
                    TQ3GeometryObject triangle,
                    const TQ3TriangleData *triangleData);
```

`triangle` A triangle.

`triangleData` A pointer to a `TQ3TriangleData` structure.

DESCRIPTION

The `Q3Triangle_SetData` function sets the data associated with the triangle specified by the `triangle` parameter to the data specified by the `triangleData` parameter.

Q3Triangle_EmptyData

You can use the `Q3Triangle_EmptyData` function to release the memory occupied by the data structure returned by a previous call to `Q3Triangle_GetData`.

```
TQ3Status Q3Triangle_EmptyData (TQ3TriangleData *triangleData);
```

`triangleData` A pointer to a `TQ3TriangleData` structure.

DESCRIPTION

The `Q3Triangle_EmptyData` function releases the memory occupied by the `TQ3TriangleData` structure pointed to by the `triangleData` parameter; that memory was allocated by a previous call to `Q3Triangle_GetData`.

Q3Triangle_GetVertexPosition

You can use the `Q3Triangle_GetVertexPosition` function to get the position of a vertex of a triangle.

```
TQ3Status Q3Triangle_GetVertexPosition (
                    TQ3GeometryObject triangle,
                    unsigned long index,
                    TQ3Point3D *point);
```

`triangle` A triangle.

`index` An index into the `vertices` array of the specified triangle. This parameter should have the value 0, 1, or 2.

`point` On exit, the position of the specified vertex.

DESCRIPTION

The `Q3Triangle_GetVertexPosition` function returns, in the `point` parameter, the position of the vertex having the index specified by the `index` parameter in the `vertices` array of the triangle specified by the `triangle` parameter.

Q3Triangle_SetVertexPosition

You can use the `Q3Triangle_SetVertexPosition` function to set the position of a vertex of a triangle.

```
TQ3Status Q3Triangle_SetVertexPosition (
                    TQ3GeometryObject triangle,
                    unsigned long index,
                    const TQ3Point3D *point);
```

`triangle` A triangle.

`index` An index into the `vertices` array of the specified triangle. This parameter should have the value 0, 1, or 2.

`point` The desired position of the specified vertex.

DESCRIPTION

The `Q3Triangle_SetVertexPosition` function sets the position of the vertex having the index specified by the `index` parameter in the `vertices` array of the triangle specified by the `triangle` parameter to that specified in the `point` parameter.

Q3Triangle_GetVertexAttributeSet

You can use the `Q3Triangle_GetVertexAttributeSet` function to get the attribute set of a vertex of a triangle.

```
TQ3Status Q3Triangle_GetVertexAttributeSet (
                    TQ3GeometryObject triangle,
                    unsigned long index,
                    TQ3AttributeSet *attributeSet);
```

`triangle` A triangle.

`index` An index into the `vertices` array of the specified triangle. This parameter should have the value 0, 1, or 2.

`attributeSet`
On exit, a pointer to a vertex attribute set for the specified vertex.

DESCRIPTION

The `Q3Triangle_GetVertexAttributeSet` function returns, in the `attributeSet` parameter, the set of attributes for the vertex having the index specified by the `index` parameter in the `vertices` array of the triangle specified by the `triangle` parameter. The reference count of the set is incremented.

Q3Triangle_SetVertexAttributeSet

You can use the `Q3Triangle_SetVertexAttributeSet` function to set the attribute set of a vertex of a triangle.

```
TQ3Status Q3Triangle_SetVertexAttributeSet (
                    TQ3GeometryObject triangle,
                    unsigned long index,
                    TQ3AttributeSet attributeSet);
```

`triangle` A triangle.

`index` An index into the `vertices` array of the specified triangle. This parameter should have the value 0, 1, or 2.

`attributeSet`
The desired set of attributes for the specified vertex.

DESCRIPTION

The `Q3Triangle_SetVertexAttributeSet` function sets the attribute set of the vertex having the index specified by the `index` parameter in the `vertices` array of the triangle specified by the `triangle` parameter to the set specified in the `attributeSet` parameter.

Creating and Editing Simple Polygons

QuickDraw 3D provides routines that you can use to create and manipulate simple polygons. See “Simple Polygons” (page 298) for the definition of a simple polygon.

Q3Polygon_New

You can use the `Q3Polygon_New` function to create a new simple polygon.

```
TQ3GeometryObject Q3Polygon_New (
                    const TQ3PolygonData *polygonData);
```

`polygonData` A pointer to a `TQ3PolygonData` structure.

DESCRIPTION

The `Q3Polygon_New` function returns, as its function result, a new simple polygon having the vertices and attributes specified by the `polygonData` parameter. If a new simple polygon could not be created, `Q3Polygon_New` returns the value `NULL`.

Q3Polygon_Submit

You can use the `Q3Polygon_Submit` function to submit an immediate simple polygon for drawing, picking, bounding, or writing.

```
TQ3Status Q3Polygon_Submit (
                    const TQ3PolygonData *polygonData,
                    TQ3ViewObject view);
```

`polygonData` A pointer to a `TQ3PolygonData` structure.

`view` A view.

DESCRIPTION

The `Q3Polygon_Submit` function submits for drawing, picking, bounding, or writing the immediate simple polygon whose shape and attribute set are specified by the `polygonData` parameter. The simple polygon is drawn, picked, bounded, or written according to the view characteristics specified in the `view` parameter.

SPECIAL CONSIDERATIONS

You should call this function only in a submitting loop.

Q3Polygon_GetData

You can use the `Q3Polygon_GetData` function to get the data that defines a simple polygon and its attributes.

```
TQ3Status Q3Polygon_GetData (
                    TQ3GeometryObject polygon,
                    TQ3PolygonData *polygonData);
```

`polygon` A simple polygon.

`polygonData` On exit, a pointer to a `TQ3PolygonData` structure that contains information about the simple polygon specified by the `polygon` parameter.

DESCRIPTION

The `Q3Polygon_GetData` function returns, through the `polygonData` parameter, information about the simple polygon specified by the `polygon` parameter. QuickDraw 3D allocates memory for the `TQ3PolygonData` structure internally; you must call `Q3Polygon_EmptyData` to dispose of that memory.

Q3Polygon_SetData

You can use the `Q3Polygon_SetData` function to set the data that defines a simple polygon and its attributes.

```
TQ3Status Q3Polygon_SetData (
                    TQ3GeometryObject polygon,
                    const TQ3PolygonData *polygonData);
```

`polygon` A simple polygon.

`polygonData` A pointer to a `TQ3PolygonData` structure.

DESCRIPTION

The `Q3Polygon_SetData` function sets the data associated with the simple polygon specified by the `polygon` parameter to the data specified by the `polygonData` parameter.

Q3Polygon_EmptyData

You can use the `Q3Polygon_EmptyData` function to release the memory occupied by the data structure returned by a previous call to `Q3Polygon_GetData`.

```
TQ3Status Q3Polygon_EmptyData (TQ3PolygonData *polygonData);
```

`polygonData` A pointer to a `TQ3PolygonData` structure.

DESCRIPTION

The `Q3Polygon_EmptyData` function releases the memory occupied by the `TQ3PolygonData` structure pointed to by the `polygonData` parameter; that memory was allocated by a previous call to `Q3Polygon_GetData`.

Q3Polygon_GetVertexPosition

You can use the `Q3Polygon_GetVertexPosition` function to get the position of a vertex of a simple polygon.

```
TQ3Status Q3Polygon_GetVertexPosition (
                    TQ3GeometryObject polygon,
                    unsigned long index,
                    TQ3Point3D *point);
```

`polygon` A simple polygon.

`index` An index into the `vertices` array of the specified simple polygon.

`point` On exit, the position of the specified vertex.

DESCRIPTION

The `Q3Polygon_GetVertexPosition` function returns, in the `point` parameter, the position of the vertex having the index specified by the `index` parameter in the `vertices` array of the simple polygon specified by the `polygon` parameter.

Q3Polygon_SetVertexPosition

You can use the `Q3Polygon_SetVertexPosition` function to set the position of a vertex of a simple polygon.

```
TQ3Status Q3Polygon_SetVertexPosition (
                    TQ3GeometryObject polygon,
                    unsigned long index,
                    const TQ3Point3D *point);
```

`polygon`	A simple polygon.
`index`	An index into the `vertices` array of the specified simple polygon.
`point`	The desired position of the specified vertex.

DESCRIPTION

The `Q3Polygon_SetVertexPosition` function sets the position of the vertex having the index specified by the `index` parameter in the `vertices` array of the simple polygon specified by the `polygon` parameter to that specified in the `point` parameter.

Q3Polygon_GetVertexAttributeSet

You can use the `Q3Polygon_GetVertexAttributeSet` function to get the attribute set of a vertex of a simple polygon.

```
TQ3Status Q3Polygon_GetVertexAttributeSet (
                    TQ3GeometryObject polygon,
                    unsigned long index,
                    TQ3AttributeSet *attributeSet);
```

`polygon` A simple polygon.

`index` An index into the `vertices` array of the specified simple polygon.

`attributeSet`
On exit, a pointer to a vertex attribute set for the specified vertex.

DESCRIPTION

The `Q3Polygon_GetVertexAttributeSet` function returns, in the `attributeSet` parameter, the set of attributes for the vertex having the index specified by the `index` parameter in the `vertices` array of the simple polygon specified by the `polygon` parameter. The reference count of the set is incremented.

Q3Polygon_SetVertexAttributeSet

You can use the `Q3Polygon_SetVertexAttributeSet` function to set the attribute set of a vertex of a simple polygon.

```
TQ3Status Q3Polygon_SetVertexAttributeSet (
                    TQ3GeometryObject polygon,
                    unsigned long index,
                    TQ3AttributeSet attributeSet);
```

`polygon` A simple polygon.

`index` An index into the `vertices` array of the specified simple polygon.

`attributeSet`
The desired set of attributes for the specified vertex.

DESCRIPTION

The `Q3Polygon_SetVertexAttributeSet` function sets the attribute set of the vertex having the index specified by the `index` parameter in the `vertices` array of the simple polygon specified by the `polygon` parameter to the set specified in the `attributeSet` parameter.

Creating and Editing General Polygons

QuickDraw 3D provides routines that you can use to create and manipulate general polygons. See "General Polygons" (page 299) for the definition of a general polygon.

Q3GeneralPolygon_New

You can use the `Q3GeneralPolygon_New` function to create a new general polygon.

```
TQ3GeometryObject Q3GeneralPolygon_New (
                    const TQ3GeneralPolygonData *generalPolygonData);
```

`generalPolygonData`
A pointer to a `TQ3GeneralPolygonData` structure.

DESCRIPTION

The `Q3GeneralPolygon_New` function returns, as its function result, a new general polygon having the contours and attributes specified by the `generalPolygonData` parameter. If a new general polygon could not be created, `Q3GeneralPolygon_New` returns the value `NULL`.

Q3GeneralPolygon_Submit

You can use the `Q3GeneralPolygon_Submit` function to submit an immediate general polygon for drawing, picking, bounding, or writing.

```
TQ3Status Q3GeneralPolygon_Submit (
                    const TQ3GeneralPolygonData *generalPolygonData,
                    TQ3ViewObject view);
```

`generalPolygonData`
A pointer to a `TQ3GeneralPolygonData` structure.

`view` A view.

DESCRIPTION

The `Q3GeneralPolygon_Submit` function submits for drawing, picking, bounding, or writing the immediate general polygon whose shape and attribute set are specified by the `generalPolygonData` parameter. The general polygon is drawn, picked, bounded, or written according to the view characteristics specified in the `view` parameter.

SPECIAL CONSIDERATIONS

You should call this function only in a submitting loop.

Q3GeneralPolygon_GetData

You can use the `Q3GeneralPolygon_GetData` function to get the data that defines a general polygon and its attributes.

```
TQ3Status Q3GeneralPolygon_GetData (
                    TQ3GeometryObject generalPolygon,
                    TQ3GeneralPolygonData *generalPolygonData);
```

`generalPolygon`
A general polygon.

`generalPolygonData`
: On exit, a pointer to a `TQ3GeneralPolygonData` structure that contains information about the general polygon specified by the `generalPolygon` parameter.

DESCRIPTION

The `Q3GeneralPolygon_GetData` function returns, through the `generalPolygonData` parameter, information about the general polygon specified by the `generalPolygon` parameter. QuickDraw 3D allocates memory for the `TQ3GeneralPolygonData` structure internally; you must call `Q3GeneralPolygon_EmptyData` to dispose of that memory.

Q3GeneralPolygon_SetData

You can use the `Q3GeneralPolygon_SetData` function to set the data that defines a general polygon and its attributes.

```
TQ3Status Q3GeneralPolygon_SetData (
                    TQ3GeometryObject generalPolygon,
                    const TQ3GeneralPolygonData *generalPolygonData);
```

`generalPolygon`
: A general polygon.

`generalPolygonData`
: A pointer to a `TQ3GeneralPolygonData` structure.

DESCRIPTION

The `Q3GeneralPolygon_SetData` function sets the data associated with the general polygon specified by the `generalPolygon` parameter to the data specified by the `generalPolygonData` parameter.

Q3GeneralPolygon_EmptyData

You can use the `Q3GeneralPolygon_EmptyData` function to release the memory occupied by the data structure returned by a previous call to `Q3GeneralPolygon_GetData`.

```
TQ3Status Q3GeneralPolygon_EmptyData (
                    TQ3GeneralPolygonData *generalPolygonData);
```

`generalPolygonData`
A pointer to a `TQ3GeneralPolygonData` structure.

DESCRIPTION

The `Q3GeneralPolygon_EmptyData` function releases the memory occupied by the `TQ3GeneralPolygonData` structure pointed to by the `generalPolygonData` parameter; that memory was allocated by a previous call to `Q3GeneralPolygon_GetData`.

Q3GeneralPolygon_GetVertexPosition

You can use the `Q3GeneralPolygon_GetVertexPosition` function to get the position of a vertex of a general polygon.

```
TQ3Status Q3GeneralPolygon_GetVertexPosition (
                    TQ3GeometryObject generalPolygon,
                    unsigned long contourIndex,
                    unsigned long pointIndex,
                    TQ3Point3D *position);
```

`generalPolygon`
A general polygon.

`contourIndex`
An index into the `contours` array of the specified general polygon. This index should be greater than or equal to 0 and less than the number of contours in the `contours` array.

`pointIndex` An index into the `vertices` array of the specified contour. This index should be greater than or equal to 0 and less than the number of points in the `vertices` array.

`position` On exit, the position of the specified vertex.

DESCRIPTION

The `Q3GeneralPolygon_GetVertexPosition` function returns, in the `position` parameter, the position of a vertex in the general polygon specified by the `generalPolygon` parameter. The vertex has the index specified by the `pointIndex` parameter in the `vertices` array of the contour specified by the `contourIndex` parameter.

Q3GeneralPolygon_SetVertexPosition

You can use the `Q3GeneralPolygon_SetVertexPosition` function to set the position of a vertex of a general polygon.

```
TQ3Status Q3GeneralPolygon_SetVertexPosition (
                    TQ3GeometryObject generalPolygon,
                    unsigned long contourIndex,
                    unsigned long pointIndex,
                    const TQ3Point3D *position);
```

`generalPolygon`
A general polygon.

`contourIndex` An index into the `contours` array of the specified general polygon.

`pointIndex` An index into the `vertices` array of the specified contour.

`position` The desired position of the specified vertex.

DESCRIPTION

The `Q3GeneralPolygon_SetVertexPosition` function sets the position of a vertex in the general polygon specified by the `generalPolygon` parameter. The vertex has the index specified by the `pointIndex` parameter in the `vertices` array of the

contour specified by the `contourIndex` parameter to the position specified in the `position` parameter.

Q3GeneralPolygon_GetVertexAttributeSet

You can use the `Q3GeneralPolygon_GetVertexAttributeSet` function to get the attribute set of a vertex of a general polygon.

```
TQ3Status Q3GeneralPolygon_GetVertexAttributeSet (
                    TQ3GeometryObject generalPolygon,
                    unsigned long contourIndex,
                    unsigned long pointIndex,
                    TQ3AttributeSet *attributeSet);
```

`generalPolygon`
A general polygon.

`contourIndex` An index into the `contours` array of the specified general polygon.

`pointIndex` An index into the `vertices` array of the specified contour.

`attributeSet` On exit, a pointer to a vertex attribute set for the specified vertex.

DESCRIPTION

The `Q3GeneralPolygon_GetVertexAttributeSet` function returns, in the `attributeSet` parameter, the set of attributes for the vertex having the index specified by the `pointIndex` parameter in the `vertices` array of the contour specified by the `contourIndex` parameter of the general polygon specified by the `generalPolygon` parameter. The reference count of the set is incremented.

Q3GeneralPolygon_SetVertexAttributeSet

You can use the `Q3GeneralPolygon_SetVertexAttributeSet` function to set the attribute set of a vertex of a general polygon.

```
TQ3Status Q3GeneralPolygon_SetVertexAttributeSet (
                    TQ3GeometryObject generalPolygon,
                    unsigned long contourIndex,
                    unsigned long pointIndex,
                    TQ3AttributeSet attributeSet);
```

`generalPolygon`
A general polygon.

`contourIndex`
An index into the `contours` array of the specified general polygon.

`pointIndex` An index into the `vertices` array of the specified contour.

`attributeSet`
The desired set of attributes for the specified vertex.

DESCRIPTION

The `Q3GeneralPolygon_SetVertexAttributeSet` function sets the attribute set of the vertex having the index specified by the `pointIndex` parameter in the `vertices` array of the contour specified by the `contourIndex` parameter in the general polygon specified by the `generalPolygon` parameter to the set specified in the `attributeSet` parameter.

Q3GeneralPolygon_GetShapeHint

You can use the `Q3GeneralPolygon_GetShapeHint` function to get the shape hint of a general polygon.

```
TQ3Status Q3GeneralPolygon_GetShapeHint (
                    TQ3GeometryObject generalPolygon,
                    TQ3GeneralPolygonShapeHint *shapeHint);
```

`generalPolygon`
A general polygon.

`shapeHint` On exit, the shape hint of the specified general polygon.

DESCRIPTION

The `Q3GeneralPolygon_GetShapeHint` function returns, in the `shapeHint` parameter, the shape hint of the general polygon specified by the `generalPolygon` parameter. See "General Polygons" (page 299) for a description of the available shape hints.

Q3GeneralPolygon_SetShapeHint

You can use the `Q3GeneralPolygon_SetShapeHint` function to set the shape hint of a general polygon.

```
TQ3Status Q3GeneralPolygon_SetShapeHint (
                    TQ3GeometryObject generalPolygon,
                    TQ3GeneralPolygonShapeHint shapeHint);
```

`generalPolygon`
A general polygon.

`shapeHint` The desired shape hint of the specified general polygon.

DESCRIPTION

The `Q3GeneralPolygon_SetShapeHint` function sets the shape hint of the general polygon specified by the `generalPolygon` parameter to the hint specified in the `shapeHint` parameter. See "General Polygons" (page 299) for a description of the available shape hints.

Creating and Editing Boxes

QuickDraw 3D provides routines that you can use to create and manipulate boxes. See "Boxes" (page 301) for the definition of a box.

Q3Box_New

You can use the `Q3Box_New` function to create a new box.

```
TQ3GeometryObject Q3Box_New (const TQ3BoxData *boxData);
```

`boxData` A pointer to a `TQ3BoxData` structure.

DESCRIPTION

The `Q3Box_New` function returns, as its function result, a new box having the sides and attributes specified by the `boxData` parameter. If a new box could not be created, `Q3Box_New` returns the value `NULL`.

Q3Box_Submit

You can use the `Q3Box_Submit` function to submit an immediate box for drawing, picking, bounding, or writing.

```
TQ3Status Q3Box_Submit (
                    const TQ3BoxData *boxData,
                    TQ3ViewObject view);
```

`boxData` A pointer to a `TQ3BoxData` structure.

`view` A view.

DESCRIPTION

The `Q3Box_Submit` function submits for drawing, picking, bounding, or writing the immediate box whose shape and attribute set are specified by the `boxData` parameter. The box is drawn, picked, bounded, or written according to the view characteristics specified in the `view` parameter.

SPECIAL CONSIDERATIONS

You should call this function only in a submitting loop.

Q3Box_GetData

You can use the `Q3Box_GetData` function to get the data that defines a box and its attributes.

```
TQ3Status Q3Box_GetData (
                    TQ3GeometryObject box,
                    TQ3BoxData *boxData);
```

`box` A box.

`boxData` On exit, a pointer to a `TQ3BoxData` structure that contains information about the box specified by the `box` parameter.

DESCRIPTION

The `Q3Box_GetData` function returns, through the `boxData` parameter, information about the box specified by the `box` parameter. QuickDraw 3D allocates memory for the `TQ3BoxData` structure internally; you must call `Q3Box_EmptyData` to dispose of that memory.

Q3Box_SetData

You can use the `Q3Box_SetData` function to set the data that defines a box and its attributes.

```
TQ3Status Q3Box_SetData (
                    TQ3GeometryObject box,
                    const TQ3BoxData *boxData);
```

`box` A box.

`boxData` A pointer to a `TQ3BoxData` structure.

DESCRIPTION

The `Q3Box_SetData` function sets the data associated with the box specified by the `box` parameter to the data specified by the `boxData` parameter.

Q3Box_EmptyData

You can use the `Q3Box_EmptyData` function to release the memory occupied by the data structure returned by a previous call to `Q3Box_GetData`.

```
TQ3Status Q3Box_EmptyData (TQ3BoxData *boxData);
```

`boxData` A pointer to a `TQ3BoxData` structure.

DESCRIPTION

The `Q3Box_EmptyData` function releases the memory occupied by the `TQ3BoxData` structure pointed to by the `boxData` parameter; that memory was allocated by a previous call to `Q3Box_GetData`.

Q3Box_GetOrigin

You can use the `Q3Box_GetOrigin` function to get the origin of a box.

```
TQ3Status Q3Box_GetOrigin (
                    TQ3GeometryObject box,
                    TQ3Point3D *origin);
```

`box` A box.

`origin` On exit, the origin of the specified box.

DESCRIPTION

The `Q3Box_GetOrigin` function returns, in the `origin` parameter, the origin of the box specified by the `box` parameter.

Q3Box_SetOrigin

You can use the `Q3Box_SetOrigin` function to set the origin of a box.

```
TQ3Status Q3Box_SetOrigin (
                    TQ3GeometryObject box,
                    const TQ3Point3D *origin);
```

`box` A box.

`origin` The desired origin of the specified box.

DESCRIPTION

The `Q3Box_SetOrigin` function sets the origin of the box specified by the `box` parameter to that specified in the `origin` parameter.

Q3Box_GetOrientation

You can use the `Q3Box_GetOrientation` function to get the orientation of a box.

```
TQ3Status Q3Box_GetOrientation (
                    TQ3GeometryObject box,
                    TQ3Vector3D *orientation);
```

`box` A box.

`orientation` On exit, the orientation of the specified box.

DESCRIPTION

The `Q3Box_GetOrientation` function returns, in the `orientation` parameter, the orientation of the box specified by the `box` parameter.

Q3Box_SetOrientation

You can use the `Q3Box_SetOrientation` function to set the orientation of a box.

```
TQ3Status Q3Box_SetOrientation (
                    TQ3GeometryObject box,
                    const TQ3Vector3D *orientation);
```

`box` A box.

`orientation` The desired orientation of the specified box.

DESCRIPTION

The `Q3Box_SetOrientation` function sets the orientation of the box specified by the `box` parameter to that specified in the `orientation` parameter.

Q3Box_GetMajorAxis

You can use the `Q3Box_GetMajorAxis` function to get the major axis of a box.

```
TQ3Status Q3Box_GetMajorAxis (
                    TQ3GeometryObject box,
                    TQ3Vector3D *majorAxis);
```

`box` A box.

`majorAxis` On exit, the major axis of the specified box.

DESCRIPTION

The `Q3Box_GetMajorAxis` function returns, in the `majorAxis` parameter, the major axis of the box specified by the `box` parameter.

Q3Box_SetMajorAxis

You can use the `Q3Box_SetMajorAxis` function to set the major axis of a box.

```
TQ3Status Q3Box_SetMajorAxis (
                    TQ3GeometryObject box,
                    const TQ3Vector3D *majorAxis);
```

`box` A box.

`majorAxis` The desired major axis of the specified box.

DESCRIPTION

The `Q3Box_SetMajorAxis` function sets the major axis of the box specified by the `box` parameter to that specified in the `majorAxis` parameter.

Q3Box_GetMinorAxis

You can use the `Q3Box_GetMinorAxis` function to get the minor axis of a box.

```
TQ3Status Q3Box_GetMinorAxis (
                    TQ3GeometryObject box,
                    TQ3Vector3D *minorAxis);
```

`box` A box.

`minorAxis` On exit, the minor axis of the specified box.

DESCRIPTION

The `Q3Box_GetMinorAxis` function returns, in the `minorAxis` parameter, the minor axis of the box specified by the `box` parameter.

Q3Box_SetMinorAxis

You can use the `Q3Box_SetMinorAxis` function to set the minor axis of a box.

```
TQ3Status Q3Box_SetMinorAxis (
                    TQ3GeometryObject box,
                    const TQ3Vector3D *minorAxis);
```

`box` A box.

`minorAxis` The desired minor axis of the specified box.

DESCRIPTION

The `Q3Box_SetMinorAxis` function sets the minor axis of the box specified by the `box` parameter to that specified in the `minorAxis` parameter.

Q3Box_GetFaceAttributeSet

You can use the `Q3Box_GetFaceAttributeSet` function to get the attribute set of a face of a box.

```
TQ3Status Q3Box_GetFaceAttributeSet (
                    TQ3GeometryObject box,
                    unsigned long faceIndex,
                    TQ3AttributeSet *faceAttributeSet);
```

`box` A box.

`faceIndex` An index into the array of faces for the specified box.

`faceAttributeSet`
On exit, a pointer to an attribute set for the specified face.

DESCRIPTION

The `Q3Box_GetFaceAttributeSet` function returns, in the `faceAttributeSet` parameter, the set of attributes for the face having the index `faceIndex` of the

box specified by the `box` parameter. The reference count of the set is incremented.

Q3Box_SetFaceAttributeSet

You can use the `Q3Box_SetFaceAttributeSet` function to set the attribute set of a face of a box.

```
TQ3Status Q3Box_SetFaceAttributeSet (
                    TQ3GeometryObject box,
                    unsigned long faceIndex,
                    TQ3AttributeSet faceAttributeSet);
```

`box` A box.

`faceIndex` An index into the array of faces for the specified box.

`faceAttributeSet`
The desired set of attributes for the specified face.

DESCRIPTION

The `Q3Box_SetFacetAttributeSet` function sets the attribute set of the face having index `faceIndex` of the box specified by the `box` parameter to the set specified by the `faceAttributeSet` parameter.

Creating and Editing Trigrids

QuickDraw 3D provides routines that you can use to create and manipulate trigrids. See “Trigrids” (page 304) for the definition of a trigrid.

Q3TriGrid_New

You can use the `Q3TriGrid_New` function to create a new trigrid.

```
TQ3GeometryObject Q3TriGrid_New (
                    const TQ3TriGridData *triGridData);
```

`triGridData` A pointer to a `TQ3TriGridData` structure.

DESCRIPTION

The `Q3TriGrid_New` function returns, as its function result, a new trigrid having the vertices and attributes specified by the `triGridData` parameter. If a new trigrid could not be created, `Q3TriGrid_New` returns the value `NULL`.

Q3TriGrid_Submit

You can use the `Q3TriGrid_Submit` function to submit an immediate trigrid for drawing, picking, bounding, or writing.

```
TQ3Status Q3TriGrid_Submit (
                    const TQ3TriGridData *triGridData,
                    TQ3ViewObject view);
```

`triGridData` A pointer to a `TQ3TriGridData` structure.

`view` A view.

DESCRIPTION

The `Q3TriGrid_Submit` function submits for drawing, picking, bounding, or writing the immediate trigrid whose shape and attribute set are specified by the `triGridData` parameter. The trigrid is drawn, picked, bounded, or written according to the view characteristics specified in the `view` parameter.

SPECIAL CONSIDERATIONS

You should call this function only in a submitting loop.

Q3TriGrid_GetData

You can use the `Q3TriGrid_GetData` function to get the data that defines a trigrid and its attributes.

```
TQ3Status Q3TriGrid_GetData (
                    TQ3GeometryObject trigrid,
                    TQ3TriGridData *triGridData);
```

`trigrid` A trigrid.

`triGridData` On exit, a pointer to a `TQ3TriGridData` structure that contains information about the trigrid specified by the `trigrid` parameter.

DESCRIPTION

The `Q3TriGrid_GetData` function returns, through the `triGridData` parameter, information about the trigrid specified by the `trigrid` parameter. QuickDraw 3D allocates memory for the `TQ3TriGridData` structure internally; you must call `Q3TriGrid_EmptyData` to dispose of that memory.

Q3TriGrid_SetData

You can use the `Q3TriGrid_SetData` function to set the data that defines a trigrid and its attributes.

```
TQ3Status Q3TriGrid_SetData (
                    TQ3GeometryObject trigrid,
                    const TQ3TriGridData *triGridData);
```

`trigrid` A trigrid.

`triGridData` A pointer to a `TQ3TriGridData` structure.

DESCRIPTION

The `Q3TriGrid_SetData` function sets the data associated with the trigrid specified by the `trigrid` parameter to the data specified by the `triGridData` parameter.

Q3TriGrid_EmptyData

You can use the `Q3TriGrid_EmptyData` function to release the memory occupied by the data structure returned by a previous call to `Q3TriGrid_GetData`.

```
TQ3Status Q3TriGrid_EmptyData (TQ3TriGridData *triGridData);
```

`triGridData` A pointer to a `TQ3TriGridData` structure.

DESCRIPTION

The `Q3TriGrid_EmptyData` function releases the memory occupied by the `TQ3TriGridData` structure pointed to by the `triGridData` parameter; that memory was allocated by a previous call to `Q3TriGrid_GetData`.

Q3TriGrid_GetVertexPosition

You can use the `Q3TriGrid_GetVertexPosition` function to get the position of a vertex of a trigrid.

```
TQ3Status Q3TriGrid_GetVertexPosition (
                    TQ3GeometryObject triGrid,
                    unsigned long rowIndex,
                    unsigned long columnIndex,
                    TQ3Point3D *position);
```

`triGrid` A trigrid.

`rowIndex` A row index into the `vertices` array of the specified trigrid.

`columnIndex` A column index into the `vertices` array of the specified trigrid.

`position` On exit, the position of the specified vertex.

DESCRIPTION

The `Q3TriGrid_GetVertexPosition` function returns, in the `position` parameter, the position of the vertex having row and column indices `rowIndex` and

columnIndex in the vertices array of the trigrid specified by the triGrid parameter.

Q3TriGrid_SetVertexPosition

You can use the Q3TriGrid_SetVertexPosition function to set the position of a vertex of a trigrid.

```
TQ3Status Q3TriGrid_SetVertexPosition (
                    TQ3GeometryObject triGrid,
                    unsigned long rowIndex,
                    unsigned long columnIndex,
                    const TQ3Point3D *position);
```

triGrid	A trigrid.
rowIndex	A row index into the vertices array of the specified trigrid.
columnIndex	A column index into the vertices array of the specified trigrid.
position	The desired position of the specified vertex.

DESCRIPTION

The Q3TriGrid_SetVertexPosition function sets the position of the vertex having row and column indices rowIndex and columnIndex in the vertices array of the trigrid specified by the triGrid parameter to that specified in the position parameter.

Q3TriGrid_GetVertexAttributeSet

You can use the Q3TriGrid_GetVertexAttributeSet function to get the attribute set of a vertex of a trigrid.

```
TQ3Status Q3TriGrid_GetVertexAttributeSet (
                    TQ3GeometryObject triGrid,
                    unsigned long rowIndex,
                    unsigned long columnIndex,
                    TQ3AttributeSet *attributeSet);
```

`triGrid` A trigrid.

`rowIndex` A row index into the `vertices` array of the specified trigrid.

`columnIndex` A column index into the `vertices` array of the specified trigrid.

`attributeSet` On exit, a pointer to a vertex attribute set for the specified vertex.

DESCRIPTION

The `Q3TriGrid_GetVertexAttributeSet` function returns, in the `attributeSet` parameter, the set of attributes for the vertex having row and column indices `rowIndex` and `columnIndex` in the `vertices` array of the trigrid specified by the `triGrid` parameter. The reference count of the set is incremented.

Q3TriGrid_SetVertexAttributeSet

You can use the `Q3TriGrid_SetVertexAttributeSet` function to set the attribute set of a vertex of a trigrid.

```
TQ3Status Q3TriGrid_SetVertexAttributeSet (
                    TQ3GeometryObject triGrid,
                    unsigned long rowIndex,
                    unsigned long columnIndex,
                    TQ3AttributeSet attributeSet);
```

`triGrid` A trigrid.

`rowIndex` A row index into the `vertices` array of the specified trigrid.

`columnIndex` A column index into the `vertices` array of the specified trigrid.

`attributeSet` The desired set of attributes for the specified vertex.

DESCRIPTION

The `Q3TriGrid_SetVertexAttributeSet` function sets the attribute set of the vertex having row and column indices `rowIndex` and `columnIndex` in the `vertices` array of the trigrid specified by the `triGrid` parameter to the set specified in the `attributeSet` parameter.

Q3TriGrid_GetFacetAttributeSet

You can use the `Q3TriGrid_GetFacetAttributeSet` function to get the attribute set of a facet of a trigrid.

```
TQ3Status Q3TriGrid_GetFacetAttributeSet (
                    TQ3GeometryObject triGrid,
                    unsigned long faceIndex,
                    TQ3AttributeSet *facetAttributeSet);
```

`triGrid` A trigrid.

`faceIndex` An index into the array of facets for the specified trigrid.

`facetAttributeSet`
On exit, a pointer to an attribute set for the specified facet.

DESCRIPTION

The `Q3TriGrid_GetFacetAttributeSet` function returns, in the `facetAttributeSet` parameter, the set of attributes for the facet having the index `faceIndex` of the trigrid specified by the `triGrid` parameter. The reference count of the set is incremented.

Q3TriGrid_SetFacetAttributeSet

You can use the `Q3TriGrid_SetFacetAttributeSet` function to set the attribute set of a facet of a trigrid.

```
TQ3Status Q3TriGrid_SetFacetAttributeSet (
                    TQ3GeometryObject triGrid,
                    unsigned long faceIndex,
                    TQ3AttributeSet facetAttributeSet);
```

`triGrid` A trigrid.

`faceIndex` An index into the array of facets for the specified trigrid.

`facetAttributeSet`
The desired set of attributes for the specified facet.

DESCRIPTION

The `Q3TriGrid_SetFacetAttributeSet` function sets the attribute set of the facet having index `faceIndex` of the trigrid specified by the `triGrid` parameter to the set specified by the `facetAttributeSet` parameter.

Creating and Editing Meshes

QuickDraw 3D provides routines that you can use to create and manipulate meshes. See "Meshes" (page 305) for the definition of a mesh and its associated types.

Q3Mesh_New

You can use the `Q3Mesh_New` function to create a new mesh.

```
TQ3GeometryObject Q3Mesh_New (void);
```

DESCRIPTION

The `Q3Mesh_New` function returns, as its function result, a new mesh. The new mesh is empty; you need to call other QuickDraw 3D routines to add vertices and faces to the mesh. If a new mesh could not be created, `Q3Mesh_New` returns the value `NULL`.

Q3Mesh_VertexNew

You can use the `Q3Mesh_VertexNew` function to add a vertex to a mesh.

```
TQ3MeshVertex Q3Mesh_VertexNew (
                    TQ3GeometryObject mesh,
                    const TQ3Vertex3D *vertex);
```

`mesh` A mesh.

`vertex` A three-dimensional vertex.

DESCRIPTION

The `Q3Mesh_VertexNew` function adds the vertex specified by the `vertex` parameter to the mesh specified by the `mesh` parameter. The mesh must already exist before you call `Q3Mesh_VertexNew`. The new mesh vertex is returned as the function result, of type `TQ3MeshVertex`.

Q3Mesh_VertexDelete

You can use the `Q3Mesh_VertexDelete` function to delete a vertex from a mesh.

```
TQ3Status Q3Mesh_VertexDelete (
                    TQ3GeometryObject mesh,
                    TQ3MeshVertex vertex);
```

`mesh` A mesh.

`vertex` A mesh vertex.

DESCRIPTION

The `Q3Mesh_VertexDelete` function deletes the mesh vertex specified by the `vertex` parameter from the mesh specified by the `mesh` parameter. All mesh faces that contain the vertex are also deleted.

Q3Mesh_FaceNew

You can use the `Q3Mesh_FaceNew` function to add a face to a mesh.

```
TQ3MeshFace Q3Mesh_FaceNew (
                    TQ3GeometryObject mesh,
                    unsigned long numVertices,
                    const TQ3MeshVertex *vertices,
                    TQ3AttributeSet attributeSet);
```

`mesh` A mesh.

`numVertices` The number of mesh vertices in the `vertices` array.

`vertices` A pointer to an array of mesh vertices defining the new mesh face. These vertices can be ordered either clockwise or counterclockwise.

`attributeSet` The desired set of attributes for the new mesh face. Set this parameter to `NULL` if you do no want any attributes for the new face.

DESCRIPTION

The `Q3Mesh_FaceNew` function adds the face specified by the `vertices` parameter to the mesh specified by the `mesh` parameter. The mesh must already exist before you call `Q3Mesh_FaceNew`. The new mesh face is returned as the function result.

Q3Mesh_FaceDelete

You can use the `Q3Mesh_FaceDelete` function to delete a face from a mesh.

```
TQ3Status Q3Mesh_FaceDelete (
                    TQ3GeometryObject mesh,
                    TQ3MeshFace face);
```

`mesh` A mesh.

`face` A mesh face.

DESCRIPTION

The `Q3Mesh_FaceDelete` function deletes the mesh face specified by the `face` parameter from the mesh specified by the `mesh` parameter. The vertices of the face are not deleted.

Q3Mesh_DelayUpdates

You can use the `Q3Mesh_DelayUpdates` function to prevent QuickDraw 3D from updating its internal list of mesh components.

```
TQ3Status Q3Mesh_DelayUpdates (TQ3GeometryObject mesh);
```

`mesh` A mesh.

DESCRIPTION

The `Q3Mesh_DelayUpdates` function prevents QuickDraw 3D from updating its internal list of components and maintaining correct face orientation (that is, vertex ordering) for the mesh specified by the `mesh` parameter. Updating the list of components can consume significant amounts of time, and it might be useful temporarily to prevent component list updating. You should later call `Q3Mesh_ResumeUpdates` to resume component list updating. Generally, if you are creating or deleting a number of vertices or faces from a mesh, it is better to bracket the entire set of changes with calls to `Q3Mesh_DelayUpdates` and `Q3Mesh_ResumeUpdates`.

Q3Mesh_ResumeUpdates

You can use the `Q3Mesh_ResumeUpdates` function to have QuickDraw 3D resume updating its internal list of mesh components.

```
TQ3Status Q3Mesh_ResumeUpdates (TQ3GeometryObject mesh);
```

`mesh` A mesh.

DESCRIPTION

The `Q3Mesh_ResumeUpdates` function instructs QuickDraw 3D to resume updating its internal list of components and maintaining correct face orientation for the mesh specified by the `mesh` parameter.

Q3Mesh_FaceToContour

You can use the `Q3Mesh_FaceToContour` function to convert a face of a mesh into a contour. The contour is then attached to another mesh face as a hole.

```
TQ3MeshContour Q3Mesh_FaceToContour (
                    TQ3GeometryObject mesh,
                    TQ3MeshFace containerFace,
                    TQ3MeshFace face);
```

`mesh` A mesh.

`containerFace`
The mesh face that is to contain the new contour.

`face` The mesh face that is to be converted into a contour. On exit, this face is no longer a valid object.

DESCRIPTION

The `Q3Mesh_FaceToContour` function returns, as its function result, a new contour created from the mesh face specified by the `mesh` and `face` parameters. The new contour is contained in the mesh face specified by the `mesh` and `containerFace` parameters. If a new contour could not be created, `Q3Mesh_FaceToContour` returns the value `NULL`.

IMPORTANT

`Q3Mesh_FaceToContour` destroys any attributes associated with the face specified by the `face` parameter. ▲

Q3Mesh_ContourToFace

You can use the `Q3Mesh_ContourToFace` function to convert a mesh contour into a mesh face.

```
TQ3MeshFace Q3Mesh_ContourToFace (
                    TQ3GeometryObject mesh,
                    TQ3MeshContour contour);
```

`mesh` A mesh.

`contour` A mesh contour. On exit, this contour is no longer a valid object.

DESCRIPTION

The `Q3Mesh_ContourToFace` function returns, as its function result, a mesh face that is the result of removing the mesh contour specified by the `mesh` and `contour` parameters from its containing face. (You can call the `Q3Mesh_GetContourFace` function to determine the face that contains a mesh contour; see page 407.) If a new face could not be created, `Q3Mesh_ContourToFace` returns the value `NULL`.

Q3Mesh_GetNumComponents

You can use the `Q3Mesh_GetNumComponents` function to determine the number of connected components of a mesh.

```
TQ3Status Q3Mesh_GetNumComponents (
                    TQ3GeometryObject mesh,
                    unsigned long *numComponents);
```

`mesh` A mesh.

`numComponents`
On exit, the number of connected components in the specified mesh.

DESCRIPTION

The `Q3Mesh_GetNumComponents` function returns, in the `numComponents` parameter, the number of connected components in the mesh specified by the `mesh` parameter. A connected component is a list of vertices, each of which is connected to all the others by some sequence of mesh edges. For example, a mesh that contains two cubes has two components.

SPECIAL CONSIDERATIONS

The `Q3Mesh_GetNumComponents` function might not accurately report the number of connected components in a mesh if called while mesh updating is delayed (that is, after a call to `Q3Mesh_DelayUpdates` but before the matching call to `Q3Mesh_ResumeUpdates`).

Q3Mesh_GetNumEdges

You can use the `Q3Mesh_GetNumEdges` function to determine the number of edges of a mesh.

```
TQ3Status Q3Mesh_GetNumEdges (
                    TQ3GeometryObject mesh,
                    unsigned long *numEdges);
```

`mesh` A mesh.

`numEdges` On exit, the number of edges in the specified mesh.

DESCRIPTION

The `Q3Mesh_GetNumEdges` function returns, in the `numEdges` parameter, the number of edges in the mesh specified by the `mesh` parameter.

Q3Mesh_GetNumVertices

You can use the `Q3Mesh_GetNumVertices` function to determine the number of vertices of a mesh.

```
TQ3Status Q3Mesh_GetNumVertices (
                    TQ3GeometryObject mesh,
                    unsigned long *numVertices);
```

`mesh` A mesh.

`numVertices` On exit, the number of vertices in the specified mesh.

DESCRIPTION

The `Q3Mesh_GetNumVertices` function returns, in the `numVertices` parameter, the number of vertices in the mesh specified by the `mesh` parameter.

Q3Mesh_GetNumFaces

You can use the `Q3Mesh_GetNumFaces` function to determine the number of faces of a mesh.

```
TQ3Status Q3Mesh_GetNumFaces (
                    TQ3GeometryObject mesh,
                    unsigned long *numFaces);
```

`mesh` A mesh.

`numFaces` On exit, the number of faces in the specified mesh.

DESCRIPTION

The `Q3Mesh_GetNumFaces` function returns, in the `numFaces` parameter, the number of faces in the mesh specified by the `mesh` parameter.

Q3Mesh_GetNumCorners

You can use the `Q3Mesh_GetNumCorners` function to determine the number of corners of a mesh that have attribute sets.

```
TQ3Status Q3Mesh_GetNumCorners (
                    TQ3GeometryObject mesh,
                    unsigned long *numCorners);
```

`mesh` A mesh.

`numCorners` On exit, the number of corners in the specified mesh that have attribute sets.

DESCRIPTION

The `Q3Mesh_GetNumCorners` function returns, in the `numCorners` parameter, the number of corners in the mesh specified by the `mesh` parameter that have attribute sets attached to them.

Q3Mesh_GetOrientable

You can use the `Q3Mesh_GetOrientable` function to determine whether the faces of a mesh can be consistently oriented.

```
TQ3Status Q3Mesh_GetOrientable (
                    TQ3GeometryObject mesh,
                    TQ3Boolean *orientable);
```

`mesh` A mesh.

`orientable` On exit, a Boolean value that indicates whether the faces of the specified mesh can be consistently oriented.

DESCRIPTION

The `Q3Mesh_GetOrientable` function returns, in the `orientable` parameter, the value `kQ3True` if the faces of the mesh specified by the `mesh` parameter can be consistently oriented; `Q3Mesh_GetOrientable` returns `kQ3False` otherwise. For example, the faces of a tessellated Möbius strip or a Klein bottle cannot be consistently oriented.

SPECIAL CONSIDERATIONS

The `Q3Mesh_GetOrientable` function might not accurately report the orientation state of a mesh if called while mesh updating is delayed (that is, after a call to `Q3Mesh_DelayUpdates` but before the matching call to `Q3Mesh_ResumeUpdates`).

Q3Mesh_GetComponentNumVertices

You can use the `Q3Mesh_GetComponentNumVertices` function to determine the number of vertices in a component of a mesh.

```
TQ3Status Q3Mesh_GetComponentNumVertices (
                    TQ3GeometryObject mesh,
                    TQ3MeshComponent component,
                    unsigned long *numVertices);
```

`mesh`	A mesh.
`component`	A mesh component.
`numVertices`	On exit, the number of vertices in the specified mesh component.

DESCRIPTION

The `Q3Mesh_GetComponentNumVertices` function returns, in the `numVertices` parameter, the number of vertices in the mesh component specified by the `mesh` and `component` parameters.

SPECIAL CONSIDERATIONS

The `Q3Mesh_GetComponentNumVertices` function might not accurately report the number of vertices in a mesh component if called while mesh updating is delayed (that is, after a call to `Q3Mesh_DelayUpdates` but before the matching call to `Q3Mesh_ResumeUpdates`).

Q3Mesh_GetComponentNumEdges

You can use the `Q3Mesh_GetComponentNumEdges` function to determine the number of edges in a component of a mesh.

```
TQ3Status Q3Mesh_GetComponentNumEdges (
                    TQ3GeometryObject mesh,
                    TQ3MeshComponent component,
                    unsigned long *numEdges);
```

`mesh`	A mesh.
`component`	A mesh component.
`numEdges`	On exit, the number of edges in the specified mesh component.

DESCRIPTION

The `Q3Mesh_GetComponentNumEdges` function returns, in the `numEdges` parameter, the number of edges in the mesh component specified by the `mesh` and `component` parameters.

SPECIAL CONSIDERATIONS

The `Q3Mesh_GetComponentNumEdges` function might not accurately report the number of edges in a mesh component if called while mesh updating is delayed (that is, after a call to `Q3Mesh_DelayUpdates` but before the matching call to `Q3Mesh_ResumeUpdates`).

Q3Mesh_GetComponentBoundingBox

You can use the `Q3Mesh_GetComponentBoundingBox` function to determine the bounding box of a component of a mesh.

```
TQ3Status Q3Mesh_GetComponentBoundingBox (
                    TQ3GeometryObject mesh,
                    TQ3MeshComponent component,
                    TQ3BoundingBox *boundingBox);
```

`mesh`	A mesh.
`component`	A mesh component.
`boundingBox`	On exit, the bounding box of the specified mesh component.

DESCRIPTION

The `Q3Mesh_GetComponentBoundingBox` function returns, in the `boundingBox` parameter, the bounding box of the mesh component specified by the `mesh` and `component` parameters.

SPECIAL CONSIDERATIONS

The `Q3Mesh_GetComponentBoundingBox` function might not accurately report the bounding box of a mesh component if called while mesh updating is delayed (that is, after a call to `Q3Mesh_DelayUpdates` but before the matching call to `Q3Mesh_ResumeUpdates`).

Q3Mesh_GetComponentOrientable

You can use the `Q3Mesh_GetComponentOrientable` function to determine whether the faces of a component of a mesh can be consistently oriented.

```
TQ3Status Q3Mesh_GetComponentOrientable (
                    TQ3GeometryObject mesh,
                    TQ3MeshComponent component,
                    TQ3Boolean *orientable);
```

`mesh`	A mesh.
`component`	A mesh component.
`orientable`	On exit, a Boolean value that indicates whether the faces of the specified mesh component can be consistently oriented.

DESCRIPTION

The `Q3Mesh_GetComponentOrientable` function returns, in the `orientable` parameter, the value `kQ3True` if the faces of the mesh component specified by

the `mesh` and `component` parameters can be consistently oriented; `Q3Mesh_GetComponentOrientable` returns `kQ3False` otherwise.

SPECIAL CONSIDERATIONS

The `Q3Mesh_GetComponentOrientable` function might not accurately report the orientation state of a mesh component if called while mesh updating is delayed (that is, after a call to `Q3Mesh_DelayUpdates` but before the matching call to `Q3Mesh_ResumeUpdates`).

Q3Mesh_GetVertexCoordinates

You can use the `Q3Mesh_GetVertexCoordinates` function to get the coordinates of a vertex of a mesh.

```
TQ3Status Q3Mesh_GetVertexCoordinates (
                    TQ3GeometryObject mesh,
                    TQ3MeshVertex vertex,
                    TQ3Point3D *coordinates);
```

`mesh`	A mesh.
`vertex`	A mesh vertex.
`coordinates`	On exit, the coordinates of the specified mesh vertex.

DESCRIPTION

The `Q3Mesh_GetVertexCoordinates` function returns, in the `coordinates` parameter, the coordinates of the mesh vertex specified by the `mesh` and `vertex` parameters.

Q3Mesh_SetVertexCoordinates

You can use the `Q3Mesh_SetVertexCoordinates` function to set the coordinates of a vertex of a mesh.

```
TQ3Status Q3Mesh_SetVertexCoordinates (
                    TQ3GeometryObject mesh,
                    TQ3MeshVertex vertex,
                    const TQ3Point3D *coordinates);
```

`mesh` A mesh.

`vertex` A mesh vertex.

`coordinates` The desired coordinates of the specified mesh vertex.

DESCRIPTION

The `Q3Mesh_SetVertexCoordinates` function sets the coordinates of the mesh vertex specified by the `mesh` and `vertex` parameters to those specified in the `coordinates` parameter.

Q3Mesh_GetVertexIndex

You can use the `Q3Mesh_GetVertexIndex` function to get the index of a mesh vertex.

```
TQ3Status Q3Mesh_GetVertexIndex (
                    TQ3GeometryObject mesh,
                    TQ3MeshVertex vertex,
                    unsigned long *index);
```

`mesh` A mesh.

`vertex` A mesh vertex.

`index` On exit, the index of the specified mesh vertex.

DESCRIPTION

The `Q3Mesh_GetVertexIndex` function returns, in the `index` parameter, the index of the mesh vertex specified by the `mesh` and `vertex` parameters. A **vertex index** is a unique integer (between 0 and the total number of vertices in the mesh minus 1) associated with a vertex.

▲ **WARNING**
Vertex indices are volatile and can be changed by functions that alter the topology of a mesh (such as functions that add or delete faces or vertices), and by writing, picking, rendering, or duplicating a mesh, or by calling `Q3Mesh_DelayUpdates`. As a result, you should rely on an index returned by `Q3Mesh_GetVertexIndex` only until you perform one of these operations. ▲

Q3Mesh_GetVertexOnBoundary

You can use the `Q3Mesh_GetVertexOnBoundary` function to determine whether a vertex lies on the boundary of a mesh.

```
TQ3Status Q3Mesh_GetVertexOnBoundary (
                    TQ3GeometryObject mesh,
                    TQ3MeshVertex vertex,
                    TQ3Boolean *onBoundary);
```

`mesh`	A mesh.
`vertex`	A mesh vertex.
`onBoundary`	On exit, a Boolean value that indicates whether the specified mesh vertex lies on the boundary of the mesh.

DESCRIPTION

The `Q3Mesh_GetVertexOnBoundary` function returns, in the `onBoundary` parameter, the value `kQ3True` if the mesh vertex specified by the `mesh` and `vertex` parameters lies on the boundary of the mesh. `Q3Mesh_GetVertexOnBoundary` returns `kQ3False` otherwise.

Q3Mesh_GetVertexComponent

You can use the `Q3Mesh_GetVertexComponent` function to get the component of a mesh to which a vertex belongs.

```
TQ3Status Q3Mesh_GetVertexComponent (
                    TQ3GeometryObject mesh,
                    TQ3MeshVertex vertex,
                    TQ3MeshComponent *component);
```

`mesh` A mesh.

`vertex` A mesh vertex.

`component` On exit, the mesh component that contains the specified mesh vertex.

DESCRIPTION

The `Q3Mesh_GetVertexComponent` function returns, in the `component` parameter, the mesh component that contains the mesh vertex specified by the `mesh` and `vertex` parameters.

SPECIAL CONSIDERATIONS

The `Q3Mesh_GetVertexComponent` function might not accurately report the mesh component that contains a mesh vertex if called while mesh updating is delayed (that is, after a call to `Q3Mesh_DelayUpdates` but before the matching call to `Q3Mesh_ResumeUpdates`).

Q3Mesh_GetVertexAttributeSet

You can use the `Q3Mesh_GetVertexAttributeSet` function to get the attribute set of a vertex of a mesh.

```
TQ3Status Q3Mesh_GetVertexAttributeSet (
                    TQ3GeometryObject mesh,
                    TQ3MeshVertex vertex,
                    TQ3AttributeSet *attributeSet);
```

`mesh` A mesh.

`vertex` A mesh vertex.

`attributeSet` On exit, a pointer to the set of attributes for the mesh vertex.

DESCRIPTION

The `Q3Mesh_GetVertexAttributeSet` function returns, in the `attributeSet` parameter, the set of attributes currently associated with the mesh vertex specified by the `mesh` and `vertex` parameters. The reference count of the set is incremented.

Q3Mesh_SetVertexAttributeSet

You can use the `Q3Mesh_SetVertexAttributeSet` function to set the attribute set of a vertex of a mesh.

```
TQ3Status Q3Mesh_SetVertexAttributeSet (
                    TQ3GeometryObject mesh,
                    TQ3MeshVertex vertex,
                    TQ3AttributeSet attributeSet);
```

`mesh` A mesh.

`vertex` A mesh vertex.

`attributeSet`
The desired set of attributes for the specified mesh vertex.

DESCRIPTION

The `Q3Mesh_SetVertexAttributeSet` function sets the attribute set of the mesh vertex specified by the `mesh` and `vertex` parameters to the set of attributes specified by the `attributeSet` parameter.

Q3Mesh_GetFaceNumVertices

You can use the `Q3Mesh_GetFaceNumVertices` function to determine the number of vertices in a face of a mesh.

```
TQ3Status Q3Mesh_GetFaceNumVertices (
                    TQ3GeometryObject mesh,
                    TQ3MeshFace face,
                    unsigned long *numVertices);
```

`mesh` A mesh.

`face` A mesh face.

`numVertices` On exit, the number of vertices in the specified mesh face.

DESCRIPTION

The `Q3Mesh_GetFaceNumVertices` function returns, in the `numVertices` parameter, the number of vertices in the mesh face specified by the `mesh` and `face` parameters.

Q3Mesh_GetFacePlaneEquation

You can use the `Q3Mesh_GetFacePlaneEquation` function to determine the plane equation of a face of a mesh.

```
TQ3Status Q3Mesh_GetFacePlaneEquation (
                    TQ3GeometryObject mesh,
                    TQ3MeshFace face,
                    TQ3PlaneEquation *planeEquation);
```

`mesh` A mesh.

`face` A mesh face.

`planeEquation`
On exit, the plane equation of the plane spanned by the vertices of the specified mesh face.

DESCRIPTION

The `Q3Mesh_GetFacePlaneEquation` function returns, in the `planeEquation` parameter, the plane equation of the plane spanned by the vertices of the mesh face specified by the `mesh` and `face` parameters. If the vertices of the mesh face

do not all lie in one plane, the information returned in the `planeEquation` parameter is only an approximation.

Q3Mesh_GetFaceNumContours

You can use the `Q3Mesh_GetFaceNumContours` function to determine the number of contours in a face of a mesh.

```
TQ3Status Q3Mesh_GetFaceNumContours (
                    TQ3GeometryObject mesh,
                    TQ3MeshFace face,
                    unsigned long *numContours);
```

`mesh` A mesh.

`face` A mesh face.

`numContours`
On exit, the number of contours in the specified mesh face.

DESCRIPTION

The `Q3Mesh_GetFaceNumContours` function returns, in the `numContours` parameter, the number of contours in the mesh face specified by the `mesh` and `face` parameters. A mesh face always contains at least one contour, which defines the face itself. Any additional contours in the face define holes in the face.

Q3Mesh_GetFaceIndex

You can use the `Q3Mesh_GetFaceIndex` function to get the index of a mesh face.

```
TQ3Status Q3Mesh_GetFaceIndex (
                    TQ3GeometryObject mesh,
                    TQ3MeshFace face,
                    unsigned long *index);
```

`mesh`	A mesh.
`face`	A mesh face.
`index`	On exit, the index of the specified mesh face.

DESCRIPTION

The `Q3Mesh_GetFaceIndex` function returns, in the `index` parameter, the index of the mesh face specified by the `mesh` and `face` parameters. A **face index** is a unique integer (between 0 and the total number of faces in the mesh minus 1) associated with a face.

▲ **WARNING**
Face indices are volatile and can be changed by functions that alter the topology of a mesh (such as functions that add or delete faces or vertices), and by writing, picking, rendering, or duplicating a mesh, or by calling `Q3Mesh_DelayUpdates`. As a result, you should rely on an index returned by `Q3Mesh_GetFaceIndex` only until you perform one of these operations. ▲

Q3Mesh_GetFaceComponent

You can use the `Q3Mesh_GetFaceComponent` function to get the component of a mesh to which a face belongs.

```
TQ3Status Q3Mesh_GetFaceComponent (
                    TQ3GeometryObject mesh,
                    TQ3MeshFace face,
                    TQ3MeshComponent *component);
```

`mesh`	A mesh.
`face`	A mesh face.
`component`	On exit, the mesh component that contains the specified mesh face.

DESCRIPTION

The `Q3Mesh_GetFaceComponent` function returns, in the `component` parameter, the mesh component that contains the mesh face specified by the `mesh` and `face` parameters.

SPECIAL CONSIDERATIONS

The `Q3Mesh_GetFaceComponent` function might not accurately report the mesh component that contains a mesh face if called while mesh updating is delayed (that is, after a call to `Q3Mesh_DelayUpdates` but before the matching call to `Q3Mesh_ResumeUpdates`).

Q3Mesh_GetFaceAttributeSet

You can use the `Q3Mesh_GetFaceAttributeSet` function to get the attribute set of a face of a mesh.

```
TQ3Status Q3Mesh_GetFaceAttributeSet (
                    TQ3GeometryObject mesh,
                    TQ3MeshFace face,
                    TQ3AttributeSet *attributeSet);
```

`mesh` A mesh.

`face` A mesh face.

`attributeSet`
On exit, a pointer to the set of attributes for the specified mesh face.

DESCRIPTION

The `Q3Mesh_GetFaceAttributeSet` function returns, in the `attributeSet` parameter, the set of attributes currently associated with the mesh face specified by the `mesh` and `face` parameters. The reference count of the set is incremented.

Q3Mesh_SetFaceAttributeSet

You can use the `Q3Mesh_SetFaceAttributeSet` function to set the attribute set of a face of a mesh.

```
TQ3Status Q3Mesh_SetFaceAttributeSet (
                    TQ3GeometryObject mesh,
                    TQ3MeshFace face,
                    TQ3AttributeSet attributeSet);
```

`mesh` A mesh.

`face` A mesh face.

`attributeSet` The desired set of attributes for the specified mesh face.

DESCRIPTION

The `Q3Mesh_SetFaceAttributeSet` function sets the attribute set of the mesh face specified by the `mesh` and `face` parameters to the set of attributes specified by the `attributeSet` parameter.

Q3Mesh_GetEdgeVertices

You can use the `Q3Mesh_GetEdgeVertices` function to get the vertices of a mesh edge.

```
TQ3Status Q3Mesh_GetEdgeVertices (
                    TQ3GeometryObject mesh,
                    TQ3MeshEdge edge,
                    TQ3MeshVertex *vertex1,
                    TQ3MeshVertex *vertex2);
```

`mesh` A mesh.

`edge` A mesh edge.

`vertex1` On exit, the first vertex of the specified mesh edge.

`vertex2` On exit, the second vertex of the specified mesh edge.

DESCRIPTION

The `Q3Mesh_GetEdgeVertices` function returns, in the `vertex1` and `vertex2` parameters, the two vertices of the mesh edge specified by the `mesh` and `edge` parameters.

Q3Mesh_GetEdgeFaces

You can use the `Q3Mesh_GetEdgeFaces` function to get the faces that share a mesh edge.

```
TQ3Status Q3Mesh_GetEdgeFaces (
                    TQ3GeometryObject mesh,
                    TQ3MeshEdge edge,
                    TQ3MeshFace *face1,
                    TQ3MeshFace *face2);
```

`mesh`	A mesh.
`edge`	A mesh edge.
`face1`	On exit, the first mesh face that shares the specified mesh edge.
`face2`	On exit, the second mesh face that shares the specified mesh edge.

DESCRIPTION

The `Q3Mesh_GetEdgeFaces` function returns, in the `face1` and `face2` parameters, the two mesh faces that shares the mesh edge specified by the `mesh` and `edge` parameters. If the edge lies on the boundary of the mesh, either `face1` or `face2` is `NULL`.

Q3Mesh_GetEdgeOnBoundary

You can use the `Q3Mesh_GetEdgeOnBoundary` function to determine whether a mesh edge lies on the boundary of the mesh.

```
TQ3Status Q3Mesh_GetEdgeOnBoundary (
                    TQ3GeometryObject mesh,
                    TQ3MeshEdge edge,
                    TQ3Boolean *onBoundary);
```

`mesh` A mesh.

`edge` A mesh edge.

`onBoundary` On exit, a Boolean value that indicates whether the specified mesh edge lies on the boundary of the mesh.

DESCRIPTION

The `Q3Mesh_GetEdgeOnBoundary` function returns, in the `onBoundary` parameter, the value `kQ3True` if the mesh edge specified by the `mesh` and `edge` parameters lies on the boundary of the mesh. `Q3Mesh_GetEdgeOnBoundary` returns `kQ3False` otherwise.

Q3Mesh_GetEdgeComponent

You can use the `Q3Mesh_GetEdgeComponent` function to get the component of a mesh to which an edge belongs.

```
TQ3Status Q3Mesh_GetEdgeComponent (
                    TQ3GeometryObject mesh,
                    TQ3MeshEdge edge,
                    TQ3MeshComponent *component);
```

`mesh` A mesh.

`edge` A mesh edge.

`component` On exit, the mesh component that contains the specified mesh edge.

DESCRIPTION

The `Q3Mesh_GetEdgeComponent` function returns, in the `component` parameter, the mesh component that contains the mesh edge specified by the `mesh` and `edge` parameters.

SPECIAL CONSIDERATIONS

The `Q3Mesh_GetEdgeComponent` function might not accurately report the mesh component that contains a mesh edge if called while mesh updating is delayed (that is, after a call to `Q3Mesh_DelayUpdates` but before the matching call to `Q3Mesh_ResumeUpdates`).

Q3Mesh_GetEdgeAttributeSet

You can use the `Q3Mesh_GetEdgeAttributeSet` function to get the attribute set of an edge of a mesh.

```
TQ3Status Q3Mesh_GetEdgeAttributeSet (
                    TQ3GeometryObject mesh,
                    TQ3MeshEdge edge,
                    TQ3AttributeSet *attributeSet);
```

`mesh` A mesh.

`edge` A mesh edge.

`attributeSet`
On exit, a pointer to the set of attributes for the specified mesh edge.

DESCRIPTION

The `Q3Mesh_GetEdgeAttributeSet` function returns, in the `attributeSet` parameter, the set of attributes currently associated with the mesh edge specified by the `mesh` and `edge` parameters. The reference count of the set is incremented.

Q3Mesh_SetEdgeAttributeSet

You can use the `Q3Mesh_SetEdgeAttributeSet` function to set the attribute set of an edge of a mesh.

```
TQ3Status Q3Mesh_SetEdgeAttributeSet (
                    TQ3GeometryObject mesh,
                    TQ3MeshEdge edge,
                    TQ3AttributeSet attributeSet);
```

`mesh` A mesh.

`edge` A mesh edge.

`attributeSet` The desired set of attributes for the specified mesh edge.

DESCRIPTION

The `Q3Mesh_SetEdgeAttributeSet` function sets the attribute set of the mesh edge specified by the `mesh` and `edge` parameters to the set of attributes specified by the `attributeSet` parameter.

Q3Mesh_GetContourFace

You can use the `Q3Mesh_GetContourFace` function to get the mesh face that contains a mesh contour.

```
TQ3Status Q3Mesh_GetContourFace (
                    TQ3GeometryObject mesh,
                    TQ3MeshContour contour,
                    TQ3MeshFace *face);
```

`mesh` A mesh.

`contour` A mesh contour.

`face` On exit, the mesh face that contains the specified contour.

DESCRIPTION

The `Q3Mesh_GetContourFace` function returns, in the `face` parameter, the mesh face that contains the mesh contour specified by the `mesh` and `contour` parameters.

Q3Mesh_GetContourNumVertices

You can use the `Q3Mesh_GetContourNumVertices` function to get the number of vertices that define a contour.

```
TQ3Status Q3Mesh_GetContourNumVertices (
                    TQ3GeometryObject mesh,
                    TQ3MeshContour contour,
                    unsigned long *numVertices);
```

`mesh` A mesh.

`contour` A mesh contour.

`numVertices`
On exit, the number of vertices in the specified mesh contour.

DESCRIPTION

The `Q3Mesh_GetContourNumVertices` function returns, in the `numVertices` parameter, the number of vertices that compose the mesh contour specified by the `mesh` and `contour` parameters.

Q3Mesh_GetCornerAttributeSet

You can use the `Q3Mesh_GetCornerAttributeSet` function to get the attribute set of a mesh corner.

```
TQ3Status Q3Mesh_GetCornerAttributeSet (
                    TQ3GeometryObject mesh,
                    TQ3MeshVertex vertex,
                    TQ3MeshFace face,
                    TQ3AttributeSet *attributeSet);
```

`mesh` A mesh.

`vertex` A mesh vertex.

`face` A mesh face. This face must contain the specified vertex in one of its contours.

`attributeSet`
On exit, the set of attributes for the corner defined by the specified mesh vertex and face.

DESCRIPTION

The `Q3Mesh_GetCornerAttributeSet` function returns, in the `attributeSet` parameter, the set of attributes of the corner defined by the `vertex` and `face` parameters in the mesh specified by the `mesh` parameter. The corner attributes override any attributes associated with the vertex alone. The reference count of the set is incremented.

Q3Mesh_SetCornerAttributeSet

You can use the `Q3Mesh_SetCornerAttributeSet` function to set the attribute set of a mesh corner.

```
TQ3Status Q3Mesh_SetCornerAttributeSet (
                    TQ3GeometryObject mesh,
                    TQ3MeshVertex vertex,
                    TQ3MeshFace face,
                    TQ3AttributeSet attributeSet);
```

`mesh` A mesh.

`vertex` A mesh vertex.

`face` A mesh face. This face must contain the specified vertex in one of its contours.

`attributeSet` The desired set of attributes for the corner defined by the specified mesh vertex and face.

DESCRIPTION

The `Q3Mesh_SetCornerAttributeSet` function sets the attribute set of the corner defined by the `vertex` and `face` parameters in the mesh specified by the `mesh` parameter to the set of attributes specified by the `attributeSet` parameter. The corner attributes override any attributes associated with the vertex alone.

Traversing Mesh Components, Vertices, Faces, and Edges

QuickDraw 3D provides a large number of functions that you can use to iterate through the components, vertices, faces, or edges of a mesh. For example, you can call the `Q3Mesh_FirstMeshComponent` function to get the first component in a mesh; then you can call the `Q3Mesh_NextMeshComponent` function to get any subsequent mesh components.

For even simpler mesh traversal, QuickDraw 3D defines a large number of macros modeled on the standard C language `for` statement. For example, the `Q3ForEachMeshComponent` macro uses the `Q3Mesh_FirstMeshComponent` function and the `Q3Mesh_NextMeshComponent` function to iterate through all the components of a mesh.

IMPORTANT

Adding or deleting vertices or faces within the scope of these iterators might produce unpredictable results. ▲

```
#define Q3ForEachMeshComponent(m,c,i)                                   \
    for ( (c) = Q3Mesh_FirstMeshComponent((m),(i));                     \
        (c);                                                            \
        (c) = Q3Mesh_NextMeshComponent((i)) )

#define Q3ForEachComponentVertex(c,v,i)                                 \
    for ( (v) = Q3Mesh_FirstComponentVertex((c),(i));                   \
        (v);                                                            \
        (v) = Q3Mesh_NextComponentVertex((i)) )
```

```
#define Q3ForEachComponentEdge(c,e,i)                                   \
    for ( (e) = Q3Mesh_FirstComponentEdge((c),(i));                     \
        (e);                                                            \
        (e) = Q3Mesh_NextComponentEdge((i)) )

#define Q3ForEachMeshVertex(m,v,i)                                      \
    for ( (v) = Q3Mesh_FirstMeshVertex((m),(i));                        \
        (v);                                                            \
        (v) = Q3Mesh_NextMeshVertex((i)) )

#define Q3ForEachMeshFace(m,f,i)                                        \
    for ( (f) = Q3Mesh_FirstMeshFace((m),(i));                          \
        (f);                                                            \
        (f) = Q3Mesh_NextMeshFace((i)) )

#define Q3ForEachMeshEdge(m,e,i)                                        \
    for ( (e) = Q3Mesh_FirstMeshEdge((m),(i));                          \
        (e);                                                            \
        (e) = Q3Mesh_NextMeshEdge((i)) )

#define Q3ForEachVertexEdge(v,e,i)                                      \
    for ( (e) = Q3Mesh_FirstVertexEdge((v),(i));                        \
        (e);                                                            \
        (e) = Q3Mesh_NextVertexEdge((i)) )

#define Q3ForEachVertexVertex(v,n,i)                                    \
    for ( (n) = Q3Mesh_FirstVertexVertex((v),(i));                      \
        (n);                                                            \
        (n) = Q3Mesh_NextVertexVertex((i)) )

#define Q3ForEachVertexFace(v,f,i)                                      \
    for ( (f) = Q3Mesh_FirstVertexFace((v),(i));                        \
        (f);                                                            \
        (f) = Q3Mesh_NextVertexFace((i)) )

#define Q3ForEachFaceEdge(f,e,i)                                        \
    for ( (e) = Q3Mesh_FirstFaceEdge((f),(i));                          \
        (e);                                                            \
        (e) = Q3Mesh_NextFaceEdge((i)) )
```

```
#define Q3ForEachFaceVertex(f,v,i)                                      \
    for ( (v) = Q3Mesh_FirstFaceVertex((f),(i));                        \
        (v);                                                            \
        (v) = Q3Mesh_NextFaceVertex((i)) )

#define Q3ForEachFaceFace(f,n,i)                                        \
    for ( (n) = Q3Mesh_FirstFaceFace((f),(i));                          \
        (n);                                                            \
        (n) = Q3Mesh_NextFaceFace((i)) )

#define Q3ForEachFaceContour(f,h,i)                                     \
    for ( (h) = Q3Mesh_FirstFaceContour((f),(i));                       \
        (h);                                                            \
        (h) = Q3Mesh_NextFaceContour((i)) )

#define Q3ForEachContourEdge(h,e,i)                                     \
    for ( (e) = Q3Mesh_FirstContourEdge((h),(i));                       \
        (e);                                                            \
        (e) = Q3Mesh_NextContourEdge((i)) )

#define Q3ForEachContourVertex(h,v,i)                                   \
    for ( (v) = Q3Mesh_FirstContourVertex((h),(i));                     \
        (v);                                                            \
        (v) = Q3Mesh_NextContourVertex((i)) )

#define Q3ForEachContourFace(h,f,i)                                     \
    for ( (f) = Q3Mesh_FirstContourFace((h),(i));                       \
        (f);                                                            \
        (f) = Q3Mesh_NextContourFace((i)) )
```

Q3Mesh_FirstMeshComponent

You can use the `Q3Mesh_FirstMeshComponent` function to get the first component of a mesh.

```
TQ3MeshComponent Q3Mesh_FirstMeshComponent (
                    TQ3GeometryObject mesh,
                    TQ3MeshIterator *iterator);
```

`mesh`	A mesh.
`iterator`	A pointer to a mesh iterator structure.

DESCRIPTION

The `Q3Mesh_FirstMeshComponent` function returns, as its function result, the first mesh component in the mesh specified by the `mesh` parameter. The `iterator` parameter is a pointer to a mesh iterator structure that `Q3Mesh_FirstMeshComponent` fills in before returning. You should pass the address of that structure to the `Q3Mesh_NextMeshComponent` function.

SPECIAL CONSIDERATIONS

The `Q3Mesh_FirstMeshComponent` function might not accurately report the first mesh component in a mesh if called while mesh updating is delayed (that is, after a call to `Q3Mesh_DelayUpdates` but before the matching call to `Q3Mesh_ResumeUpdates`).

Q3Mesh_NextMeshComponent

You can use the `Q3Mesh_NextMeshComponent` function to get the next component in a mesh.

```
TQ3MeshComponent Q3Mesh_NextMeshComponent (
                    TQ3MeshIterator *iterator);
```

`iterator`	A pointer to a mesh iterator structure.

DESCRIPTION

The `Q3Mesh_NextMeshComponent` function returns, as its function result, the next mesh component in the iteration specified by the `iterator` parameter, which must have been filled in by a previous call to `Q3Mesh_FirstMeshComponent` or `Q3Mesh_NextMeshComponent`. If there are no more mesh components, this function returns `NULL`.

SPECIAL CONSIDERATIONS

The `Q3Mesh_NextMeshComponent` function might not accurately report the next mesh component in a mesh if called while mesh updating is delayed (that is, after a call to `Q3Mesh_DelayUpdates` but before the matching call to `Q3Mesh_ResumeUpdates`).

Q3Mesh_FirstComponentVertex

You can use the `Q3Mesh_FirstComponentVertex` function to get the first vertex in a mesh component.

```
TQ3MeshVertex Q3Mesh_FirstComponentVertex (
                    TQ3MeshComponent component,
                    TQ3MeshIterator *iterator);
```

`component` A mesh component.

`iterator` A pointer to a mesh iterator structure.

DESCRIPTION

The `Q3Mesh_FirstComponentVertex` function returns, as its function result, the first vertex in the mesh component specified by the `component` parameter. The `iterator` parameter is a pointer to a mesh iterator structure that `Q3Mesh_FirstComponentVertex` fills in before returning. You should pass the address of that structure to the `Q3Mesh_NextComponentVertex` function.

SPECIAL CONSIDERATIONS

The `Q3Mesh_FirstComponentVertex` function might not accurately report the first vertex in a mesh component if called while mesh updating is delayed (that is, after a call to `Q3Mesh_DelayUpdates` but before the matching call to `Q3Mesh_ResumeUpdates`).

Q3Mesh_NextComponentVertex

You can use the `Q3Mesh_NextComponentVertex` function to get the next vertex in a mesh component.

```
TQ3MeshVertex Q3Mesh_NextComponentVertex (
                    TQ3MeshIterator *iterator);
```

`iterator` A pointer to a mesh iterator structure.

DESCRIPTION

The `Q3Mesh_NextComponentVertex` function returns, as its function result, the next vertex in the iteration specified by the `iterator` parameter, which must have been filled in by a previous call to `Q3Mesh_FirstComponentVertex` or `Q3Mesh_NextComponentVertex`. If there are no more vertices, this function returns `NULL`.

SPECIAL CONSIDERATIONS

The `Q3Mesh_NextComponentVertex` function might not accurately report the next vertex in a mesh component if called while mesh updating is delayed (that is, after a call to `Q3Mesh_DelayUpdates` but before the matching call to `Q3Mesh_ResumeUpdates`).

Q3Mesh_FirstComponentEdge

You can use the `Q3Mesh_FirstComponentEdge` function to get the first edge in a mesh component.

```
TQ3MeshEdge Q3Mesh_FirstComponentEdge (
                    TQ3MeshComponent component,
                    TQ3MeshIterator *iterator);
```

`component` A mesh component.

`iterator` A pointer to a mesh iterator structure.

DESCRIPTION

The `Q3Mesh_FirstComponentEdge` function returns, as its function result, the first edge in the mesh component specified by the `component` parameter. The `iterator` parameter is a pointer to a mesh iterator structure that `Q3Mesh_FirstComponentEdge` fills in before returning. You should pass the address of that structure to the `Q3Mesh_NextComponentEdge` function.

SPECIAL CONSIDERATIONS

The `Q3Mesh_FirstComponentEdge` function might not accurately report the first edge in a mesh component if called while mesh updating is delayed (that is, after a call to `Q3Mesh_DelayUpdates` but before the matching call to `Q3Mesh_ResumeUpdates`).

Q3Mesh_NextComponentEdge

You can use the `Q3Mesh_NextComponentEdge` function to get the next edge in a mesh component.

```
TQ3MeshEdge Q3Mesh_NextComponentEdge (TQ3MeshIterator *iterator);
```

`iterator` A pointer to a mesh iterator structure.

DESCRIPTION

The `Q3Mesh_NextComponentEdge` function returns, as its function result, the next edge in the iteration specified by the `iterator` parameter, which must have been filled in by a previous call to `Q3Mesh_FirstComponentEdge` or `Q3Mesh_NextComponentEdge`. If there are no more edges, this function returns `NULL`.

SPECIAL CONSIDERATIONS

The `Q3Mesh_NextComponentEdge` function might not accurately report the next edge in a mesh component if called while mesh updating is delayed (that is, after a call to `Q3Mesh_DelayUpdates` but before the matching call to `Q3Mesh_ResumeUpdates`).

Q3Mesh_FirstMeshVertex

You can use the `Q3Mesh_FirstMeshVertex` function to get the first vertex in a mesh.

```
TQ3MeshVertex Q3Mesh_FirstMeshVertex (
                    TQ3GeometryObject mesh,
                    TQ3MeshIterator *iterator);
```

`mesh` A mesh.

`iterator` A pointer to a mesh iterator structure.

DESCRIPTION

The `Q3Mesh_FirstMeshVertex` function returns, as its function result, the first vertex in the mesh specified by the `mesh` parameter. The `iterator` parameter is a pointer to a mesh iterator structure that `Q3Mesh_FirstMeshVertex` fills in before returning. You should pass the address of that structure to the `Q3Mesh_NextMeshVertex` function.

Q3Mesh_NextMeshVertex

You can use the `Q3Mesh_NextMeshVertex` function to get the next vertex in a mesh.

```
TQ3MeshVertex Q3Mesh_NextMeshVertex (TQ3MeshIterator *iterator);
```

`iterator` A pointer to a mesh iterator structure.

DESCRIPTION

The `Q3Mesh_NextMeshVertex` function returns, as its function result, the next vertex in the iteration specified by the `iterator` parameter, which must have been filled in by a previous call to `Q3Mesh_FirstMeshVertex` or `Q3Mesh_NextMeshVertex`. If there are no more vertices, this function returns `NULL`.

Q3Mesh_FirstMeshFace

You can use the `Q3Mesh_FirstMeshFace` function to get the first face in a mesh.

```
TQ3MeshFace Q3Mesh_FirstMeshFace (
                    TQ3GeometryObject mesh,
                    TQ3MeshIterator *iterator);
```

`mesh` A mesh.

`iterator` A pointer to a mesh iterator structure.

DESCRIPTION

The `Q3Mesh_FirstMeshFace` function returns, as its function result, the first face in the mesh specified by the `mesh` parameter. The `iterator` parameter is a pointer to a mesh iterator structure that `Q3Mesh_FirstMeshFace` fills in before returning. You should pass the address of that structure to the `Q3Mesh_NextMeshFace` function.

Q3Mesh_NextMeshFace

You can use the `Q3Mesh_NextMeshFace` function to get the next face in a mesh.

```
TQ3MeshFace Q3Mesh_NextMeshFace (TQ3MeshIterator *iterator);
```

`iterator` A pointer to a mesh iterator structure.

DESCRIPTION

The `Q3Mesh_NextMeshFace` function returns, as its function result, the next face in the iteration specified by the `iterator` parameter, which must have been filled in by a previous call to `Q3Mesh_FirstMeshFace` or `Q3Mesh_NextMeshFace`. If there are no more faces, this function returns `NULL`.

Q3Mesh_FirstMeshEdge

You can use the `Q3Mesh_FirstMeshEdge` function to get the first edge in a mesh.

```
TQ3MeshEdge Q3Mesh_FirstMeshEdge (
                    TQ3GeometryObject mesh,
                    TQ3MeshIterator *iterator);
```

`mesh` A mesh.

`iterator` A pointer to a mesh iterator structure.

DESCRIPTION

The `Q3Mesh_FirstMeshEdge` function returns, as its function result, the first edge in the mesh specified by the `mesh` parameter. The `iterator` parameter is a pointer to a mesh iterator structure that `Q3Mesh_FirstMeshEdge` fills in before returning. You should pass the address of that structure to the `Q3Mesh_NextMeshEdge` function.

Q3Mesh_NextMeshEdge

You can use the `Q3Mesh_NextMeshEdge` function to get the next edge in a mesh.

```
TQ3MeshEdge Q3Mesh_NextMeshEdge (TQ3MeshIterator *iterator);
```

`iterator` A pointer to a mesh iterator structure.

DESCRIPTION

The `Q3Mesh_NextMeshEdge` function returns, as its function result, the next edge in the iteration specified by the `iterator` parameter, which must have been filled in by a previous call to `Q3Mesh_FirstMeshEdge` or `Q3Mesh_NextMeshEdge`. If there are no more edges, this function returns `NULL`.

Q3Mesh_FirstVertexEdge

You can use the `Q3Mesh_FirstVertexEdge` function to get the first edge around a vertex.

```
TQ3MeshEdge Q3Mesh_FirstVertexEdge (
                    TQ3MeshVertex vertex,
                    TQ3MeshIterator *iterator);
```

`vertex` A mesh vertex.

`iterator` A pointer to a mesh iterator structure.

DESCRIPTION

The `Q3Mesh_FirstVertexEdge` function returns, as its function result, the first edge around the vertex specified by the `vertex` parameter, in a counterclockwise ordering. The `iterator` parameter is a pointer to a mesh iterator structure that `Q3Mesh_FirstVertexEdge` fills in before returning. You should pass the address of that structure to the `Q3Mesh_NextVertexEdge` function.

Q3Mesh_NextVertexEdge

You can use the `Q3Mesh_NextVertexEdge` function to get the next edge around a vertex, in a counterclockwise order.

```
TQ3MeshEdge Q3Mesh_NextVertexEdge (TQ3MeshIterator *iterator);
```

`iterator` A pointer to a mesh iterator structure.

DESCRIPTION

The `Q3Mesh_NextVertexEdge` function returns, as its function result, the next edge counterclockwise in the iteration specified by the `iterator` parameter, which must have been filled in by a previous call to `Q3Mesh_FirstVertexEdge` or `Q3Mesh_NextVertexEdge`. If there are no more edges, this function returns `NULL`.

Q3Mesh_FirstVertexVertex

You can use the `Q3Mesh_FirstVertexVertex` function to get the first vertex connected to a vertex by an edge.

```
TQ3MeshVertex Q3Mesh_FirstVertexVertex (
                    TQ3MeshVertex vertex,
                    TQ3MeshIterator *iterator);
```

`vertex` A mesh vertex.

`iterator` A pointer to a mesh iterator structure.

DESCRIPTION

The `Q3Mesh_FirstVertexVertex` function returns, as its function result, the first vertex neighboring the vertex specified by the `vertex` parameter. The `iterator` parameter is a pointer to a mesh iterator structure that `Q3Mesh_FirstVertexVertex` fills in before returning. You should pass the address of that structure to the `Q3Mesh_NextVertexVertex` function.

Q3Mesh_NextVertexVertex

You can use the `Q3Mesh_NextVertexVertex` function to get the next vertex connected to a vertex by an edge, in a counterclockwise order.

```
TQ3MeshVertex Q3Mesh_NextVertexVertex (TQ3MeshIterator *iterator);
```

`iterator` A pointer to a mesh iterator structure.

DESCRIPTION

The `Q3Mesh_NextVertexVertex` function returns, as its function result, the next vertex counterclockwise in the iteration specified by the `iterator` parameter, which must have been filled in by a previous call to `Q3Mesh_FirstVertexVertex` or `Q3Mesh_NextVertexVertex`. If there are no more vertices, this function returns `NULL`.

Q3Mesh_FirstVertexFace

You can use the `Q3Mesh_FirstVertexFace` function to get the first face around a vertex.

```
TQ3MeshFace Q3Mesh_FirstVertexFace (
                    TQ3MeshVertex vertex,
                    TQ3MeshIterator *iterator);
```

`vertex` A mesh vertex.

`iterator` A pointer to a mesh iterator structure.

DESCRIPTION

The `Q3Mesh_FirstVertexFace` function returns, as its function result, the first face around the vertex specified by the `vertex` parameter. The `iterator` parameter is a pointer to a mesh iterator structure that `Q3Mesh_FirstVertexFace` fills in before returning. You should pass the address of that structure to the `Q3Mesh_NextVertexVertex` function.

Q3Mesh_NextVertexFace

You can use the `Q3Mesh_NextVertexFace` function to get the next face around a vertex, in a counterclockwise order.

```
TQ3MeshFace Q3Mesh_NextVertexFace (TQ3MeshIterator *iterator);
```

`iterator` A pointer to a mesh iterator structure.

DESCRIPTION

The `Q3Mesh_NextVertexFace` function returns, as its function result, the next face counterclockwise in the iteration specified by the `iterator` parameter, which must have been filled in by a previous call to `Q3Mesh_FirstVertexFace` or `Q3Mesh_NextVertexFace`. If there are no more faces, this function returns `NULL`.

Q3Mesh_FirstFaceEdge

You can use the `Q3Mesh_FirstFaceEdge` function to get the first edge of a mesh face.

```
TQ3MeshEdge Q3Mesh_FirstFaceEdge (
                    TQ3MeshFace face,
                    TQ3MeshIterator *iterator);
```

`face` A mesh face.

`iterator` A pointer to a mesh iterator structure.

DESCRIPTION

The `Q3Mesh_FirstFaceEdge` function returns, as its function result, the first edge of the face specified by the `face` parameter. The `iterator` parameter is a pointer to a mesh iterator structure that `Q3Mesh_FirstFaceEdge` fills in before returning. You should pass the address of that structure to the `Q3Mesh_NextFaceEdge` function.

Q3Mesh_NextFaceEdge

You can use the `Q3Mesh_NextFaceEdge` function to get the next edge of a mesh face, in a counterclockwise order.

```
TQ3MeshEdge Q3Mesh_NextFaceEdge (TQ3MeshIterator *iterator);
```

`iterator` A pointer to a mesh iterator structure.

DESCRIPTION

The `Q3Mesh_NextFaceEdge` function returns, as its function result, the next edge counterclockwise in the iteration specified by the `iterator` parameter, which must have been filled in by a previous call to `Q3Mesh_FirstFaceEdge` or `Q3Mesh_NextFaceEdge`. If there are no more edges, this function returns `NULL`. This function iterates over all the contours in the face.

Q3Mesh_FirstFaceVertex

You can use the `Q3Mesh_FirstFaceVertex` function to get the first vertex of a mesh face.

```
TQ3MeshVertex Q3Mesh_FirstFaceVertex (
                    TQ3MeshFace face,
                    TQ3MeshIterator *iterator);
```

`face` A mesh face.

`iterator` A pointer to a mesh iterator structure.

DESCRIPTION

The `Q3Mesh_FirstFaceVertex` function returns, as its function result, the first vertex of the face specified by the `face` parameter. The `iterator` parameter is a pointer to a mesh iterator structure that `Q3Mesh_FirstFaceVertex` fills in before returning. You should pass the address of that structure to the `Q3Mesh_NextFaceVertex` function.

Q3Mesh_NextFaceVertex

You can use the `Q3Mesh_NextFaceVertex` function to get the next vertex of a mesh face, in a counterclockwise order.

```
TQ3MeshVertex Q3Mesh_NextFaceVertex (TQ3MeshIterator *iterator);
```

`iterator` A pointer to a mesh iterator structure.

DESCRIPTION

The `Q3Mesh_NextFaceVertex` function returns, as its function result, the next vertex counterclockwise in the iteration specified by the `iterator` parameter, which must have been filled in by a previous call to `Q3Mesh_FirstFaceVertex` or `Q3Mesh_NextFaceVertex`. If there are no more vertices, this function returns `NULL`. This function iterates over all the contours in the face.

Q3Mesh_FirstFaceFace

You can use the `Q3Mesh_FirstFaceFace` function to get the first face surrounding a mesh face.

```
TQ3MeshFace Q3Mesh_FirstFaceFace (
                    TQ3MeshFace face,
                    TQ3MeshIterator *iterator);
```

`face` A mesh face.

`iterator` A pointer to a mesh iterator structure.

DESCRIPTION

The `Q3Mesh_FirstFaceFace` function returns, as its function result, the first face surrounding the face specified by the `face` parameter. The `iterator` parameter is a pointer to a mesh iterator structure that `Q3Mesh_FirstFaceFace` fills in before returning. You should pass the address of that structure to the `Q3Mesh_NextFaceFace` function.

Q3Mesh_NextFaceFace

You can use the `Q3Mesh_NextFaceFace` function to get the next face surrounding a mesh face, in a counterclockwise order.

```
TQ3MeshFace Q3Mesh_NextFaceFace (TQ3MeshIterator *iterator);
```

`iterator` A pointer to a mesh iterator structure.

DESCRIPTION

The `Q3Mesh_NextFaceFace` function returns, as its function result, the next face counterclockwise in the iteration specified by the `iterator` parameter, which must have been filled in by a previous call to `Q3Mesh_FirstFaceFace` or `Q3Mesh_NextFaceFace`. If there are no more faces, this function returns `NULL`.

Q3Mesh_FirstFaceContour

You can use the `Q3Mesh_FirstFaceContour` function to get the first contour of a mesh face.

```
TQ3MeshContour Q3Mesh_FirstFaceContour (
                    TQ3MeshFace face,
                    TQ3MeshIterator *iterator);
```

`face` A mesh face.

`iterator` A pointer to a mesh iterator structure.

DESCRIPTION

The `Q3Mesh_FirstFaceContour` function returns, as its function result, the first contour of the face specified by the `face` parameter. The `iterator` parameter is a pointer to a mesh iterator structure that `Q3Mesh_FirstFaceContour` fills in before returning. You should pass the address of that structure to the `Q3Mesh_NextFaceContour` function.

Q3Mesh_NextFaceContour

You can use the `Q3Mesh_NextFaceContour` function to get the next contour of a mesh face.

```
TQ3MeshContour Q3Mesh_NextFaceContour (
                    TQ3MeshIterator *iterator);
```

`iterator` A pointer to a mesh iterator structure.

DESCRIPTION

The `Q3Mesh_NextFaceContour` function returns, as its function result, the next contour in the iteration specified by the `iterator` parameter, which must have been filled in by a previous call to `Q3Mesh_FirstFaceContour` or `Q3Mesh_NextFaceContour`. If there are no more contours, this function returns `NULL`.

Q3Mesh_FirstContourEdge

You can use the `Q3Mesh_FirstContourEdge` function to get the first edge of a mesh contour.

```
TQ3MeshEdge Q3Mesh_FirstContourEdge (
                    TQ3MeshContour contour,
                    TQ3MeshIterator *iterator);
```

`contour` A mesh contour.

`iterator` A pointer to a mesh iterator structure.

DESCRIPTION

The `Q3Mesh_FirstContourEdge` function returns, as its function result, the first edge of the mesh contour specified by the `contour` parameter. The `iterator` parameter is a pointer to a mesh iterator structure that `Q3Mesh_FirstContourEdge` fills in before returning. You should pass the address of that structure to the `Q3Mesh_NextContourEdge` function.

Q3Mesh_NextContourEdge

You can use the `Q3Mesh_NextContourEdge` function to get the next edge of a mesh contour, in a counterclockwise order.

```
TQ3MeshEdge Q3Mesh_NextContourEdge (TQ3MeshIterator *iterator);
```

`iterator` A pointer to a mesh iterator structure.

DESCRIPTION

The `Q3Mesh_NextContourEdge` function returns, as its function result, the next edge counterclockwise in the iteration specified by the `iterator` parameter, which must have been filled in by a previous call to `Q3Mesh_FirstContourEdge` or `Q3Mesh_NextContourEdge`. If there are no more edges, this function returns `NULL`.

Q3Mesh_FirstContourVertex

You can use the `Q3Mesh_FirstContourVertex` function to get the first vertex of a mesh contour.

```
TQ3MeshVertex Q3Mesh_FirstContourVertex (
                    TQ3MeshContour contour,
                    TQ3MeshIterator *iterator);
```

`contour` A mesh contour.

`iterator` A pointer to a mesh iterator structure.

DESCRIPTION

The `Q3Mesh_FirstContourVertex` function returns, as its function result, the first vertex of the mesh contour specified by the `contour` parameter. The `iterator` parameter is a pointer to a mesh iterator structure that `Q3Mesh_FirstContourVertex` fills in before returning. You should pass the address of that structure to the `Q3Mesh_NextContourVertex` function.

Q3Mesh_NextContourVertex

You can use the `Q3Mesh_NextContourVertex` function to get the next vertex of a mesh contour, in a counterclockwise order.

```
TQ3MeshVertex Q3Mesh_NextContourVertex (
                    TQ3MeshIterator *iterator);
```

`iterator` A pointer to a mesh iterator structure.

DESCRIPTION

The `Q3Mesh_NextContourVertex` function returns, as its function result, the next vertex in the iteration specified by the `iterator` parameter, which must have been filled in by a previous call to `Q3Mesh_FirstContourVertex` or `Q3Mesh_NextContourVertex`. If there are no more vertices, this function returns `NULL`.

Q3Mesh_FirstContourFace

You can use the `Q3Mesh_FirstContourFace` function to get the first face surrounding a mesh contour.

```
TQ3MeshFace Q3Mesh_FirstContourFace (
                    TQ3MeshContour contour,
                    TQ3MeshIterator *iterator);
```

`contour` A mesh contour.

`iterator` A pointer to a mesh iterator structure.

DESCRIPTION

The `Q3Mesh_FirstContourFace` function returns, as its function result, the first face of the mesh contour specified by the `contour` parameter. The `iterator` parameter is a pointer to a mesh iterator structure that `Q3Mesh_FirstContourFace` fills in before returning. You should pass the address of that structure to the `Q3Mesh_NextContourFace` function.

Q3Mesh_NextContourFace

You can use the `Q3Mesh_NextContourFace` function to get the next face surrounding a mesh contour, in a counterclockwise order.

```
TQ3MeshFace Q3Mesh_NextContourFace (TQ3MeshIterator *iterator);
```

`iterator` A pointer to a mesh iterator structure.

DESCRIPTION

The `Q3Mesh_NextContourFace` function returns, as its function result, the next face counterclockwise in the iteration specified by the `iterator` parameter, which must have been filled in by a previous call to `Q3Mesh_FirstContourFace` or `Q3Mesh_NextContourFace`. If there are no more faces, this function returns `NULL`.

Creating and Editing Trimeshes

QuickDraw 3D provides routines that you can use to create and manipulate trimeshes. See “Trimeshes” (page 307) for the definition of a trimesh.

Q3TriMesh_New

You can use the `Q3TriMesh_New` function to create a new trimesh.

```
TQ3GeometryObject Q3TriMesh_New (const TQ3TriMeshData *triMeshData);
```

`triMeshData` A pointer to a `TQ3TriMeshData` structure.

DESCRIPTION

The `Q3TriMesh_New` function returns, as its function result, a new trimesh having the shape and attributes specified by the `triMeshData` parameter. If a new trimesh could not be created, `Q3TriMesh_New` returns the value `NULL`.

Q3TriMesh_Submit

You can use the `Q3TriMesh_Submit` function to submit an immediate trimesh for drawing, picking, bounding, or writing.

```
TQ3Status Q3TriMesh_Submit (
                    const TQ3TriMeshData *triMeshData,
                    TQ3ViewObject view);
```

`triMeshData` A pointer to a `TQ3TriMeshData` structure.

`view` A view.

DESCRIPTION

The `Q3TriMesh_Submit` function submits for drawing, picking, bounding, or writing the immediate trimesh whose shape and attribute set are specified by

the `triMeshData` parameter. The trimesh is drawn, picked, bounded, or written according to the view characteristics specified in the `view` parameter.

SPECIAL CONSIDERATIONS

You should call this function only in a submitting loop.

Q3TriMesh_GetData

You can use the `Q3TriMesh_GetData` function to get the data that defines a trimesh and its attributes.

```
TQ3Status Q3TriMesh_GetData (
                    TQ3GeometryObject triMesh,
                    TQ3TriMeshData *triMeshData);
```

`triMesh` A trimesh.

`triMeshData` On exit, a pointer to a `TQ3TriMeshData` structure that contains information about the trimesh specified by the `triMesh` parameter.

DESCRIPTION

The `Q3TriMesh_GetData` function returns, through the `triMeshData` parameter, information about the trimesh specified by the `triMesh` parameter. QuickDraw 3D allocates memory for the `TQ3TriMeshData` structure internally; you must call `Q3TriMesh_EmptyData` to dispose of that memory.

Q3TriMesh_SetData

You can use the `Q3TriMesh_SetData` function to set the data that defines a trimesh and its attributes.

```
TQ3Status Q3TriMesh_SetData (
                    TQ3GeometryObject triMesh,
                    const TQ3TriMeshData *triMeshData);
```

`triMesh` A trimesh.

`triMeshData` A pointer to a `TQ3TriMeshData` structure.

DESCRIPTION

The `Q3TriMesh_SetData` function sets the data associated with the trimesh specified by the `triMesh` parameter to the data specified by the `triMeshData` parameter.

Q3TriMesh_EmptyData

You can use the `Q3TriMesh_EmptyData` function to release the memory occupied by the data structure returned by a previous call to `Q3TriMesh_GetData`.

```
TQ3Status Q3TriMesh_EmptyData (TQ3TriMeshData *triMeshData);
```

`triMeshData` A pointer to a `TQ3TriMeshData` structure.

DESCRIPTION

The `Q3TriMesh_EmptyData` function releases the memory occupied by the `TQ3TriMeshData` structure pointed to by the `triMeshData` parameter; that memory was allocated by a previous call to `Q3TriMesh_GetData`.

Creating and Editing Polyhedra

QuickDraw 3D provides routines that you can use to create and manipulate polyhedra. See “Polyhedra” (page 311) for the definition of a polyhedron.

Q3Polyhedron_New

You can use the `Q3Polyhedron_New` function to create a new polyhedron.

```
TQ3GeometryObject Q3Polyhedron_New (
                    const TQ3PolyhedronData *polyhedronData);
```

`polyhedronData`
A pointer to a `TQ3PolyhedronData` structure.

DESCRIPTION

The `Q3Polyhedron_New` function returns, as its function result, a new polyhedron having the shape and attributes specified by the `polyhedronData` parameter. If a new polyhedron could not be created, `Q3Polyhedron_New` returns the value `NULL`.

Q3Polyhedron_Submit

You can use the `Q3Polyhedron_Submit` function to submit an immediate polyhedron for drawing, picking, bounding, or writing.

```
TQ3Status Q3Polyhedron_Submit (
                    const TQ3PolyhedronData *polyhedronData,
                    TQ3ViewObject view);
```

`polyhedronData`
A pointer to a `TQ3PolyhedronData` structure.

`view` A view.

DESCRIPTION

The `Q3Polyhedron_Submit` function submits for drawing, picking, bounding, or writing the immediate polyhedron whose shape and attribute set are specified by the `polyhedronData` parameter. The polyhedron is drawn, picked, bounded, or written according to the view characteristics specified in the `view` parameter.

SPECIAL CONSIDERATIONS

You should call this function only in a submitting loop.

Q3Polyhedron_GetData

You can use the `Q3Polyhedron_GetData` function to get the data that defines a polyhedron and its attributes.

```
TQ3Status Q3Polyhedron_GetData (
                    TQ3GeometryObject polyhedron,
                    TQ3PolyhedronData *polyhedronData);
```

`polyhedron` A polyhedron.

`polyhedronData`
On exit, a pointer to a `TQ3PolyhedronData` structure that contains information about the polyhedron specified by the `polyhedron` parameter.

DESCRIPTION

The `Q3Polyhedron_GetData` function returns, through the `polyhedronData` parameter, information about the polyhedron specified by the `polyhedron` parameter. QuickDraw 3D allocates memory for the `TQ3PolyhedronData` structure internally; you must call `Q3Polyhedron_EmptyData` to dispose of that memory.

Q3Polyhedron_SetData

You can use the `Q3Polyhedron_SetData` function to set the data that defines a polyhedron and its attributes.

```
TQ3Status Q3Polyhedron_SetData (
                    TQ3GeometryObject polyhedron,
                    const TQ3PolyhedronData *polyhedronData);
```

`polyhedron` A polyhedron.

`polyhedronData`
A pointer to a `TQ3PolyhedronData` structure.

DESCRIPTION

The `Q3Polyhedron_SetData` function sets the data associated with the polyhedron specified by the `polyhedron` parameter to the data specified by the `polyhedronData` parameter.

Q3Polyhedron_EmptyData

You can use the `Q3Polyhedron_EmptyData` function to release the memory occupied by the data structure returned by a previous call to `Q3Polyhedron_GetData`.

```
TQ3Status Q3Polyhedron_EmptyData (TQ3PolyhedronData *polyhedronData);
```

`polyhedronData`
A pointer to a `TQ3PolyhedronData` structure.

DESCRIPTION

The `Q3Polyhedron_EmptyData` function releases the memory occupied by the `TQ3PolyhedronData` structure pointed to by the `polyhedronData` parameter; that memory was allocated by a previous call to `Q3Polyhedron_GetData`.

Q3Polyhedron_GetVertexPosition

You can use the `Q3Polyhedron_GetVertexPosition` function to get the position of a vertex of a polyhedron.

```
TQ3Status Q3Polyhedron_GetVertexPosition (
                    TQ3GeometryObject    polyhedron,
                    unsigned long        index,
                    TQ3Point3D           *point);
```

`polyhedron` A polyhedron.

`index` An index into an array of three-dimensional points.

`point` A pointer to the array of points in the polyhedron.

DESCRIPTION

The `Q3Polyhedron_GetVertexPosition` function returns, in the `point` parameter, the position of the vertex having the index specified by the `index` parameter in the `vertices` array of the polyhedron specified by the `polyhedron` parameter.

Q3Polyhedron_SetVertexPosition

You can use the `Q3Polyhedron_SetVertexPosition` function to set the position of a vertex of a polyhedron.

```
TQ3Status Q3Polyhedron_SetVertexPosition (
                    TQ3GeometryObject    polyhedron,
                    unsigned long        index,
                    const TQ3Point3D     *point);
```

`polyhedron` A polyhedron.

`index` An index into an array of three-dimensional points.

`point` A pointer to the array of points in the polyhedron.

DESCRIPTION

The `Q3Polyhedron_SetVertexPosition` function sets the position of the vertex having the index specified by the `index` parameter in the `vertices` array of the polyhedron specified by the `polyhedron` parameter to that specified in the `point` parameter.

Q3Polyhedron_GetVertexAttributeSet

You can use the `Q3Polyhedron_GetVertexAttributeSet` function to get the attribute set of a vertex of a polyhedron.

```
TQ3Status Q3Polyhedron_GetVertexAttributeSet (
                    TQ3GeometryObject    polyhedron,
                    unsigned long        index,
                    TQ3AttributeSet      *attributeSet);
```

`polyhedron` A polyhedron.

`index` An index into an array of three-dimensional points.

`attributeSet` On exit, a pointer to a vertex attribute set.

DESCRIPTION

The `Q3Polyhedron_GetVertexAttributeSet` function returns, in the `attributeSet` parameter, the set of attributes for the vertex having the index specified by the `index` parameter in the `vertices` array of the polyhedron specified by the `polyhedron` parameter. The reference count of the set is incremented.

Q3Polyhedron_SetVertexAttributeSet

You can use the `Q3Polyhedron_SetVertexAttributeSet` function to set the attribute set of a vertex of a polyhedron.

```
TQ3Status Q3Polyhedron_SetVertexAttributeSet (
                    TQ3GeometryObject    polyhedron,
                    unsigned long        index,
                    TQ3AttributeSet      attributeSet);
```

`polyhedron` A polyhedron.

`index` An index into an array of three-dimensional points.

`attributeSet` On exit, a pointer to a vertex attribute set.

DESCRIPTION

The `Q3Polyhedron_SetVertexAttributeSet` function sets the attribute set of the vertex having the index specified by the `index` parameter in the `vertices` array of the polyhedron specified by the `polyhedron` parameter to the set specified in the `attributeSet` parameter.

Q3Polyhedron_GetTriangleData

You can use the `Q3Polyhedron_GetTriangleData` function to get the data for a face in a polyhedron.

```
TQ3Status Q3Polyhedron_GetTriangleData (
                    TQ3GeometryObject polyhedron,
                    unsigned long triangleIndex,
                    TQ3PolyhedronTriangleData *triangleData);
```

`polyhedron` A polyhedron.

`triangleIndex` A triangle index. The value in this parameter should be greater than or equal to 0 and less than the total number of triangles (that is, faces) in the specified polyhedron.

`triangleData` On entry, a pointer to a polyhedron triangle data structure. On exit, the data in that structure is set to the specified triangle in the specified polyhedron.

DESCRIPTION

The `Q3Polyhedron_GetTriangleData` function returns, in the `triangleData` parameter, the data for the triangle specified by the `triangleIndex` parameter in the polyhedron specified by the `polyhedron` parameter.

Q3Polyhedron_SetTriangleData

You can use the `Q3Polyhedron_SetTriangleData` function to set the data for a face in a polyhedron.

```
TQ3Status Q3Polyhedron_SetTriangleData (
                    TQ3GeometryObject polyhedron,
                    unsigned long triangleIndex,
                    const TQ3PolyhedronTriangleData *triangleData);
```

`polyhedron` A polyhedron.

`triangleIndex` A triangle index. The value in this parameter should be greater than or equal to 0 and less than the total number of triangles (that is, faces) in the specified polyhedron.

`triangleData` A pointer to a polyhedron triangle data structure.

DESCRIPTION

The `Q3Polyhedron_SetTriangleData` function sets the data for the triangle specified by the `triangleIndex` parameter in the polyhedron specified by the `polyhedron` parameter to the data specified by the `triangleData` parameter.

Q3Polyhedron_GetEdgeData

You can use the `Q3Polyhedron_GetEdgeData` function to get the data that describe an edge in a polyhedron.

```
TQ3Status Q3Polyhedron_GetEdgeData (
                    TQ3GeometryObject polyhedron,
                    unsigned long edgeIndex,
                    TQ3PolyhedronEdgeData *edgeData);
```

`polyhedron` A polyhedron.

`edgeIndex` An edge index. The value in this parameter should be greater than or equal to 0 and less than the total number of edges in the specified polyhedron.

`edgeData` On entry, a pointer to a polyhedron edge data structure. On exit, the data in that structure is set to the specified edge in the specified polyhedron.

DESCRIPTION

The `Q3Polyhedron_GetEdgeData` function returns, in the `edgeData` parameter, the data for the edge specified by the `edgeIndex` parameter in the polyhedron specified by the `polyhedron` parameter.

Q3Polyhedron_SetEdgeData

You can use the `Q3Polyhedron_SetEdgeData` function to set the data that describe an edge in a polyhedron.

```
TQ3Status Q3Polyhedron_SetEdgeData (
                    TQ3GeometryObject polyhedron,
                    unsigned long edgeIndex,
                    const TQ3PolyhedronEdgeData *edgeData);
```

`polyhedron` A polyhedron.

`edgeIndex` An edge index. The value in this parameter should be greater than or equal to 0 and less than the total number of edges in the specified polyhedron.

`edgeData` A pointer to a polyhedron edge data structure.

DESCRIPTION

The `Q3Polyhedron_SetEdgeData` function sets the data for the edge specified by the `edgeIndex` parameter in the polyhedron specified by the `polyhedron` parameter to the data specified by the `edgeData` parameter.

Creating and Editing Ellipses

QuickDraw 3D provides routines that you can use to create and manipulate ellipses. See “Ellipses” (page 314) for the definition of an ellipse.

Q3Ellipse_New

You can use the `Q3Ellipse_New` function to create a new ellipse.

```
TQ3GeometryObject Q3Ellipse_New (
                    const TQ3EllipseData *ellipseData);
```

`ellipseData` A pointer to a `TQ3EllipseData` structure.

DESCRIPTION

The `Q3Ellipse_New` function returns, as its function result, a new ellipse having the shape and attributes specified by the `ellipseData` parameter. If a new ellipse could not be created, `Q3Ellipse_New` returns the value `NULL`.

Q3Ellipse_Submit

You can use the `Q3Ellipse_Submit` function to submit an immediate ellipse for drawing, picking, bounding, or writing.

```
TQ3Status Q3Ellipse_Submit (
                    const TQ3EllipseData *ellipseData,
                    TQ3ViewObject view);
```

`ellipseData` A pointer to a `TQ3EllipseData` structure.

`view` A view.

DESCRIPTION

The `Q3Ellipse_Submit` function submits for drawing, picking, bounding, or writing the immediate ellipse whose shape and attribute set are specified by the `ellipseData` parameter. The ellipse is drawn, picked, bounded, or written according to the view characteristics specified in the `view` parameter.

SPECIAL CONSIDERATIONS

You should call this function only in a submitting loop.

Q3Ellipse_GetData

You can use the `Q3Ellipse_GetData` function to get the data that defines an ellipse and its attributes.

```
TQ3Status Q3Ellipse_GetData (
                    TQ3GeometryObject ellipse,
                    TQ3EllipseData *ellipseData);
```

`ellipse` An ellipse.

`ellipseData` On exit, a pointer to a `TQ3EllipseData` structure that contains information about the ellipse specified by the `ellipse` parameter.

DESCRIPTION

The `Q3Ellipse_GetData` function returns, through the `ellipseData` parameter, information about the ellipse specified by the `ellipse` parameter. QuickDraw 3D allocates memory for the `TQ3EllipseData` structure internally; you must call `Q3Ellipse_EmptyData` to dispose of that memory.

Q3Ellipse_SetData

You can use the `Q3Ellipse_SetData` function to set the data that defines an ellipse and its attributes.

```
TQ3Status Q3Ellipse_SetData (
                    TQ3GeometryObject ellipse,
                    const TQ3EllipseData *ellipseData);
```

`ellipse` An ellipse.

`ellipseData` A pointer to a `TQ3EllipseData` structure.

DESCRIPTION

The `Q3Ellipse_SetData` function sets the data associated with the ellipse specified by the `ellipse` parameter to the data specified by the `ellipseData` parameter.

Q3Ellipse_EmptyData

You can use the `Q3Ellipse_EmptyData` function to release the memory occupied by the data structure returned by a previous call to `Q3Ellipse_GetData`.

```
TQ3Status Q3Ellipse_EmptyData (TQ3EllipseData *ellipseData);
```

`ellipseData` A pointer to a `TQ3EllipseData` structure.

DESCRIPTION

The `Q3Ellipse_EmptyData` function releases the memory occupied by the `TQ3EllipseData` structure pointed to by the `ellipseData` parameter; that memory was allocated by a previous call to `Q3Ellipse_GetData`.

Q3Ellipse_GetOrigin

You can use the `Q3Ellipse_GetOrigin` function to get the origin of an ellipse.

```
TQ3Status Q3Ellipse_GetOrigin (
                    TQ3GeometryObject ellipse,
                    TQ3Point3D *origin);
```

`ellipse` An ellipse.

`origin` On exit, the origin of the specified ellipse.

DESCRIPTION

The `Q3Ellipse_GetOrigin` function returns, in the `origin` parameter, the origin of the ellipse specified by the `ellipse` parameter.

Q3Ellipse_SetOrigin

You can use the `Q3Ellipse_SetOrigin` function to set the origin of an ellipse.

```
TQ3Status Q3Ellipse_SetOrigin (
                    TQ3GeometryObject ellipse,
                    const TQ3Point3D *origin);
```

`ellipse` An ellipse.

`origin` The desired origin of the specified ellipse.

DESCRIPTION

The `Q3Ellipse_SetOrigin` function sets the origin of the ellipse specified by the `ellipse` parameter to that specified in the `origin` parameter

Q3Ellipse_GetMajorRadius

You can use the `Q3Ellipse_GetMajorRadius` function to get the major radius of an ellipse.

```
TQ3Status Q3Ellipse_GetMajorRadius (
                    TQ3GeometryObject ellipse,
                    TQ3Vector3D *majorRadius);
```

`ellipse` An ellipse.

`majorRadius` On exit, the major radius of the specified ellipse.

DESCRIPTION

The `Q3Ellipse_GetMajorRadius` function returns, in the `majorRadius` parameter, the major radius of the ellipse specified by the `ellipse` parameter.

Q3Ellipse_SetMajorRadius

You can use the `Q3Ellipse_SetMajorRadius` function to set the major radius of an ellipse.

```
TQ3Status Q3Ellipse_SetMajorRadius (
                    TQ3GeometryObject ellipse,
                    const TQ3Vector3D *majorRadius);
```

`ellipse` An ellipse.

`majorRadius` The desired major radius of the specified ellipse.

DESCRIPTION

The `Q3Ellipse_SetMajorRadius` function sets the major radius of the ellipse specified by the `ellipse` parameter to that specified in the `majorRadius` parameter.

Q3Ellipse_GetMinorRadius

You can use the `Q3Ellipse_GetMinorRadius` function to get the minor radius of an ellipse.

```
TQ3Status Q3Ellipse_GetMinorRadius (
                    TQ3GeometryObject ellipse,
                    TQ3Vector3D *minorRadius);
```

`ellipse` An ellipse.

`minorRadius` On exit, the minor radius of the specified ellipse.

DESCRIPTION

The `Q3Ellipse_GetMinorRadius` function returns, in the `minorRadius` parameter, the minor radius of the ellipse specified by the `ellipse` parameter.

Q3Ellipse_SetMinorRadius

You can use the `Q3Ellipse_SetMinorRadius` function to set the minor radius of an ellipse.

```
TQ3Status Q3Ellipse_SetMinorRadius (
                    TQ3GeometryObject ellipse,
                    const TQ3Vector3D *minorRadius);
```

`ellipse` An ellipse.

`minorRadius` The desired minor radius of the specified ellipse.

DESCRIPTION

The `Q3Ellipse_SetMinorRadius` function sets the minor radius of the ellipse specified by the `ellipse` parameter to that specified in the `minorRadius` parameter.

Creating and Editing NURB Curves

QuickDraw 3D provides routines that you can use to create and manipulate NURB curves. See “NURB Curves” (page 315) for the definition of a NURB curve.

Q3NURBCurve_New

You can use the `Q3NURBCurve_New` function to create a new NURB curve.

```
TQ3GeometryObject Q3NURBCurve_New (const TQ3NURBCurveData *curveData);
```

`curveData` A pointer to a `TQ3NURBCurveData` structure.

DESCRIPTION

The `Q3NURBCurve_New` function returns, as its function result, a new NURB curve having the shape and attributes specified by the `curveData` parameter. If a new NURB curve could not be created, `Q3NURBCurve_New` returns the value `NULL`.

Q3NURBCurve_Submit

You can use the `Q3NURBCurve_Submit` function to submit an immediate NURB curve for drawing, picking, bounding, or writing.

```
TQ3Status Q3NURBCurve_Submit (
                    const TQ3NURBCurveData *curveData,
                    TQ3ViewObject view);
```

`curveData` A pointer to a `TQ3NURBCurveData` structure.

`view` A view.

DESCRIPTION

The `Q3NURBCurve_Submit` function submits for drawing, picking, bounding, or writing the immediate NURB curve whose shape and attribute set are specified by the `curveData` parameter. The NURB curve is drawn, picked, bounded, or written according to the view characteristics specified in the `view` parameter.

SPECIAL CONSIDERATIONS

You should call this function only in a submitting loop.

Q3NURBCurve_GetData

You can use the `Q3NURBCurve_GetData` function to get the data that defines a NURB curve and its attributes.

```
TQ3Status Q3NURBCurve_GetData (
                    TQ3GeometryObject curve,
                    TQ3NURBCurveData *nurbCurveData);
```

`curve` A NURB curve.

`nurbCurveData`
On exit, a pointer to a `TQ3NURBCurveData` structure that contains information about the NURB curve specified by the `curve` parameter.

DESCRIPTION

The `Q3NURBCurve_GetData` function returns, through the `nurbCurveData` parameter, information about the NURB curve specified by the `curve` parameter. QuickDraw 3D allocates memory for the `TQ3NURBCurveData` structure internally; you must call `Q3NURBCurve_EmptyData` to dispose of that memory.

Q3NURBCurve_SetData

You can use the `Q3NURBCurve_SetData` function to set the data that defines a NURB curve and its attributes.

```
TQ3Status Q3NURBCurve_SetData (
                    TQ3GeometryObject curve,
                    const TQ3NURBCurveData *nurbCurveData);
```

`curve` A NURB curve.

`nurbCurveData`
A pointer to a `TQ3NURBCurveData` structure.

DESCRIPTION

The `Q3NURBCurve_SetData` function sets the data associated with the NURB curve specified by the `curve` parameter to the data specified by the `nurbCurveData` parameter.

Q3NURBCurve_EmptyData

You can use the `Q3NURBCurve_EmptyData` function to release the memory occupied by the data structure returned by a previous call to `Q3NURBCurve_GetData`.

```
TQ3Status Q3NURBCurve_EmptyData (TQ3NURBCurveData *nurbCurveData);
```

`nurbCurveData`
A pointer to a `TQ3NURBCurveData` structure.

DESCRIPTION

The `Q3NURBCurve_EmptyData` function releases the memory occupied by the `TQ3NURBCurveData` structure pointed to by the `nurbCurveData` parameter; that memory was allocated by a previous call to `Q3NURBCurve_GetData`.

Q3NURBCurve_GetControlPoint

You can use the `Q3NURBCurve_GetControlPoint` function to get a four-dimensional control point for a NURB curve.

```
TQ3Status Q3NURBCurve_GetControlPoint (
                    TQ3GeometryObject curve,
                    unsigned long pointIndex,
                    TQ3RationalPoint4D *point4D);
```

`curve`	A NURB curve.
`pointIndex`	An index into the `controlPoints` array of control points for the specified NURB curve.
`point4D`	On exit, the control point having the specified index in the `controlPoints` array of control points for the specified NURB curve.

DESCRIPTION

The `Q3NURBCurve_GetControlPoint` function returns, in the `point4D` parameter, the four-dimensional control point of the NURB curve specified by the `curve` parameter having the index in the array of control points specified by the `pointIndex` parameter.

Q3NURBCurve_SetControlPoint

You can use the `Q3NURBCurve_SetControlPoint` function to set a four-dimensional control point for a NURB curve.

```
TQ3Status Q3NURBCurve_SetControlPoint (
                    TQ3GeometryObject curve,
                    unsigned long pointIndex,
                    const TQ3RationalPoint4D *point4D);
```

`curve` A NURB curve.

`pointIndex` An index into the `controlPoints` array of control points for the specified NURB curve.

`point4D` The desired four-dimensional control point.

DESCRIPTION

The `Q3NURBCurve_SetControlPoint` function sets the four-dimensional control point of the NURB curve specified by the `curve` parameter having the index in the array of control points specified by the `pointIndex` parameter to the point specified by the `point4D` parameter.

Q3NURBCurve_GetKnot

You can use the `Q3NURBCurve_GetKnot` function to get a knot of a NURB curve.

```
TQ3Status Q3NURBCurve_GetKnot (
                    TQ3GeometryObject curve,
                    unsigned long knotIndex,
                    float *knotValue);
```

`curve` A NURB curve.

`knotIndex` An index into the `knots` array for the specified NURB curve.

`knotValue` On exit, the value of the specified knot of the specified NURB curve.

DESCRIPTION

The `Q3NURBCurve_GetKnot` function returns, in the `knotValue` parameter, the value of the knot having the index specified by the `knotIndex` parameter in the `knots` array of the NURB curve specified by the `curve` parameter.

Q3NURBCurve_SetKnot

You can use the `Q3NURBCurve_SetKnot` function to set a knot of a NURB curve.

```
TQ3Status Q3NURBCurve_SetKnot (
                    TQ3GeometryObject curve,
                    unsigned long knotIndex,
                    float knotValue);
```

`curve` A NURB curve.

`knotIndex` An index into the `knots` array of knots for the specified NURB curve.

`knotValue` The desired value of the specified knot of the specified NURB curve.

DESCRIPTION

The `Q3NURBCurve_SetKnot` function sets the value of the knot having the index specified by the `knotIndex` parameter in the `knots` array of the NURB curve specified by the `curve` parameter to the value specified in the `knotValue` parameter.

Creating and Editing NURB Patches

QuickDraw 3D provides routines that you can use to create and manipulate NURB patches. See “NURB Patches” (page 317) for the definition of a NURB patch.

Q3NURBPatch_New

You can use the `Q3NURBPatch_New` function to create a new NURB patch.

```
TQ3GeometryObject Q3NURBPatch_New (
                    const TQ3NURBPatchData *nurbPatchData);
```

`nurbPatchData`
A pointer to a `TQ3NURBPatchData` structure.

DESCRIPTION

The `Q3NURBPatch_New` function returns, as its function result, a new NURB patch having the shape and attributes specified by the `nurbPatchData` parameter. If a new NURB patch could not be created, `Q3NURBPatch_New` returns the value `NULL`.

Q3NURBPatch_Submit

You can use the `Q3NURBPatch_Submit` function to submit an immediate NURB patch for drawing, picking, bounding, or writing.

```
TQ3Status Q3NURBPatch_Submit (
                    const TQ3NURBPatchData *nurbPatchData,
                    TQ3ViewObject view);
```

`nurbPatchData`
A pointer to a `TQ3NURBPatchData` structure.

`view` A view.

DESCRIPTION

The `Q3NURBPatch_Submit` function submits for drawing, picking, bounding, or writing the immediate NURB patch whose shape and attribute set are specified by the `nurbPatchData` parameter. The NURB patch is drawn, picked, bounded, or written according to the view characteristics specified in the `view` parameter.

SPECIAL CONSIDERATIONS

You should call this function only in a submitting loop.

Q3NURBPatch_GetData

You can use the `Q3NURBPatch_GetData` function to get the data that defines a NURB patch and its attributes.

```
TQ3Status Q3NURBPatch_GetData (
                    TQ3GeometryObject nurbPatch,
                    TQ3NURBPatchData *nurbPatchData);
```

`nurbPatch` A NURB patch.

`nurbPatchData`
On exit, a pointer to a `TQ3NURBPatchData` structure that contains information about the NURB patch specified by the `nurbPatch` parameter.

DESCRIPTION

The `Q3NURBPatch_GetData` function returns, through the `nurbPatchData` parameter, information about the NURB patch specified by the `nurbPatch` parameter. QuickDraw 3D allocates memory for the `TQ3NURBPatchData` structure internally; you must call `Q3NURBPatch_EmptyData` to dispose of that memory.

Q3NURBPatch_SetData

You can use the `Q3NURBPatch_SetData` function to set the data that defines a NURB patch and its attributes.

```
TQ3Status Q3NURBPatch_SetData (
                    TQ3GeometryObject nurbPatch,
                    const TQ3NURBPatchData *nurbPatchData);
```

`nurbPatch` A NURB patch.

`nurbPatchData`
A pointer to a `TQ3NURBPatchData` structure.

DESCRIPTION

The `Q3NURBPatch_SetData` function sets the data associated with the NURB patch specified by the `nurbPatch` parameter to the data specified by the `nurbPatchData` parameter.

Q3NURBPatch_EmptyData

You can use the `Q3NURBPatch_EmptyData` function to release the memory occupied by the data structure returned by a previous call to `Q3NURBPatch_GetData`.

```
TQ3Status Q3NURBPatch_EmptyData (
                    TQ3NURBPatchData *nurbPatchData);
```

`nurbPatchData`
A pointer to a `TQ3NURBPatchData` structure.

DESCRIPTION

The `Q3NURBPatch_EmptyData` function releases the memory occupied by the `TQ3NURBPatchData` structure pointed to by the `nurbPatchData` parameter; that memory was allocated by a previous call to `Q3NURBPatch_GetData`.

Q3NURBPatch_GetControlPoint

You can use the `Q3NURBPatch_GetControlPoint` function to get a control point for a NURB patch.

```
TQ3Status Q3NURBPatch_GetControlPoint (
                    TQ3GeometryObject nurbPatch,
                    unsigned long rowIndex,
                    unsigned long columnIndex,
                    TQ3RationalPoint4D *point4D);
```

`nurbPatch`	A NURB patch.
`rowIndex`	A row index into the array of control points for the specified NURB patch.
`columnIndex`	A column index into the array of control points for the specified NURB patch.
`point4D`	On exit, the control point having the specified row and column indices in the `controlPoints` array of control points for the specified NURB patch.

DESCRIPTION

The `Q3NURBPatch_GetControlPoint` function returns, in the `point4D` parameter, the four-dimensional control point of the NURB patch specified by the `nurbPatch` parameter having the row and column indices `rowIndex` and `columnIndex` in the `controlPoints` array of control points.

Q3NURBPatch_SetControlPoint

You can use the `Q3NURBPatch_SetControlPoint` function to set a control point for a NURB patch.

```
TQ3Status Q3NURBPatch_SetControlPoint (
                    TQ3GeometryObject nurbPatch,
                    unsigned long rowIndex,
                    unsigned long columnIndex,
                    const TQ3RationalPoint4D *point4D);
```

`nurbPatch`	A NURB patch.
`rowIndex`	A row index into the array of control points for the specified NURB patch.
`columnIndex`	A column index into the array of control points for the specified NURB patch.
`point4D`	The desired four-dimensional control point.

DESCRIPTION

The `Q3NURBPatch_SetControlPoint` function sets the four-dimensional control point having the row and column indices `rowIndex` and `columnIndex` in the `controlPoints` array of control points of the NURB patch specified by the `nurbPatch` parameter to the point specified by the `point4D` parameter.

Q3NURBPatch_GetUKnot

You can use the `Q3NURBPatch_GetUKnot` function to get the value of a knot in the *u* parametric direction.

```
TQ3Status Q3NURBPatch_GetUKnot (
                    TQ3GeometryObject nurbPatch,
                    unsigned long knotIndex,
                    float *knotValue);
```

`nurbPatch` A NURB patch.

`knotIndex` An index into the `uKnots` field of the specified NURB patch.

`knotValue` On exit, the value of the specified knot.

DESCRIPTION

The `Q3NURBPatch_GetUKnot` function returns, in the `knotValue` parameter, the knot value of the NURB patch specified by the `nurbPatch` parameter having the knot index specified by the `knotIndex` parameter in the `uKnots` array of *u* knots.

Q3NURBPatch_SetUKnot

You can use the `Q3NURBPatch_SetUKnot` function to set the value of a knot in the *u* parametric direction.

```
TQ3Status Q3NURBPatch_SetUKnot (
                    TQ3GeometryObject nurbPatch,
                    unsigned long knotIndex,
                    float knotValue);
```

`nurbPatch`	A NURB patch.
`knotIndex`	An index into the `uKnots` field of the specified NURB patch.
`knotValue`	The desired value of the specified knot.

DESCRIPTION

The `Q3NURBPatch_SetUKnot` function sets the knot value of the NURB patch specified by the `nurbPatch` parameter having the knot index specified by the `knotIndex` parameter in the `uKnots` array of *u* knots to the value specified by the `knotValue` parameter.

Q3NURBPatch_GetVKnot

You can use the `Q3NURBPatch_GetVKnot` function to get the value of a knot in the *v* parametric direction.

```
TQ3Status Q3NURBPatch_GetVKnot (
                    TQ3GeometryObject nurbPatch,
                    unsigned long knotIndex,
                    float *knotValue);
```

`nurbPatch`	A NURB patch.
`knotIndex`	An index into the `vKnots` field of the specified NURB patch.
`knotValue`	On exit, the value of the specified knot.

DESCRIPTION

The `Q3NURBPatch_GetVKnot` function returns, in the `knotValue` parameter, the knot value of the NURB patch specified by the `nurbPatch` parameter having the knot index specified by the `knotIndex` parameter in the `vKnots` array of *v* knots.

Q3NURBPatch_SetVKnot

You can use the `Q3NURBPatch_SetVKnot` function to set the value of a knot in the v parametric direction.

```
TQ3Status Q3NURBPatch_SetVKnot (
                    TQ3GeometryObject nurbPatch,
                    unsigned long knotIndex,
                    float knotValue);
```

`nurbPatch` A NURB patch.

`knotIndex` An index into the `vKnots` field of the specified NURB patch.

`knotValue` The desired value of the specified knot.

DESCRIPTION

The `Q3NURBPatch_SetVKnot` function sets the knot value of the NURB patch specified by the `nurbPatch` parameter having the knot index specified by the `knotIndex` parameter in the `vKnots` array of v knots to the value specified by the `knotValue` parameter.

Creating and Editing Ellipsoids

QuickDraw 3D provides routines that you can use to create and manipulate ellipsoids. See "Ellipsoids" (page 320) for the definition of an ellipsoid.

Q3Ellipsoid_New

You can use the `Q3Ellipsoid_New` function to create a new ellipsoid.

```
TQ3GeometryObject Q3Ellipsoid_New (
                    const TQ3EllipsoidData *ellipsoidData);
```

`ellipsoidData`
A pointer to a `TQ3EllipsoidData` structure.

DESCRIPTION

The `Q3Ellipsoid_New` function returns, as its function result, a new ellipsoid having the shape and attributes specified by the `ellipsoidData` parameter. If a new ellipsoid could not be created, `Q3Ellipsoid_New` returns the value `NULL`.

Q3Ellipsoid_Submit

You can use the `Q3Ellipsoid_Submit` function to submit an immediate ellipsoid for drawing, picking, bounding, or writing.

```
TQ3Status Q3Ellipsoid_Submit (
                    const TQ3EllipsoidData *ellipsoidData,
                    TQ3ViewObject view);
```

`ellipsoidData`
A pointer to a `TQ3EllipsoidData` structure.

`view`
A view.

DESCRIPTION

The `Q3Ellipsoid_Submit` function submits for drawing, picking, bounding, or writing the immediate ellipsoid whose shape and attribute set are specified by the `ellipsoidData` parameter. The ellipsoid is drawn, picked, bounded, or written according to the view characteristics specified in the `view` parameter.

SPECIAL CONSIDERATIONS

You should call this function only in a submitting loop.

Q3Ellipsoid_GetData

You can use the `Q3Ellipsoid_GetData` function to get the data that defines an ellipsoid and its attributes.

```
TQ3Status Q3Ellipsoid_GetData (
                    TQ3GeometryObject ellipsoid,
                    TQ3EllipsoidData *ellipsoidData);
```

`ellipsoid` An ellipsoid.

`ellipsoidData`
On exit, a pointer to a `TQ3EllipsoidData` structure that contains information about the ellipsoid specified by the `ellipsoid` parameter.

DESCRIPTION

The `Q3Ellipsoid_GetData` function returns, through the `ellipsoidData` parameter, information about the ellipsoid specified by the `ellipsoid` parameter. QuickDraw 3D allocates memory for the `TQ3EllipsoidData` structure internally; you must call `Q3Ellipsoid_EmptyData` to dispose of that memory.

Q3Ellipsoid_SetData

You can use the `Q3Ellipsoid_SetData` function to set the data that defines an ellipsoid and its attributes.

```
TQ3Status Q3Ellipsoid_SetData (
                    TQ3GeometryObject ellipsoid,
                    const TQ3EllipsoidData *ellipsoidData);
```

`ellipsoid` An ellipsoid.

`ellipsoidData`
A pointer to a `TQ3EllipsoidData` structure.

DESCRIPTION

The `Q3Ellipsoid_SetData` function sets the data associated with the ellipsoid specified by the `ellipsoid` parameter to the data specified by the `ellipsoidData` parameter.

Q3Ellipsoid_EmptyData

You can use the `Q3Ellipsoid_EmptyData` function to release the memory occupied by the data structure returned by a previous call to `Q3Ellipsoid_GetData`.

```
TQ3Status Q3Ellipsoid_EmptyData (TQ3EllipsoidData *ellipsoidData);
```

`ellipsoidData`
A pointer to a `TQ3EllipsoidData` structure.

DESCRIPTION

The `Q3Ellipsoid_EmptyData` function releases the memory occupied by the `TQ3EllipsoidData` structure pointed to by the `ellipsoidData` parameter; that memory was allocated by a previous call to `Q3Ellipsoid_GetData`.

Q3Ellipsoid_GetOrigin

You can use the `Q3Ellipsoid_GetOrigin` function to get the origin of an ellipsoid.

```
TQ3Status Q3Ellipsoid_GetOrigin (
                    TQ3GeometryObject ellipsoid,
                    TQ3Point3D *origin);
```

`ellipsoid` An ellipsoid.

`origin` On exit, the origin of the specified ellipsoid.

DESCRIPTION

The `Q3Ellipsoid_GetOrigin` function returns, in the `origin` parameter, the origin of the ellipsoid specified by the `ellipsoid` parameter.

Q3Ellipsoid_SetOrigin

You can use the `Q3Ellipsoid_SetOrigin` function to set the origin of an ellipsoid.

```
TQ3Status Q3Ellipsoid_SetOrigin (
                    TQ3GeometryObject ellipsoid,
                    const TQ3Point3D *origin);
```

`ellipsoid` An ellipsoid.

`origin` The desired origin of the specified ellipsoid.

DESCRIPTION

The `Q3Ellipsoid_SetOrigin` function sets the origin of the ellipsoid specified by the `ellipsoid` parameter to that specified in the `origin` parameter.

Q3Ellipsoid_GetOrientation

You can use the `Q3Ellipsoid_GetOrientation` function to get the orientation of an ellipsoid.

```
TQ3Status Q3Ellipsoid_GetOrientation (
                    TQ3GeometryObject ellipsoid,
                    TQ3Vector3D *orientation);
```

`ellipsoid` An ellipsoid.

`orientation` On exit, the orientation of the specified ellipsoid.

DESCRIPTION

The `Q3Ellipsoid_GetOrientation` function returns, in the `orientation` parameter, the orientation of the ellipsoid specified by the `ellipsoid` parameter.

Q3Ellipsoid_SetOrientation

You can use the `Q3Ellipsoid_SetOrientation` function to set the orientation of an ellipsoid.

```
TQ3Status Q3Ellipsoid_SetOrientation (
                    TQ3GeometryObject ellipsoid,
                    const TQ3Vector3D *orientation);
```

`ellipsoid` An ellipsoid.

`orientation` The desired orientation of the specified ellipsoid.

DESCRIPTION

The `Q3Ellipsoid_SetOrientation` function sets the orientation of the ellipsoid specified by the `ellipsoid` parameter to that specified in the `orientation` parameter.

Q3Ellipsoid_GetMajorRadius

You can use the `Q3Ellipsoid_GetMajorRadius` function to get the major radius of an ellipsoid.

```
TQ3Status Q3Ellipsoid_GetMajorRadius (
                    TQ3GeometryObject ellipsoid,
                    TQ3Vector3D *majorRadius);
```

`ellipsoid` An ellipsoid.

`majorRadius` On exit, the major radius of the specified ellipsoid.

DESCRIPTION

The `Q3Ellipsoid_GetMajorRadius` function returns, in the `majorRadius` parameter, the major radius of the ellipsoid specified by the `ellipsoid` parameter.

Q3Ellipsoid_SetMajorRadius

You can use the `Q3Ellipsoid_SetMajorRadius` function to set the major radius of an ellipsoid.

```
TQ3Status Q3Ellipsoid_SetMajorRadius (
                    TQ3GeometryObject ellipsoid,
                    const TQ3Vector3D *majorRadius);
```

`ellipsoid` An ellipsoid.

`majorRadius` The desired major radius of the specified ellipsoid.

DESCRIPTION

The `Q3Ellipsoid_SetMajorRadius` function sets the major radius of the ellipsoid specified by the `ellipsoid` parameter to that specified in the `majorRadius` parameter.

Q3Ellipsoid_GetMinorRadius

You can use the `Q3Ellipsoid_GetMinorRadius` function to get the minor radius of an ellipsoid.

```
TQ3Status Q3Ellipsoid_GetMinorRadius (
                    TQ3GeometryObject ellipsoid,
                    TQ3Vector3D *minorRadius);
```

`ellipsoid` An ellipsoid.

`minorRadius` On exit, the minor radius of the specified ellipsoid.

DESCRIPTION

The `Q3Ellipsoid_GetMinorRadius` function returns, in the `minorRadius` parameter, the minor radius of the ellipsoid specified by the `ellipsoid` parameter.

Q3Ellipsoid_SetMinorRadius

You can use the `Q3Ellipsoid_SetMinorRadius` function to set the minor radius of an ellipsoid.

```
TQ3Status Q3Ellipsoid_SetMinorRadius (
                    TQ3GeometryObject ellipsoid,
                    const TQ3Vector3D *minorRadius);
```

`ellipsoid` An ellipsoid.

`minorRadius` The desired minor radius of the specified ellipsoid.

DESCRIPTION

The `Q3Ellipsoid_SetMinorRadius` function sets the minor radius of the ellipsoid specified by the `ellipsoid` parameter to that specified in the `minorRadius` parameter.

Creating and Editing Cylinders

QuickDraw 3D provides routines that you can use to create and manipulate cylinders. See “Cylinders” (page 322) for the definition of a cylinder.

Q3Cylinder_New

You can use the `Q3Cylinder_New` function to create a new cylinder.

```
TQ3GeometryObject Q3Cylinder_New (const TQ3CylinderData *cylinderData);
```

`cylinderData` A pointer to a `TQ3CylinderData` structure.

DESCRIPTION

The `Q3Cylinder_New` function returns, as its function result, a new cylinder having the shape attributes specified by the `cylinderData` parameter. If a new cylinder could not be created, `Q3Cylinder_New` returns the value `NULL`.

Q3Cylinder_Submit

You can use the `Q3Cylinder_Submit` function to submit an immediate cylinder for drawing, picking, bounding, or writing.

```
TQ3Status Q3Cylinder_Submit (
                    const TQ3CylinderData *cylinderData,
                    TQ3ViewObject view);
```

`cylinderData` A pointer to a `TQ3CylinderData` structure.

`view` A view.

DESCRIPTION

The `Q3Cylinder_Submit` function submits for drawing, picking, bounding, or writing the immediate cylinder whose shape and attribute set are specified by the `cylinderData` parameter. The cylinder is drawn, picked, bounded, or written according to the view characteristics specified in the `view` parameter.

SPECIAL CONSIDERATIONS

You should call this function only in a submitting loop.

Q3Cylinder_GetData

You can use the `Q3Cylinder_GetData` function to get the data that defines a cylinder and its attributes.

```
TQ3Status Q3Cylinder_GetData (
                    TQ3GeometryObject cylinder,
                    TQ3CylinderData *cylinderData);
```

`cylinder` A cylinder.

`cylinderData`

On exit, a pointer to a `TQ3CylinderData` structure that contains information about the cylinder specified by the `cylinder` parameter.

DESCRIPTION

The `Q3Cylinder_GetData` function returns, through the `cylinderData` parameter, information about the cylinder specified by the `cylinder` parameter. QuickDraw 3D allocates memory for the `TQ3CylinderData` structure internally; you must call `Q3Cylinder_EmptyData` to dispose of that memory.

Q3Cylinder_SetData

You can use the `Q3Cylinder_SetData` function to set the data that defines a cylinder and its attributes.

```
TQ3Status Q3Cylinder_SetData (
                    TQ3GeometryObject cylinder,
                    const TQ3CylinderData *cylinderData);
```

`cylinder` A cylinder.

`cylinderData`
A pointer to a `TQ3CylinderData` structure.

DESCRIPTION

The `Q3Cylinder_SetData` function sets the data associated with the cylinder specified by the `cylinder` parameter to the data specified by the `cylinderData` parameter.

Q3Cylinder_EmptyData

You can use the `Q3Cylinder_EmptyData` function to release the memory occupied by the data structure returned by a previous call to `Q3Cylinder_GetData`.

```
TQ3Status Q3Cylinder_EmptyData (TQ3CylinderData *cylinderData);
```

`cylinderData`
A pointer to a `TQ3CylinderData` structure.

DESCRIPTION

The `Q3Cylinder_EmptyData` function releases the memory occupied by the `TQ3CylinderData` structure pointed to by the `cylinderData` parameter; that memory was allocated by a previous call to `Q3Cylinder_GetData`.

Q3Cylinder_GetOrigin

You can use the `Q3Cylinder_GetOrigin` function to get the origin of a cylinder.

```
TQ3Status Q3Cylinder_GetOrigin (
                    TQ3GeometryObject cylinder,
                    TQ3Point3D *origin);
```

`cylinder` A cylinder.

`origin` On exit, the origin of the specified cylinder.

DESCRIPTION

The `Q3Cylinder_GetOrigin` function returns, in the `origin` parameter, the origin of the cylinder specified by the `cylinder` parameter.

Q3Cylinder_SetOrigin

You can use the `Q3Cylinder_SetOrigin` function to set the origin of a cylinder.

```
TQ3Status Q3Cylinder_SetOrigin (
                    TQ3GeometryObject cylinder,
                    const TQ3Point3D *origin);
```

`cylinder` A cylinder.

`origin` The desired origin of the specified cylinder.

DESCRIPTION

The `Q3Cylinder_SetOrigin` function sets the origin of the cylinder specified by the `cylinder` parameter to that specified in the `origin` parameter.

Q3Cylinder_GetOrientation

You can use the `Q3Cylinder_GetOrientation` function to get the orientation of a cylinder.

```
TQ3Status Q3Cylinder_GetOrientation (
                    TQ3GeometryObject cylinder,
                    TQ3Vector3D *orientation);
```

`cylinder` A cylinder.

`orientation` On exit, the orientation of the specified cylinder.

DESCRIPTION

The `Q3Cylinder_GetOrientation` function returns, in the `orientation` parameter, the orientation of the cylinder specified by the `cylinder` parameter.

Q3Cylinder_SetOrientation

You can use the `Q3Cylinder_SetOrientation` function to set the orientation of a cylinder.

```
TQ3Status Q3Cylinder_SetOrientation (
                    TQ3GeometryObject cylinder,
                    const TQ3Vector3D *orientation);
```

`cylinder` A cylinder.

`orientation` The desired orientation of the specified cylinder.

DESCRIPTION

The `Q3Cylinder_SetOrientation` function sets the orientation of the cylinder specified by the `cylinder` parameter to that specified in the `orientation` parameter.

Q3Cylinder_GetMajorRadius

You can use the `Q3Cylinder_GetMajorRadius` function to get the major radius of a cylinder.

```
TQ3Status Q3Cylinder_GetMajorRadius (
                    TQ3GeometryObject cylinder,
                    TQ3Vector3D *majorRadius);
```

`cylinder` A cylinder.

`majorRadius` On exit, the major radius of the specified cylinder.

DESCRIPTION

The `Q3Cylinder_GetMajorRadius` function returns, in the `majorRadius` parameter, the major radius of the cylinder specified by the `cylinder` parameter.

Q3Cylinder_SetMajorRadius

You can use the `Q3Cylinder_SetMajorRadius` function to set the major radius of a cylinder.

```
TQ3Status Q3Cylinder_SetMajorRadius (
                    TQ3GeometryObject cylinder,
                    const TQ3Vector3D *majorRadius);
```

`cylinder` A cylinder.

`majorRadius` The desired major radius of the specified cylinder.

DESCRIPTION

The `Q3Cylinder_SetMajorRadius` function sets the major radius of the cylinder specified by the `cylinder` parameter to that specified in the `majorRadius` parameter.

Q3Cylinder_GetMinorRadius

You can use the `Q3Cylinder_GetMinorRadius` function to get the minor radius of a cylinder.

```
TQ3Status Q3Cylinder_GetMinorRadius (
                    TQ3GeometryObject cylinder,
                    TQ3Vector3D *minorRadius);
```

`cylinder` A cylinder.

`minorRadius` On exit, the minor radius of the specified cylinder.

DESCRIPTION

The `Q3Cylinder_GetMinorRadius` function returns, in the `minorRadius` parameter, the minor radius of the cylinder specified by the `cylinder` parameter.

Q3Cylinder_SetMinorRadius

You can use the `Q3Cylinder_SetMinorRadius` function to set the minor radius of a cylinder.

```
TQ3Status Q3Cylinder_SetMinorRadius (
                    TQ3GeometryObject cylinder,
                    const TQ3Vector3D *minorRadius);
```

`cylinder` A cylinder.

`minorRadius` The desired minor radius of the specified cylinder.

DESCRIPTION

The `Q3Cylinder_SetMinorRadius` function sets the minor radius of the cylinder specified by the `cylinder` parameter to that specified in the `minorRadius` parameter.

Q3Cylinder_GetCaps

You can use the `Q3Cylinder_GetCaps` function to get the style of caps of a cylinder.

```
TQ3Status Q3Cylinder_GetCaps (
                    TQ3GeometryObject cylinder,
                    TQ3EndCap *caps);
```

`cylinder` A cylinder.

`caps` On exit, the caps style of the specified cylinder.

DESCRIPTION

The `Q3Cylinder_GetCaps` function returns, in the `caps` parameter, the style of caps of the cylinder specified by the `cylinder` parameter.

Q3Cylinder_SetCaps

You can use the `Q3Cylinder_SetCaps` function to set the style of caps of a cylinder.

```
TQ3Status Q3Cylinder_SetCaps (
                    TQ3GeometryObject cylinder,
                    TQ3EndCap caps);
```

`cylinder` A cylinder.

`caps` The desired style of end caps of the specified cylinder.

DESCRIPTION

The `Q3Cylinder_SetCaps` function sets the style of end caps of the cylinder specified by the `cylinder` parameter to that specified in the `caps` parameter.

Q3Cylinder_GetTopAttributeSet

You can use the `Q3Cylinder_GetTopAttributeSet` function to get the top attribute set of a cylinder.

```
TQ3Status Q3Cylinder_GetTopAttributeSet (
                    TQ3GeometryObject cylinder,
                    TQ3AttributeSet *topAttributeSet);
```

`cylinder` A cylinder.

`topAttributeSet`
On exit, the attribute set of the top of the specified cylinder.

DESCRIPTION

The `Q3Cylinder_GetTopAttributeSet` function returns, in the `topAttributeSet` parameter, the attribute set of the top of the cylinder specified by the `cylinder` parameter. The reference count of the set is incremented.

Q3Cylinder_SetTopAttributeSet

You can use the `Q3Cylinder_SetTopAttributeSet` function to set the top attribute set of a cylinder.

```
TQ3Status Q3Cylinder_SetTopAttributeSet (
                    TQ3GeometryObject cylinder,
                    TQ3AttributeSet topAttributeSet);
```

`cylinder` A cylinder.

`topAttributeSet`
The desired attribute set of the top of the specified cylinder.

DESCRIPTION

The `Q3Cylinder_SetTopAttributeSet` function sets the attribute set of the top of the cylinder specified by the `cylinder` parameter to that specified in the `topAttributeSet` parameter.

Q3Cylinder_GetFaceAttributeSet

You can use the `Q3Cylinder_GetFaceAttributeSet` function to get the face attribute set of a cylinder.

```
TQ3Status Q3Cylinder_GetFaceAttributeSet (
                    TQ3GeometryObject cylinder,
                    TQ3AttributeSet *faceAttributeSet);
```

`cylinder` A cylinder.

`faceAttributeSet`
On exit, the attribute set of the face of the specified cylinder.

DESCRIPTION

The `Q3Cylinder_GetFaceAttributeSet` function returns, in the `faceAttributeSet` parameter, the attribute set of the face of the cylinder specified by the `cylinder` parameter. The reference count of the set is incremented.

Q3Cylinder_SetFaceAttributeSet

You can use the `Q3Cylinder_SetFaceAttributeSet` function to set the face attribute set of a cylinder.

```
TQ3Status Q3Cylinder_SetFaceAttributeSet (
                    TQ3GeometryObject cylinder,
                    TQ3AttributeSet faceAttributeSet);
```

`cylinder` A cylinder.

`faceAttributeSet`
The desired attribute set of the face of the specified cylinder.

DESCRIPTION

The `Q3Cylinder_SetFaceAttributeSet` function sets the attribute set of the face of the cylinder specified by the `cylinder` parameter to that specified in the `faceAttributeSet` parameter.

Q3Cylinder_GetBottomAttributeSet

You can use the `Q3Cylinder_GetBottomAttributeSet` function to get the bottom attribute set of a cylinder.

```
TQ3Status Q3Cylinder_GetBottomAttributeSet (
                    TQ3GeometryObject cylinder,
                    TQ3AttributeSet *bottomAttributeSet);
```

`cylinder` A cylinder.

`bottomAttributeSet`
On exit, the attribute set of the bottom of the specified cylinder.

DESCRIPTION

The `Q3Cylinder_GetBottomAttributeSet` function returns, in the `bottomAttributeSet` parameter, the attribute set of the bottom of the cylinder specified by the `cylinder` parameter. The reference count of the set is incremented.

Q3Cylinder_SetBottomAttributeSet

You can use the `Q3Cylinder_SetBottomAttributeSet` function to set the bottom attribute set of a cylinder.

```
TQ3Status Q3Cylinder_SetBottomAttributeSet (
                    TQ3GeometryObject cylinder,
                    TQ3AttributeSet bottomAttributeSet);
```

`cylinder` A cylinder.

`bottomAttributeSet`
The desired attribute set of the bottom of the specified cylinder.

DESCRIPTION

The `Q3Cylinder_SetBottomAttributeSet` function sets the attribute set of the bottom of the cylinder specified by the `cylinder` parameter to that specified in the `bottomAttributeSet` parameter.

Creating and Editing Disks

QuickDraw 3D provides routines that you can use to create and manipulate disks. See "Disks" (page 323) for the definition of a disk.

Q3Disk_New

You can use the `Q3Disk_New` function to create a new disk.

```
TQ3GeometryObject Q3Disk_New (const TQ3DiskData *diskData);
```

`diskData` A pointer to a `TQ3DiskData` structure.

DESCRIPTION

The `Q3Disk_New` function returns, as its function result, a new disk having the shape and attributes specified by the `diskData` parameter. If a new disk could not be created, `Q3Disk_New` returns the value `NULL`.

Q3Disk_Submit

You can use the `Q3Disk_Submit` function to submit an immediate disk for drawing, picking, bounding, or writing.

```
TQ3Status Q3Disk_Submit (
                    const TQ3DiskData *diskData,
                    TQ3ViewObject view);
```

`diskData` A pointer to a `TQ3DiskData` structure.

`view` A view.

DESCRIPTION

The `Q3Disk_Submit` function submits for drawing, picking, bounding, or writing the immediate disk whose shape and attribute set are specified by the `diskData` parameter. The disk is drawn, picked, bounded, or written according to the view characteristics specified in the `view` parameter.

SPECIAL CONSIDERATIONS

You should call this function only in a submitting loop.

Q3Disk_GetData

You can use the `Q3Disk_GetData` function to get the data that defines a disk and its attributes.

```
TQ3Status Q3Disk_GetData (
                    TQ3GeometryObject disk,
                    TQ3DiskData *diskData);
```

`disk` A disk.

`diskData` On exit, a pointer to a `TQ3DiskData` structure that contains information about the disk specified by the `disk` parameter.

DESCRIPTION

The `Q3Disk_GetData` function returns, through the `diskData` parameter, information about the disk specified by the `disk` parameter. QuickDraw 3D allocates memory for the `TQ3DiskData` structure internally; you must call `Q3Disk_EmptyData` to dispose of that memory.

Q3Disk_SetData

You can use the `Q3Disk_SetData` function to set the data that defines a disk and its attributes.

```
TQ3Status Q3Disk_SetData (
                    TQ3GeometryObject disk,
                    const TQ3DiskData *diskData);
```

`disk` A disk.

`diskData` A pointer to a `TQ3DiskData` structure.

DESCRIPTION

The `Q3Disk_SetData` function sets the data associated with the disk specified by the `disk` parameter to the data specified by the `diskData` parameter.

Q3Disk_EmptyData

You can use the `Q3Disk_EmptyData` function to release the memory occupied by the data structure returned by a previous call to `Q3Disk_GetData`.

```
TQ3Status Q3Disk_EmptyData (TQ3DiskData *diskData);
```

`diskData` A pointer to a `TQ3DiskData` structure.

DESCRIPTION

The `Q3Disk_EmptyData` function releases the memory occupied by the `TQ3DiskData` structure pointed to by the `diskData` parameter; that memory was allocated by a previous call to `Q3Disk_GetData`.

Q3Disk_GetOrigin

You can use the `Q3Disk_GetOrigin` function to get the origin of a disk.

```
TQ3Status Q3Disk_GetOrigin (
                    TQ3GeometryObject disk,
                    TQ3Point3D *origin);
```

`disk` A disk.

`origin` On exit, the origin of the specified disk.

DESCRIPTION

The `Q3Disk_GetOrigin` function returns, in the `origin` parameter, the origin of the disk specified by the `disk` parameter.

Q3Disk_SetOrigin

You can use the `Q3Disk_SetOrigin` function to set the origin of a disk.

```
TQ3Status Q3Disk_SetOrigin (
                    TQ3GeometryObject disk,
                    const TQ3Point3D *origin);
```

`disk` A disk.

`origin` The desired origin of the specified disk.

DESCRIPTION

The `Q3Disk_SetOrigin` function sets the origin of the disk specified by the `disk` parameter to that specified in the `origin` parameter.

Q3Disk_GetMajorRadius

You can use the `Q3Disk_GetMajorRadius` function to get the major radius of a disk.

```
TQ3Status Q3Disk_GetMajorRadius (
                    TQ3GeometryObject disk,
                    TQ3Vector3D *majorRadius);
```

`disk` A disk.

`majorRadius` On exit, the major radius of the specified disk.

DESCRIPTION

The `Q3Disk_GetMajorRadius` function returns, in the `majorRadius` parameter, the major radius of the disk specified by the `disk` parameter.

Q3Disk_SetMajorRadius

You can use the `Q3Disk_SetMajorRadius` function to set the major radius of a disk.

```
TQ3Status Q3Disk_SetMajorRadius (
                    TQ3GeometryObject disk,
                    const TQ3Vector3D *majorRadius);
```

`disk` A disk.

`majorRadius` The desired major radius of the specified disk.

DESCRIPTION

The `Q3Disk_SetMajorRadius` function sets the major radius of the disk specified by the `disk` parameter to that specified in the `majorRadius` parameter.

Q3Disk_GetMinorRadius

You can use the `Q3Disk_GetMinorRadius` function to get the minor radius of a disk.

```
TQ3Status Q3Disk_GetMinorRadius (
                    TQ3GeometryObject disk,
                    TQ3Vector3D *minorRadius);
```

`disk` A disk.

`minorRadius` On exit, the minor radius of the specified disk.

DESCRIPTION

The `Q3Disk_GetMinorRadius` function returns, in the `minorRadius` parameter, the minor radius of the disk specified by the `disk` parameter.

Q3Disk_SetMinorRadius

You can use the `Q3Disk_SetMinorRadius` function to set the minor radius of a disk.

```
TQ3Status Q3Disk_SetMinorRadius (
                    TQ3GeometryObject disk,
                    const TQ3Vector3D *minorRadius);
```

`disk` A disk.

`minorRadius` The desired minor radius of the specified disk.

DESCRIPTION

The `Q3Disk_SetMinorRadius` function sets the minor radius of the disk specified by the `disk` parameter to that specified in the `minorRadius` parameter.

Creating and Editing Cones

QuickDraw 3D provides routines that you can use to create and manipulate cones. See "Cones" (page 325) for the definition of a cone.

Q3Cone_New

You can use the `Q3Cone_New` function to create a new cone.

```
TQ3GeometryObject Q3Cone_New (const TQ3ConeData *coneData);
```

`coneData` A pointer to a `TQ3ConeData` structure.

DESCRIPTION

The `Q3Cone_New` function returns, as its function result, a new cone having the shape and attributes specified by the `coneData` parameter. If a new cone could not be created, `Q3Cone_New` returns the value `NULL`.

Q3Cone_Submit

You can use the `Q3Cone_Submit` function to submit an immediate cone for drawing, picking, bounding, or writing.

```
TQ3Status Q3Cone_Submit (
                    const TQ3ConeData *coneData,
                    TQ3ViewObject view);
```

`coneData` A pointer to a `TQ3ConeData` structure.

`view` A view.

DESCRIPTION

The `Q3Cone_Submit` function submits for drawing, picking, bounding, or writing the immediate cone whose shape and attribute set are specified by the `coneData` parameter. The cone is drawn, picked, bounded, or written according to the view characteristics specified in the `view` parameter.

SPECIAL CONSIDERATIONS

You should call this function only in a submitting loop.

Q3Cone_GetData

You can use the `Q3Cone_GetData` function to get the data that defines a cone and its attributes.

```
TQ3Status Q3Cone_GetData (TQ3GeometryObject cone, TQ3ConeData *coneData);
```

`cone`	A cone.
`coneData`	On exit, a pointer to a `TQ3ConeData` structure that contains information about the cone specified by the `cone` parameter.

DESCRIPTION

The `Q3Cone_GetData` function returns, through the `coneData` parameter, information about the cone specified by the `cone` parameter. QuickDraw 3D allocates memory for the `TQ3ConeData` structure internally; you must call `Q3Cone_EmptyData` to dispose of that memory.

Q3Cone_SetData

You can use the `Q3Cone_SetData` function to set the data that defines a cone and its attributes.

```
TQ3Status Q3Cone_SetData (
                    TQ3GeometryObject cone,
                    const TQ3ConeData *coneData);
```

`cone` A cone.

`coneData` A pointer to a `TQ3ConeData` structure.

DESCRIPTION

The `Q3Cone_SetData` function sets the data associated with the cone specified by the `cone` parameter to the data specified by the `coneData` parameter.

Q3Cone_EmptyData

You can use the `Q3Cone_EmptyData` function to release the memory occupied by the data structure returned by a previous call to `Q3Cone_GetData`.

```
TQ3Status Q3Cone_EmptyData (TQ3ConeData *coneData);
```

`coneData` A pointer to a `TQ3ConeData` structure.

DESCRIPTION

The `Q3Cone_EmptyData` function releases the memory occupied by the `TQ3ConeData` structure pointed to by the `coneData` parameter; that memory was allocated by a previous call to `Q3Cone_GetData`.

Q3Cone_GetOrigin

You can use the `Q3Cone_GetOrigin` function to get the origin of a cone.

```
TQ3Status Q3Cone_GetOrigin (TQ3GeometryObject cone, TQ3Point3D *origin);
```

`cone` A cone.

`origin` On exit, the origin of the specified cone.

DESCRIPTION

The `Q3Cone_GetOrigin` function returns, in the `origin` parameter, the origin of the cone specified by the `cone` parameter.

Q3Cone_SetOrigin

You can use the `Q3Cone_SetOrigin` function to set the origin of a cone.

```
TQ3Status Q3Cone_SetOrigin (
                    TQ3GeometryObject cone,
                    const TQ3Point3D *origin);
```

`cone` A cone.

`origin` The desired origin of the specified cone.

DESCRIPTION

The `Q3Cone_SetOrigin` function sets the origin of the cone specified by the `cone` parameter to that specified in the `origin` parameter.

Q3Cone_GetOrientation

You can use the `Q3Cone_GetOrientation` function to get the orientation of a cone.

```
TQ3Status Q3Cone_GetOrientation (
                    TQ3GeometryObject cone,
                    TQ3Vector3D *orientation);
```

`cone` A cone.

`orientation` On exit, the orientation of the specified cone.

DESCRIPTION

The `Q3Cone_GetOrientation` function returns, in the `orientation` parameter, the orientation of the cone specified by the `cone` parameter.

Q3Cone_SetOrientation

You can use the `Q3Cone_SetOrientation` function to set the orientation of a cone.

```
TQ3Status Q3Cone_SetOrientation (
                    TQ3GeometryObject cone,
                    const TQ3Vector3D *orientation);
```

`cone` A cone.

`orientation` The desired orientation of the specified cone.

DESCRIPTION

The `Q3Cone_SetOrientation` function sets the orientation of the cone specified by the `cone` parameter to that specified in the `orientation` parameter.

Q3Cone_GetMajorRadius

You can use the `Q3Cone_GetMajorRadius` function to get the major radius of a cone.

```
TQ3Status Q3Cone_GetMajorRadius (
                    TQ3GeometryObject cone,
                    TQ3Vector3D *majorRadius);
```

`cone` A cone.

`majorRadius` On exit, the major radius of the specified cone.

DESCRIPTION

The `Q3Cone_GetMajorRadius` function returns, in the `majorRadius` parameter, the major radius of the cone specified by the `cone` parameter.

Q3Cone_SetMajorRadius

You can use the `Q3Cone_SetMajorRadius` function to set the major radius of a cone.

```
TQ3Status Q3Cone_SetMajorRadius (
                    TQ3GeometryObject cone,
                    const TQ3Vector3D *majorRadius);
```

`cone` A cone.

`majorRadius` The desired major radius of the specified cone.

DESCRIPTION

The `Q3Cone_SetMajorRadius` function sets the major radius of the cone specified by the `cone` parameter to that specified in the `majorRadius` parameter.

Q3Cone_GetMinorRadius

You can use the `Q3Cone_GetMinorRadius` function to get the minor radius of a cone.

```
TQ3Status Q3Cone_GetMinorRadius (
                    TQ3GeometryObject cone,
                    TQ3Vector3D *minorRadius);
```

`cone` A cone.

`minorRadius` On exit, the minor radius of the specified cone.

DESCRIPTION

The `Q3Cone_GetMinorRadius` function returns, in the `minorRadius` parameter, the minor radius of the cone specified by the `cone` parameter.

Q3Cone_SetMinorRadius

You can use the `Q3Cone_SetMinorRadius` function to set the minor radius of a cone.

```
TQ3Status Q3Cone_SetMinorRadius (
                    TQ3GeometryObject cone,
                    const TQ3Vector3D *minorRadius);
```

`cone` A cone.

`minorRadius` The desired minor radius of the specified cone.

DESCRIPTION

The `Q3Cone_SetMinorRadius` function sets the minor radius of the cone specified by the `cone` parameter to that specified in the `minorRadius` parameter.

Q3Cone_GetCaps

You can use the `Q3Cone_GetCaps` function to get the cap style of a cone.

```
TQ3Status Q3Cone_GetCaps (TQ3GeometryObject cone, TQ3EndCap *caps);
```

`cone` A cone.

`caps` On exit, the cap style of the specified cone.

DESCRIPTION

The `Q3Cone_GetCaps` function returns, in the `caps` parameter, the current cap style of the cone specified by the `cone` parameter.

Q3Cone_SetCaps

You can use the `Q3Cone_SetCaps` function to set the cap style of a cone.

```
TQ3Status Q3Cone_SetCaps (TQ3GeometryObject cone, TQ3EndCap caps);
```

`cone` A cone.

`caps` The desired style of cone cap.

DESCRIPTION

The `Q3Cone_SetCaps` function sets the cap of the cone specified by the `cone` parameter to the style indicated by the `caps` parameter.

Q3Cone_GetFaceAttributeSet

You can use the `Q3Cone_GetFaceAttributeSet` function to get the face attribute set of a cone.

```
TQ3Status Q3Cone_GetFaceAttributeSet (
                    TQ3GeometryObject cone,
                    TQ3AttributeSet *faceAttributeSet);
```

`cone` A cone.

`faceAttributeSet`
On exit, the attribute set of the face of the specified cone.

DESCRIPTION

The `Q3Cone_GetFaceAttributeSet` function returns, in the `faceAttributeSet` parameter, the attribute set of the face of the cone specified by the `cone` parameter. The reference count of the set is incremented.

Q3Cone_SetFaceAttributeSet

You can use the `Q3Cone_SetFaceAttributeSet` function to set the face attribute set of a cone.

```
TQ3Status Q3Cone_SetFaceAttributeSet (
                    TQ3GeometryObject cone,
                    TQ3AttributeSet faceAttributeSet);
```

`cone` A cone.

`faceAttributeSet`
The desired attribute set of the face of the specified cone.

DESCRIPTION

The `Q3Cone_SetFaceAttributeSet` function sets the attribute set of the face of the cone specified by the `cone` parameter to that specified in the `faceAttributeSet` parameter.

Q3Cone_GetBottomAttributeSet

You can use the `Q3Cone_GetBottomAttributeSet` function to get the bottom attribute set of a cone.

```
TQ3Status Q3Cone_GetBottomAttributeSet (
                    TQ3GeometryObject cone,
                    TQ3AttributeSet *bottomAttributeSet);
```

`cone` A cone.

`bottomAttributeSet`
On exit, the attribute set of the bottom of the specified cone.

DESCRIPTION

The `Q3Cone_GetBottomAttributeSet` function returns, in the `bottomAttributeSet` parameter, the attribute set of the bottom of the cone specified by the `cone` parameter. The reference count of the set is incremented.

Q3Cone_SetBottomAttributeSet

You can use the `Q3Cone_SetBottomAttributeSet` function to set the bottom attribute set of a cone.

```
TQ3Status Q3Cone_SetBottomAttributeSet (
                    TQ3GeometryObject cone,
                    TQ3AttributeSet bottomAttributeSet);
```

`cone` A cone.

`bottomAttributeSet`
The desired attribute set of the bottom of the specified cone.

DESCRIPTION

The `Q3Cone_SetBottomAttributeSet` function sets the attribute set of the bottom of the cone specified by the `cone` parameter to that specified in the `bottomAttributeSet` parameter.

Creating and Editing Tori

QuickDraw 3D provides routines that you can use to create and manipulate tori. See "Tori" (page 326) for the definition of a torus.

Q3Torus_New

You can use the `Q3Torus_New` function to create a new torus.

```
TQ3GeometryObject Q3Torus_New (const TQ3TorusData *torusData);
```

`torusData` A pointer to a `TQ3TorusData` structure.

DESCRIPTION

The `Q3Torus_New` function returns, as its function result, a new torus having the shape attributes specified by the `torusData` parameter. If a new torus could not be created, `Q3Torus_New` returns the value `NULL`.

Q3Torus_Submit

You can use the `Q3Torus_Submit` function to submit an immediate torus for drawing, picking, bounding, or writing.

```
TQ3Status Q3Torus_Submit (
                    const TQ3TorusData *torusData,
                    TQ3ViewObject view);
```

`torusData` A pointer to a `TQ3TorusData` structure.

`view` A view.

DESCRIPTION

The `Q3Torus_Submit` function submits for drawing, picking, bounding, or writing the immediate torus whose shape and attribute set are specified by the `torusData` parameter. The torus is drawn, picked, bounded, or written according to the view characteristics specified in the `view` parameter.

SPECIAL CONSIDERATIONS

You should call this function only in a submitting loop.

Q3Torus_GetData

You can use the `Q3Torus_GetData` function to get the data that defines a torus and its attributes.

```
TQ3Status Q3Torus_GetData (
                    TQ3GeometryObject torus,
                    TQ3TorusData *torusData);
```

`torus` A torus.

`torusData` On exit, a pointer to a `TQ3TorusData` structure that contains information about the torus specified by the `torus` parameter.

DESCRIPTION

The `Q3Torus_GetData` function returns, through the `torusData` parameter, information about the torus specified by the `torus` parameter. QuickDraw 3D allocates memory for the `TQ3TorusData` structure internally; you must call `Q3Torus_EmptyData` to dispose of that memory.

Q3Torus_SetData

You can use the `Q3Torus_SetData` function to set the data that defines a torus and its attributes.

```
TQ3Status Q3Torus_SetData (
                    TQ3GeometryObject torus,
                    const TQ3TorusData *torusData);
```

`torus` A torus.

`torusData` A pointer to a `TQ3TorusData` structure.

DESCRIPTION

The `Q3Torus_SetData` function sets the data associated with the torus specified by the `torus` parameter to the data specified by the `torusData` parameter.

Q3Torus_EmptyData

You can use the `Q3Torus_EmptyData` function to release the memory occupied by the data structure returned by a previous call to `Q3Torus_GetData`.

```
TQ3Status Q3Torus_EmptyData (TQ3TorusData *torusData);
```

`torusData` A pointer to a `TQ3TorusData` structure.

DESCRIPTION

The `Q3Torus_EmptyData` function releases the memory occupied by the `TQ3TorusData` structure pointed to by the `torusData` parameter; that memory was allocated by a previous call to `Q3Torus_GetData`.

Q3Torus_GetOrigin

You can use the `Q3Torus_GetOrigin` function to get the origin of a torus.

```
TQ3Status Q3Torus_GetOrigin (
                    TQ3GeometryObject torus,
                    TQ3Point3D *origin);
```

`torus` A torus.

`origin` On exit, the origin of the specified torus.

DESCRIPTION

The `Q3Torus_GetOrigin` function returns, in the `origin` parameter, the origin of the torus specified by the `torus` parameter.

Q3Torus_SetOrigin

You can use the `Q3Torus_SetOrigin` function to set the origin of a torus.

```
TQ3Status Q3Torus_SetOrigin (
                    TQ3GeometryObject torus,
                    const TQ3Point3D *origin);
```

`torus` A torus.

`origin` The desired origin of the specified torus.

DESCRIPTION

The `Q3Torus_SetOrigin` function sets the origin of the torus specified by the `torus` parameter to the point specified in the `origin` parameter.

Q3Torus_GetOrientation

You can use the `Q3Torus_GetOrientation` function to get the orientation of a torus.

```
TQ3Status Q3Torus_GetOrientation (
                    TQ3GeometryObject torus,
                    TQ3Vector3D *orientation);
```

`torus` A torus.

`orientation` On exit, the orientation of the specified torus.

DESCRIPTION

The `Q3Torus_GetOrientation` function returns, in the `orientation` parameter, the orientation of the torus specified by the `torus` parameter.

Q3Torus_SetOrientation

You can use the `Q3Torus_SetOrientation` function to set the orientation of a torus.

```
TQ3Status Q3Torus_SetOrientation (
                    TQ3GeometryObject torus,
                    const TQ3Vector3D *orientation);
```

`torus` A torus.

`orientation` The desired orientation of the specified torus.

DESCRIPTION

The `Q3Torus_SetOrientation` function sets the orientation of the torus specified by the `torus` parameter to the vector specified in the `orientation` parameter.

Q3Torus_GetMajorRadius

You can use the `Q3Torus_GetMajorRadius` function to get the major radius of a torus.

```
TQ3Status Q3Torus_GetMajorRadius (
                    TQ3GeometryObject torus,
                    TQ3Vector3D *majorRadius);
```

`torus` A torus.

`majorRadius` On exit, the major radius of the specified torus.

DESCRIPTION

The `Q3Torus_GetMajorRadius` function returns, in the `majorRadius` parameter, the major radius of the torus specified by the `torus` parameter.

Q3Torus_SetMajorRadius

You can use the `Q3Torus_SetMajorRadius` function to set the major radius of a torus.

```
TQ3Status Q3Torus_SetMajorRadius (
                    TQ3GeometryObject torus,
                    const TQ3Vector3D *majorRadius);
```

`torus` A torus.

`majorRadius` The desired major radius of the specified torus.

DESCRIPTION

The `Q3Torus_SetMajorRadius` function sets the major radius of the torus specified by the `torus` parameter to the vector specified in the `majorRadius` parameter.

Q3Torus_GetMinorRadius

You can use the `Q3Torus_GetMinorRadius` function to get the minor radius of a torus.

```
TQ3Status Q3Torus_GetMinorRadius (
                    TQ3GeometryObject torus,
                    TQ3Vector3D *minorRadius);
```

`torus` A torus.

`minorRadius` On exit, the minor radius of the specified torus.

DESCRIPTION

The `Q3Torus_GetMinorRadius` function returns, in the `minorRadius` parameter, the minor radius of the torus specified by the `torus` parameter.

Q3Torus_SetMinorRadius

You can use the `Q3Torus_SetMinorRadius` function to set the minor radius of a torus.

```
TQ3Status Q3Torus_SetMinorRadius (
                    TQ3GeometryObject torus,
                    const TQ3Vector3D *minorRadius);
```

`torus` A torus.

`minorRadius` The desired minor radius of the specified torus.

DESCRIPTION

The `Q3Torus_SetMinorRadius` function sets the minor radius of the torus specified by the `torus` parameter to the vector specified in the `minorRadius` parameter.

Q3Torus_GetRatio

You can use the `Q3Torus_GetRatio` function to get the ratio of a torus.

```
TQ3Status Q3Torus_GetRatio (
                    TQ3GeometryObject torus,
                    float *ratio);
```

`torus` A torus.

`ratio` On exit, the ratio of the specified torus.

DESCRIPTION

The `Q3Torus_GetRatio` function returns, in the `ratio` parameter, the ratio of the torus specified by the `torus` parameter.

Q3Torus_SetRatio

You can use the `Q3Torus_SetRatio` function to set the ratio of a torus.

```
TQ3Status Q3Torus_SetRatio (
                    TQ3GeometryObject torus,
                    float ratio);
```

`torus` A torus.

`ratio` The desired ratio of the specified torus.

DESCRIPTION

The `Q3Torus_SetRatio` function sets the ratio of the torus specified by the `torus` parameter to the value passed in the `ratio` parameter.

Creating and Editing Bitmap Markers

QuickDraw 3D provides routines that you can use to create and manipulate bitmap markers. See "Markers" (page 329) for the definition of a bitmap marker.

Q3Marker_New

You can use the `Q3Marker_New` function to create a new marker.

```
TQ3GeometryObject Q3Marker_New (const TQ3MarkerData *markerData);
```

`markerData` A pointer to a `TQ3MarkerData` structure.

DESCRIPTION

The `Q3Marker_New` function returns, as its function result, a new marker having the location, shape, offset, and attributes specified by the `markerData` parameter. If a new marker could not be created, `Q3Marker_New` returns the value `NULL`.

Q3Marker_Submit

You can use the `Q3Marker_Submit` function to submit an immediate marker for drawing, picking, bounding, or writing.

```
TQ3Status Q3Marker_Submit (
                    const TQ3MarkerData *markerData,
                    TQ3ViewObject view);
```

`markerData` A pointer to a `TQ3MarkerData` structure.

`view` A view.

DESCRIPTION

The `Q3Marker_Submit` function submits for drawing, picking, bounding, or writing the immediate marker whose location, shape, offset, and attribute set are specified by the `markerData` parameter. The marker is drawn, picked, bounded, or written according to the view characteristics specified in the `view` parameter.

SPECIAL CONSIDERATIONS

You should call this function only in a submitting loop.

Q3Marker_GetData

You can use the `Q3Marker_GetData` function to get the data associated with a marker.

```
TQ3Status Q3Marker_GetData (
                    TQ3GeometryObject marker,
                    TQ3MarkerData *markerData);
```

`marker` A marker.

`markerData` On exit, a pointer to a `TQ3MarkerData` structure.

DESCRIPTION

The `Q3Marker_GetData` function returns, through the `markerData` parameter, information about the marker specified by the `marker` parameter. QuickDraw 3D allocates memory for the `TQ3MarkerData` structure internally; you must call `Q3Marker_EmptyData` to dispose of that memory.

Q3Marker_SetData

You can use the `Q3Marker_SetData` function to set the data associated with a marker.

```
TQ3Status Q3Marker_SetData (
                    TQ3GeometryObject marker,
                    const TQ3MarkerData *markerData);
```

`marker` A marker.

`markerData` A pointer to a `TQ3MarkerData` structure.

DESCRIPTION

The `Q3Marker_SetData` function sets the data associated with the marker specified by the `marker` parameter to the data specified by the `markerData` parameter.

Q3Marker_EmptyData

You can use the `Q3Marker_EmptyData` function to release the memory occupied by the data structure returned by a previous call to `Q3Marker_GetData`.

```
TQ3Status Q3Marker_EmptyData (TQ3MarkerData *markerData);
```

`markerData` A pointer to a `TQ3MarkerData` structure.

DESCRIPTION

The `Q3Marker_EmptyData` function releases the memory occupied by the `TQ3MarkerData` structure pointed to by the `markerData` parameter; that memory was allocated by a previous call to `Q3Marker_GetData`.

Q3Marker_GetPosition

You can use the `Q3Marker_GetPosition` function to get the position of a marker.

```
TQ3Status Q3Marker_GetPosition (
                    TQ3GeometryObject marker,
                    TQ3Point3D *location);
```

marker	A marker.
location	On exit, the location of the specified marker.

DESCRIPTION

The Q3Marker_GetPosition function returns, in the location parameter, the location of the marker specified by the marker parameter.

Q3Marker_SetPosition

You can use the Q3Marker_SetPosition function to set the position of a marker.

```
TQ3Status Q3Marker_SetPosition (
                    TQ3GeometryObject marker,
                    const TQ3Point3D *location);
```

marker	A marker.
location	The desired location of the specified marker.

DESCRIPTION

The Q3Marker_SetPosition function sets the position of the marker specified by the marker parameter to the point specified in the position parameter.

Q3Marker_GetXOffset

You can use the Q3Marker_GetXOffset function to get the horizontal offset of a marker.

```
TQ3Status Q3Marker_GetXOffset (TQ3GeometryObject marker, long *xOffset);
```

marker	A marker.
xOffset	On exit, the horizontal offset of the specified marker.

DESCRIPTION

The `Q3Marker_GetXOffset` function returns, in the `xOffset` parameter, the horizontal offset of the marker specified by the `marker` parameter.

Q3Marker_SetXOffset

You can use the `Q3Marker_SetXOffset` function to set the horizontal offset of a marker.

```
TQ3Status Q3Marker_SetXOffset (TQ3GeometryObject marker, long xOffset);
```

`marker` A marker.

`xOffset` The desired horizontal offset of the specified marker.

DESCRIPTION

The `Q3Marker_SetXOffset` function sets the horizontal offset of the marker specified by the `marker` parameter to the value specified in the `xOffset` parameter.

Q3Marker_GetYOffset

You can use the `Q3Marker_GetYOffset` function to get the vertical offset of a marker.

```
TQ3Status Q3Marker_GetYOffset (TQ3GeometryObject marker, long *yOffset);
```

`marker` A marker.

`yOffset` On exit, the vertical offset of the specified marker.

DESCRIPTION

The `Q3Marker_GetYOffset` function returns, in the `yOffset` parameter, the vertical offset of the marker specified by the `marker` parameter.

Q3Marker_SetYOffset

You can use the `Q3Marker_SetYOffset` function to set the vertical offset of a marker.

```
TQ3Status Q3Marker_SetYOffset (TQ3GeometryObject marker, long yOffset);
```

`marker` A marker.

`yOffset` The desired vertical offset of the specified marker.

DESCRIPTION

The `Q3Marker_SetYOffset` function sets the vertical offset of the marker specified by the `marker` parameter to the value specified in the `yOffset` parameter.

Q3Marker_GetBitmap

You can use the `Q3Marker_GetBitmap` function to get the bitmap of a marker.

```
TQ3Status Q3Marker_GetBitmap (
                    TQ3GeometryObject marker,
                    TQ3Bitmap *bitmap);
```

`marker` A marker.

`bitmap` On exit, the bitmap of the specified marker.

DESCRIPTION

The `Q3Marker_GetBitmap` function returns, in the `bitmap` parameter, a copy of the bitmap of the marker specified by the `marker` parameter. `Q3Marker_GetBitmap` allocates memory internally for the returned bitmap; when you're done using the bitmap, you should call the `Q3Bitmap_Empty` function to dispose of that memory.

Q3Marker_SetBitmap

You can use the `Q3Marker_SetBitmap` function to set the bitmap of a marker.

```
TQ3Status Q3Marker_SetBitmap (
                    TQ3GeometryObject marker,
                    const TQ3Bitmap *bitmap);
```

`marker` A marker.

`bitmap` The desired bitmap of the specified marker.

DESCRIPTION

The `Q3Marker_SetBitmap` function sets the bitmap of the marker specified by the `marker` parameter to that specified in the `bitmap` parameter. `Q3Marker_SetBitmap` copies the bitmap to internal QuickDraw 3D memory, so you can dispose of the specified bitmap after calling `Q3Marker_SetBitmap`.

Creating and Editing Pixmap Markers

QuickDraw 3D provides routines that you can use to create and manipulate pixmap markers. See “Markers” (page 329) for the definition of a pixmap marker.

Q3PixmapMarker_New

You can use the `Q3PixmapMarker_New` function to create a new pixmap marker.

```
TQ3GeometryObject Q3PixmapMarker_New (
                    const TQ3PixmapMarkerData *pixmapMarkerData);
```

`pixmapMarkerData`
A pointer to a `TQ3PixmapMarkerData` structure.

DESCRIPTION

The `Q3PixmapMarker_New` function returns, as its function result, a new pixmap marker having the position, shape, offset, and attributes specified by the `pixmapMarkerData` parameter. If a new pixmap marker could not be created, `Q3PixmapMarker_New` returns the value `NULL`.

Q3PixmapMarker_Submit

You can use the `Q3PixmapMarker_Submit` function to submit an immediate pixmap marker for drawing, picking, bounding, or writing.

```
TQ3Status Q3PixmapMarker_Submit (
                    const TQ3PixmapMarkerData *pixmapMarkerData,
                    TQ3ViewObject view);
```

`pixmapMarkerData`
A pointer to a `TQ3PixmapMarkerData` structure.

`view` A view.

DESCRIPTION

The `Q3PixmapMarker_Submit` function submits for drawing, picking, bounding, or writing the immediate pixmap marker whose position, shape, offset, and attribute set are specified by the `pixmapMarkerData` parameter. The pixmap marker is drawn, picked, bounded, or written according to the view characteristics specified in the `view` parameter.

SPECIAL CONSIDERATIONS

You should call this function only in a submitting loop.

Q3PixmapMarker_GetData

You can use the `Q3PixmapMarker_GetData` function to get the data associated with a pixmap marker.

```
TQ3Status Q3PixmapMarker_GetData (
                    TQ3GeometryObject geometry,
                    TQ3PixmapMarkerData *pixmapMarkerData);
```

`geometry` A pixmap marker.

`pixmapMarkerData`
On exit, a pointer to a `TQ3PixmapMarkerData` structure.

DESCRIPTION

The `Q3PixmapMarker_GetData` function returns, through the `pixmapMarkerData` parameter, information about the pixmap marker specified by the `geometry` parameter. QuickDraw 3D allocates memory for the `TQ3PixmapMarkerData` structure internally; you must call `Q3PixmapMarker_EmptyData` to dispose of that memory.

Q3PixmapMarker_SetData

You can use the `Q3PixmapMarker_SetData` function to set the data associated with a pixmap marker.

```
TQ3Status Q3PixmapMarker_SetData (
                    TQ3GeometryObject geometry,
                    const TQ3PixmapMarkerData *pixmapMarkerData);
```

`geometry` A pixmap marker.

`pixmapMarkerData`
A pointer to a `TQ3PixmapMarkerData` structure.

DESCRIPTION

The `Q3PixmapMarker_SetData` function sets the data associated with the pixmap marker specified by the `geometry` parameter to the data specified by the `pixmapMarkerData` parameter.

Q3PixmapMarker_EmptyData

You can use the `Q3PixmapMarker_EmptyData` function to release the memory occupied by the data structure returned by a previous call to `Q3PixmapMarker_GetData`.

```
TQ3Status Q3PixmapMarker_EmptyData  (
                    TQ3PixmapMarkerData *pixmapMarkerData);
```

`pixmapMarkerData`
A pointer to a `TQ3PixmapMarkerData` structure.

DESCRIPTION

The `Q3PixmapMarker_EmptyData` function releases the memory occupied by the `TQ3PixmapMarkerData` structure pointed to by the `pixmapMarkerData` parameter; that memory was allocated by a previous call to `Q3PixmapMarker_GetData`.

Q3PixmapMarker_GetPosition

You can use the `Q3PixmapMarker_GetPosition` function to get the position of a pixmap marker.

```
TQ3Status Q3PixmapMarker_GetPosition (
                    TQ3GeometryObject pixmapMarker,
                    TQ3Point3D *position);
```

`pixmapMarker` A pixmap marker.

`position` On exit, the position of the specified pixmap marker.

DESCRIPTION

The `Q3PixmapMarker_GetPosition` function returns, in the `position` parameter, the location of the pixmap marker specified by the `pixmapMarker` parameter.

Q3PixmapMarker_SetPosition

You can use the `Q3PixmapMarker_SetPosition` function to set the position of a pixmap marker.

```
TQ3Status Q3PixmapMarker_SetPosition (
                    TQ3GeometryObject pixmapMarker,
                    const TQ3Point3D *position);
```

`pixmapMarker` A pixmap marker.

`position` The desired position of the specified pixmap marker.

DESCRIPTION

The `Q3PixmapMarker_SetPosition` function sets the position of the pixmap marker specified by the `pixmapMarker` parameter to the point specified in the `position` parameter.

Q3PixmapMarker_GetXOffset

You can use the `Q3PixmapMarker_GetXOffset` function to get the horizontal offset of a pixmap marker.

```
TQ3Status Q3PixmapMarker_GetXOffset (
                    TQ3GeometryObject pixmapMarker,
                    long *xOffset);
```

`pixmapMarker` A pixmap marker.

`xOffset` On exit, the horizontal offset of the specified pixmap marker.

DESCRIPTION

The `Q3PixmapMarker_GetXOffset` function returns, in the `xOffset` parameter, the horizontal offset of the pixmap marker specified by the `pixmapMarker` parameter.

Q3PixmapMarker_SetXOffset

You can use the `Q3PixmapMarker_SetXOffset` function to set the horizontal offset of a pixmap marker.

```
TQ3Status Q3PixmapMarker_SetXOffset (
                    TQ3GeometryObject pixmapMarker,
                    long xOffset);
```

`pixmapMarker` A pixmap marker.

`xOffset` The desired horizontal offset of the specified pixmap marker.

DESCRIPTION

The `Q3PixmapMarker_SetXOffset` function sets the horizontal offset of the pixmap marker specified by the `pixmapMarker` parameter to the value specified in the `xOffset` parameter.

Q3PixmapMarker_GetYOffset

You can use the `Q3PixmapMarker_GetYOffset` function to get the vertical offset of a pixmap marker.

```
TQ3Status Q3PixmapMarker_GetYOffset (
                    TQ3GeometryObject pixmapMarker,
                    long *yOffset);
```

`pixmapMarker` A pixmap marker.

`yOffset` On exit, the vertical offset of the specified pixmap marker.

DESCRIPTION

The `Q3PixmapMarker_GetYOffset` function returns, in the `yOffset` parameter, the vertical offset of the pixmap marker specified by the `pixmapMarker` parameter.

Q3PixmapMarker_SetYOffset

You can use the `Q3PixmapMarker_SetYOffset` function to set the vertical offset of a pixmap marker.

```
TQ3Status Q3PixmapMarker_SetYOffset (
                    TQ3GeometryObject pixmapMarker,
                    long yOffset);
```

`pixmapMarker` A pixmap marker.

`yOffset` The desired vertical offset of the specified pixmap marker.

DESCRIPTION

The `Q3PixmapMarker_SetYOffset` function sets the vertical offset of the pixmap marker specified by the `pixmapMarker` parameter to the value specified in the `yOffset` parameter.

Q3PixmapMarker_GetPixmap

You can use the `Q3PixmapMarker_GetPixmap` function to get the pixmap of a pixmap marker.

```
TQ3Status Q3PixmapMarker_GetPixmap (
                    TQ3GeometryObject pixmapMarker,
                    TQ3StoragePixmap *pixmap);
```

`pixmapMarker` A pixmap marker.

`pixmap` On exit, the pixmap of the specified pixmap marker.

DESCRIPTION

The `Q3PixmapMarker_GetPixmap` function returns, in the `pixmap` parameter, a pointer to the pixmap associated with the pixmap marker specified by the `pixmapMarker` parameter.

Q3PixmapMarker_SetPixmap

You can use the `Q3PixmapMarker_SetPixmap` function to set the pixmap of a pixmap marker.

```
TQ3Status Q3PixmapMarker_SetPixmap (
                    TQ3GeometryObject pixmapMarker,
                    const TQ3StoragePixmap *pixmap);
```

`pixmapMarker` A pixmap marker.

`pixmap` The desired pixmap of the specified pixmap marker.

DESCRIPTION

The `Q3PixmapMarker_SetPixmap` function sets the pixmap of the pixmap marker specified by the `pixmapMarker` parameter to that specified in the `pixmap` parameter. `Q3PixmapMarker_SetPixmap` copies the pixmap to internal QuickDraw 3D memory, so you can dispose of the specified pixmap after calling `Q3PixmapMarker_SetPixmap`.

Managing Bitmaps

QuickDraw 3D provides routines that you can use to dispose of the memory occupied by a bitmap and to determine the size of the memory occupied by a bitmap.

Q3Bitmap_Empty

You can use the `Q3Bitmap_Empty` function to release the memory occupied by a bitmap that was allocated by a previous call to some QuickDraw 3D routine.

```
TQ3Status Q3Bitmap_Empty (TQ3Bitmap *bitmap);
```

`bitmap` A pointer to a bitmap obtained by a previous call to some QuickDraw 3D routine such as `Q3Marker_GetData`, `Q3Marker_GetBitmap`, `Q3DrawContext_GetMask`, or `Q3ViewHints_GetMask`.

DESCRIPTION

The `Q3Bitmap_Empty` function releases the memory occupied by the bitmap pointed to by the `bitmap` parameter; that memory must have been allocated by a previous call to some QuickDraw 3D routine (for example, `Q3Marker_GetBitmap`). You should not call `Q3Bitmap_Empty` to deallocate bitmaps that you allocated yourself.

Q3Bitmap_GetImageSize

You can use the `Q3Bitmap_GetImageSize` function to determine how much memory is occupied by a bitmap of a particular size.

```
unsigned long Q3Bitmap_GetImageSize (
                    unsigned long width,
                    unsigned long height);
```

`width` The width, in bits, of a bitmap.

`height` The height of a bitmap.

DESCRIPTION

The `Q3Bitmap_GetImageSize` function returns, as its function result, the size, in bytes, of the smallest block of memory required to hold a bitmap having a width and height specified by the `width` and `height` parameters, respectively.

Geometry Errors, Warnings, and Notices

The following is a list of errors, warnings, and notices that geometry routines can return. A list of general QuickDraw 3D errors is given in "QuickDraw 3D Errors, Warnings, and Notices" (page 87).

```
kQ3ErrorDegenerateGeometry
kQ3ErrorGeometryInsufficientNumberOfPoints
kQ3WarningVector3DNotUnitLength
kQ3WarningQuaternionEntriesAreZero
```

```
kQ3NoticeMeshVertexHasNoComponent
kQ3NoticeMeshInvalidVertexFacePair
kQ3NoticeMeshEdgeVertexDoNotCorrespond
kQ3NoticeMeshEdgeIsNotBoundary
```

CHAPTER 5

Attribute Objects

This chapter describes attribute objects (or attributes) and attribute sets. Attributes store information about the characteristics of the materials that make up the objects in a model. For example, you can attach an attribute to a geometric object that specifies the object's color. You can also attach an attribute to part of an object, for example to a vertex of a mesh. QuickDraw 3D provides a wide range of predefined attribute types, and you can define custom attribute types if you wish.

To use this chapter, you should already be familiar with the QuickDraw 3D class hierarchy, described in the chapter "QuickDraw 3D Objects" earlier in this book. To attach attribute sets to geometric objects, you should also be familiar with the routines described in the chapter "Geometric Objects" in this book.

This chapter begins by describing attributes and attribute sets. Then it shows how to create attribute sets and attach them to parts of a model. The section "Attribute Objects Reference," beginning on page 526 provides a complete description of attributes and attribute sets and of the routines you can use to create and manipulate them.

About Attribute Objects

An **attribute object** (or, more briefly, an **attribute**) is a type of QuickDraw 3D object that determines some of the characteristics of a model, such as the color of objects or parts of objects in the model, the transparency of objects, and so forth. In general, attributes define material properties of the surfaces of objects in a model.

An attribute is defined as an attribute type and some associated data. You apply an attribute to an object by creating an instance of a specific attribute type, defining its data, and then attaching it to the object. QuickDraw 3D defines

many types of attributes, including diffuse color, specular color, transparency color, surface normals, and surface tangents.

In general, however, attributes are not applied to objects individually. Instead, you usually create an **attribute set,** which is a collection of zero or more different attribute types and their associated data. For example, to create a transparent red triangle, you create an attribute set, add both color and transparency attributes to it, and then attach the attribute set to the triangle. An attribute set is of type `TQ3AttributeSet`, a type of `TQ3SetObject`.

Types of Attributes and Attribute Sets

QuickDraw 3D defines a large number of basic attribute types, which represent information such as surface color, transparency, parameterization, normal, tangent, and so forth. In addition, if the basic QuickDraw 3D attribute types are not sufficient for the needs of your application, you can define custom attribute types. For example, you might want to maintain information about the temperature over time of each point on the surface of an object. To do so, you can define a new attribute type and a data structure to hold the relevant information. You also need to define an **attribute metahandler,** which contains methods for handling your custom attribute data. (QuickDraw 3D defines metahandlers for all the basic attribute types.)

The basic attributes types are defined by constants. See "Attribute Types" (page 527) for a complete description of these attribute types.

```
typedef enum TQ3AttributeTypes {
    kQ3AttributeTypeNone                                    = 0,
    kQ3AttributeTypeSurfaceUV                               = 1,
    kQ3AttributeTypeShadingUV                               = 2,
    kQ3AttributeTypeNormal                                  = 3,
    kQ3AttributeTypeAmbientCoefficient                      = 4,
    kQ3AttributeTypeDiffuseColor                            = 5,
    kQ3AttributeTypeSpecularColor                           = 6,
    kQ3AttributeTypeSpecularControl                         = 7,
    kQ3AttributeTypeTransparencyColor                       = 8,
    kQ3AttributeTypeSurfaceTangent                          = 9,
    kQ3AttributeTypeHighlightState                          = 10,
    kQ3AttributeTypeSurfaceShader                           = 11,
    kQ3AttributeTypeNumTypes
} TQ3AttributeTypes;
```

You can attach a set of attributes to a view, to a group of objects, to a single geometric object, to a face of an object, or to a vertex of an object. In addition, you can attach edge and corner attributes to meshes. For each of these levels, QuickDraw 3D defines a set of **natural attributes.** For example, the surface normal attribute (which defines the normal vector at a point) makes no sense when applied to a view or a nonpolygonal geometric object. It does, however, make sense to include the surface normal attribute in a set of face or vertex attributes. Accordingly, the surface normal attribute is contained in the natural sets of attributes for faces and vertices, but not for views, groups, or nonpolygonal geometric objects. Table 5-1 lists the natural attributes that can be assigned to objects in the QuickDraw 3D object hierarchy.

IMPORTANT

You can, if you wish, include in the attribute set of any kind of object attributes that are not natural to that object. For instance, you can put a surface normal attribute into an attribute set attached to a view. You can then access that unnatural attribute in precisely the same way you access any other attribute in the set. The only difference between natural and unnatural attributes is that unnatural attributes in an attribute set are not inherited by objects lower down in the class hierarchy. See "Attribute Inheritance" (page 518) for details. ▲

Table 5-1 Natural sets of attributes for objects in a hierarchy

Object type	Natural attributes in the set
View object Group object Geometric object Face	kQ3AttributeTypeAmbientCoefficient kQ3AttributeTypeDiffuseColor kQ3AttributeTypeSpecularColor kQ3AttributeTypeSpecularControl kQ3AttributeTypeTransparencyColor kQ3AttributeTypeHighlightState kQ3AttributeTypeSurfaceShader
Vertex	kQ3AttributeTypeSurfaceUV kQ3AttributeTypeShadingUV kQ3AttributeTypeNormal kQ3AttributeTypeAmbientCoefficient kQ3AttributeTypeDiffuseColor kQ3AttributeTypeSpecularColor kQ3AttributeTypeSpecularControl kQ3AttributeTypeTransparencyColor kQ3AttributeTypeSurfaceTangent

Note

Surface normals assigned to faces are ignored by renderers, as are the surface normals that are computed geometrically from the points that make up the face. ◆

Attribute Inheritance

During the rendering of the objects in a view, attribute sets of objects higher in the view hierarchy are inherited by objects below them. For example, if the

attribute set of a view specifies a particular diffuse color, then all objects in that view are rendered with that diffuse color, *unless* some other attribute set overrides the color specified in the view attributes. That is, if some face of some object has an attribute set containing a different diffuse color, the face's diffuse color overrides the diffuse color that otherwise would have been inherited from the view attribute set.

Attribute inheritance always occurs in this order:

1. view
2. group
3. geometric object
4. face
5. mesh edge
6. vertex
7. mesh corner

In other words, view attributes are always inherited by all groups of objects in the model, unless a group contains overriding attributes. Similarly, any attributes assigned to a geometric object are inherited by all faces of the object, unless a face contains overriding attributes.

This attribute inheritance applies only to the natural attributes contained in any attribute set. If, for example, an attribute set of a view contains a surface normal attribute (which is *not* a natural attribute for view attribute sets), that attribute is not inherited by any objects lower down in the hierarchy.

If you define a custom attribute, you can specify whether you want that attribute to be inherited along the attribute inheritance path by including an attribute inheritance method in your attribute metahandler. See "Defining Custom Attribute Types" (page 522) for a sample attribute metahandler that specifies that the temperature attribute is to be inherited. If you do not supply an attribute inheritance method, QuickDraw 3D assumes you want no such inheritance for your custom attribute.

Using Attribute Objects

This section describes the basic capabilities that QuickDraw 3D provides to create and configure attribute sets. It also shows how to read the attributes in an attribute set and, if necessary, change those attributes. In general, it's very simple to create, configure, and modify attribute sets.

This section also shows how to define a custom attribute type. To do so, you need to provide definitions of the data associated with that attribute type and an attribute metahandler to define a set of attribute-handling methods. See "Defining Custom Attribute Types," beginning on page 522 for complete details.

Creating and Configuring Attribute Sets

You create a new attribute set by calling the `Q3AttributeSet_New` function. You configure the attribute set by adding the desired attributes to the set, using the `Q3AttributeSet_Add` function. Finally, you attach the configured attribute set to an object by calling an appropriate QuickDraw 3D routine. For example, to attach an attribute set to a vertex of a triangle, you call the function `Q3Triangle_SetVertexAttributeSet`. Listing 5-1 illustrates how to set the three vertices of a triangle to a specific diffuse color.

Listing 5-1 Creating and configuring a vertex attribute set

```
TQ3Status MySetTriangleVerticesDiffuseColor
                    (TQ3GeometryObject triangle, TQ3ColorRGB color)
{
    TQ3AttributeSet                 myAttrSet;      /*attribute set*/
    TQ3Status                       myResult;       /*result code*/
    unsigned long                   myIndex;        /*vertex index*/

    /*Create a new empty attribute set.*/
    myAttrSet = Q3AttributeSet_New();
    if (myAttrSet == NULL)
        return (kQ3Failure);
```

```
    /*Add the specified color attribute to the attribute set.*/
    myResult = Q3AttributeSet_Add
                (myAttrSet, kQ3AttributeTypeDiffuseColor, &color);
    if (myResult == kQ3Failure)
        return (kQ3Failure);

    /*Attach the attribute set to each triangle vertex.*/
    for (myIndex = 0; myIndex < 3; myIndex++) {
        myResult = Q3Triangle_SetVertexAttributeSet
                                        (triangle, myIndex, myAttrSet);
        if (myResult == kQ3Failure)
            return (kQ3Failure);
    }

    return (kQ3Success);
}
```

You can assign any number of different attribute types to a single attribute set. The function defined in Listing 5-1 assigns only one attribute—a diffuse color—to the new attribute set.

If you want to change the value of a certain attribute in an attribute set, you can simply overwrite the data associated with that attribute by calling `Q3AttributeSet_Add` once again. You can remove an attribute from an attribute set by calling `Q3AttributeSet_Clear`. To remove all attributes from an attribute set, you can call `Q3AttributeSet_Empty`.

Iterating Through an Attribute Set

QuickDraw 3D provides the `Q3AttributeSet_GetNextAttributeType` function that you can use to iterate through the attributes in an attribute set. To get the first attribute in an attribute set, pass the constant `kQ3AttributeTypeNone` to `Q3AttributeSet_GetNextAttributeType`. You can retrieve any subsequent attributes by successively calling `Q3AttributeSet_GetNextAttributeType`, which returns `kQ3AttributeTypeNone` when you reach the end of the list of attributes. Listing 5-2 illustrates how to use `Q3AttributeSet_GetNextAttributeType` to determine the number of attributes in an attribute set.

Listing 5-2 Counting the attributes in an attribute set

```
unsigned long MyCountAttributesInSet (TQ3AttributeSet mySet)
{
    unsigned long                   myCount;        /*attribute count*/
    TQ3AttributeType                myType;         /*attribute type*/
    TQ3Status                       myResult;       /*result code*/

    for (myCount = 0,
          myType = kQ3AttributeTypeNone,
          myResult =
                Q3AttributeSet_GetNextAttributeType(mySet, &myType);
          myType != kQ3AttributeTypeNone;
          myResult =
            Q3AttributeSet_GetNextAttributeType(mySet, &myType)) {
        myCount++;
    }

    return (myCount);
}
```

Notice that the `Q3AttributeSet_GetNextAttributeType` function returns a result code that indicates whether the call succeeded or failed. In general, the call fails only if the attribute set is invalid in some way.

Defining Custom Attribute Types

QuickDraw 3D allows you to define custom attribute types so that you can attach to a vertex (or face, or geometric object, or group, or view) types of data different from those associated with the basic attribute types defined by QuickDraw 3D. Once you have defined and registered your custom attribute type, you manipulate attributes of that type exactly as you manipulate the standard QuickDraw 3D attributes. For example, you add a custom attribute to an attribute set by calling `Q3AttributeSet_Add`, and you retrieve the data associated with a custom attribute by calling `Q3AttributeSet_Get`.

To define a custom attribute type, you first define the internal structure of the data associated with your custom attribute type. Then you must write an attribute metahandler to define a set of attribute-handling methods. QuickDraw 3D calls those methods at certain times to handle operations on attribute sets that contain your custom attribute. For example, when you call

`Q3Triangle_Write` to write a triangle to a file, QuickDraw 3D might need to call your attribute's handler to write your custom attribute data to the file.

Suppose that you want to define a custom attribute that contains data about temperature over time. You might use the `MyTemperatureData` structure, defined like this:

```
typedef struct MyTemperatureData {
    unsigned long               startTime;          /*starting time*/
    unsigned long               nTemps;             /*no. temps in
array*/
    float                       *temperatures;      /*array of temps*/
} MyTemperatureData;
```

Your attribute metahandler is an application-defined function that returns the addresses of the methods associated with the custom attribute type. A metahandler can define some or all of the methods indicated by these constants:

```
kQ3MethodTypeObjectDelete
kQ3MethodTypeObjectReadData
kQ3MethodTypeObjectTraverse
kQ3MethodTypeObjectWrite
kQ3MethodTypeElementCopyAdd
kQ3MethodTypeElementDelete
kQ3MethodTypeElementCopyDuplicate
kQ3MethodTypeElementCopyGet
kQ3MethodTypeElementCopyReplace
kQ3MethodTypeAttributeInterpolate
kQ3MethodTypeAttributeCopyInherit
kQ3MethodTypeAttributeInherit
```

Listing 5-3 defines a simple attribute metahandler. See "Defining an Object Metahandler," beginning on page 176 for a more complete description of metahandlers.

Listing 5-3 Reporting custom attribute methods

```
TQ3FunctionPointer MyTemperatureDataMetaHandler (TQ3MethodType methodType)
{
    switch (methodType) {
        case kQ3MethodTypeElementDelete:
            return (TQ3FunctionPointer) MyTemperatureDataDispose;
        case kQ3MethodTypeElementCopyReplace:
            return (TQ3FunctionPointer) MyTemperatureDataCopyReplace;
        case kQ3MethodTypeAttributeCopyInherit:
            return (TQ3FunctionPointer) kQ3True;
        case kQ3MethodTypeAttributeInherit:
            return (TQ3FunctionPointer) kQ3True;
        default:
            return (NULL);
    }
}
```

The `MyTemperatureDataMetaHandler` metahandler simply returns the appropriate function address, or `NULL` if the metahandler does not implement a particular method type. All the method types listed above are optional. (In fact, you don't need to specify a metahandler at all if you want QuickDraw 3D to use its default methods to handle your custom attribute type.)

The metahandler in Listing 5-3 installs the `MyTemperatureDataDispose` function as the custom attribute's dispose method, which QuickDraw 3D calls whenever you clear your custom attribute or replace an existing one. A dispose method is passed a pointer to the data associated with an attribute. Your dispose method should deallocate any storage you allocated, as shown in Listing 5-4.

Listing 5-4 Disposing of a custom attribute's data

```
TQ3Status MyTemperatureDataDispose (MyTemperatureData *tmpData)
{
    if (tData->temperatures != NULL) {
        free(tmpData->temperatures);
        tData->temperatures = NULL;
    }
    return kQ3Success;
}
```

If you do not define a dispose method, QuickDraw 3D automatically disposes of the block of data allocated when a custom attribute was added to an attribute set. If the data associated with a custom attribute is always of a fixed size and does not contain any pointers to other data that needs to be disposed of, you do not need to define a dispose or copy method.

The metahandler in Listing 5-3 installs the `MyTemperatureDataCopyReplace` function as the custom attribute's copy method. A copy method is passed two pointers, specifying the source and target addresses of the data to copy. Listing 5-5 shows a simple copy method.

Listing 5-5 Copying a custom attribute's data

```
TQ3Status MyTemperatureDataCopyReplace
            (const MyTemperatureData *src, MyTemperatureData *dst)
{
    float                           *temp;

    if (dst->nTemps != src->nTemps) {
        temp = realloc(dst->temperatures, nTemps * sizeof(float));
        if (temp == NULL)
            return (kQ3Failure);
    }
    dst->startTime      = src->startTime;
    dst->nTemps         = src->nTemps;
    dst->temperatures = temp;

    memcpy(temp, dst->temperatures, dst->nTemps * sizeof(float));

    return (kQ3Success);
}
```

If you do not define a copy method, QuickDraw 3D automatically copies the block of data using a default memory copy method.

The inherit method simply requests a Boolean value that indicates whether you want your custom attribute to be inherited down the class hierarchy. You should return `kQ3True` if you want your attribute to be inherited or `kQ3False` if not.

Before you can use a custom attribute type, you need to register your attribute metahandler with QuickDraw 3D by calling the `Q3AttributeClass_Register` function. You might execute the `MyStartUpQuickDraw3D` function defined in Listing 5-6 at application startup time.

Listing 5-6 Initializing QuickDraw 3D and registering a custom attribute type

```
TQ3AttributeType                    gAttributeType_Temperature;

void MyStartUpQuickDraw3D (void)
{
    TQ3ObjectClass                  myAttrib;

    if (Q3Initialize() == kQ3Failure)       /*initialize QuickDraw 3D*/
        MyFailRoutine();
                                        /*register attribute type*/
    myAttrib = Q3AttributeClass_Register(
                                    gAttributeTypeTemperature,
                                    “MyCompany:SurfWorks:Temperature”,
                                    sizeof(MyTemperatureData),
                                    MyTemperatureData_MetaHandler);

    if (myAttrib == kQ3ObjectTypeInvalid)
        MyFailRoutine();
}
```

Attribute Objects Reference

This section describes the constants and routines that you can use to manage an object’s attributes and attribute sets.

Constants

This section describes the constants that you use to define attribute types.

Attribute Types

Every attribute has a unique attribute type. QuickDraw 3D defines a large number of attribute types, and your application can define additional attribute types. Attribute types are defined by constants. Attribute type values greater than 0 are reserved for use by QuickDraw 3D. Your custom attribute types must have attribute type values that are less than 0. Here are the attribute types currently defined by QuickDraw 3D.

```
typedef enum TQ3AttributeTypes {
    kQ3AttributeTypeNone                                = 0,
    kQ3AttributeTypeSurfaceUV                           = 1,
    kQ3AttributeTypeShadingUV                           = 2,
    kQ3AttributeTypeNormal                              = 3,
    kQ3AttributeTypeAmbientCoefficient                  = 4,
    kQ3AttributeTypeDiffuseColor                        = 5,
    kQ3AttributeTypeSpecularColor                       = 6,
    kQ3AttributeTypeSpecularControl                     = 7,
    kQ3AttributeTypeTransparencyColor                   = 8,
    kQ3AttributeTypeSurfaceTangent                      = 9,
    kQ3AttributeTypeHighlightState                      = 10,
    kQ3AttributeTypeSurfaceShader                       = 11,
    kQ3AttributeTypeNumTypes
} TQ3AttributeTypes;
```

Constant descriptions

`kQ3AttributeTypeNone`
: The attribute has no type. You can pass this constant to the `Q3AttributeSet_GetNextAttributeType` function to get the first attribute type in an attribute set. When there are no more attribute types in a set, `Q3AttributeSet_GetNextAttributeType` returns `kQ3AttributeTypeNone`.

`kQ3AttributeTypeSurfaceUV`
: The attribute is a surface *uv* parameterization, of type `TQ3Param2D`.

`kQ3AttributeTypeShadingUV`
: The attribute is a shading *uv* parameterization, of type `TQ3Param2D`. A shading *uv* parameterization is an alternative to the surface *uv* parameterization that is used for shading.

See the chapter "Shader Objects" for more information about shading *uv* parameterizations.

`kQ3AttributeTypeNormal`
The attribute is a surface normal, of type `TQ3Vector3D`.

`kQ3AttributeTypeAmbientCoefficient`
The attribute is an ambient coefficient, of type `float`. An **ambient coefficient** determines the amount of ambient light reflected from an object's surface. An ambient coefficient should be between 0.0 (no reflection of ambient light) and 1.0 (complete reflection of ambient light).

`kQ3AttributeTypeDiffuseColor`
The attribute is a diffuse color, of type `TQ3ColorRGB`.

`kQ3AttributeTypeSpecularColor`
The attribute is a specular color, of type `TQ3ColorRGB`.

`kQ3AttributeTypeSpecularControl`
The attribute is a specular control, of type `float`.

`kQ3AttributeTypeTransparencyColor`
The attribute is a transparency color, of type `TQ3ColorRGB`. A **transparency color** determines the amount of light that can pass through a surface. The color (0, 0, 0) indicates complete transparency, and (1, 1, 1) indicates complete opacity. QuickDraw 3D multiplies an object's transparency color by its diffuse color when a transparency color attribute is attached to the object.

`kQ3AttributeTypeSurfaceTangent`
The attribute is a surface tangent, of type `TQ3Tangent2D`.

`kQ3AttributeTypeHighlightState`
The attribute is a highlight state, of type `TQ3Boolean`. A highlight state determines whether a highlight style overrides the material attributes of an object (`kQ3True`) or not (`kQ3False`).

`kQ3AttributeTypeSurfaceShader`
The attribute is a surface shader, of type `TQ3SurfaceShaderObject`. See the chapter "Shader Objects" for information on creating surface shaders and adding them to attribute sets. Note that when you include a surface shader in an attribute set, the reference count of the shader is incremented.

`kQ3AttributeTypeNumTypes`
The number of attribute types currently defined.

Attribute Objects Routines

This section describes routines you can use to manage attributes.

Drawing Attributes

QuickDraw 3D provides a routine that you can use to draw an attribute.

Q3Attribute_Submit

You can use the `Q3Attribute_Submit` function to submit an attribute in immediate mode.

```
TQ3Status Q3Attribute_Submit (
                    TQ3AttributeType attributeType,
                    const void *data,
                    TQ3ViewObject view);
```

`attributeType`
An attribute type.

`data` A pointer to the attribute's data.

`view` A view.

DESCRIPTION

The `Q3Attribute_Submit` function submits the attribute specified by the `attributeType` and `data` parameters into the view specified by the `view` parameter.

SPECIAL CONSIDERATIONS

You should call `Q3Attribute_Submit` only in a submitting loop.

Creating and Managing Attribute Sets

QuickDraw 3D provides a number of routines for creating and managing attribute sets.

Q3AttributeSet_New

You can use the `Q3AttributeSet_New` function to create an attribute set.

```
TQ3AttributeSet Q3AttributeSet_New (void);
```

DESCRIPTION

The `Q3AttributeSet_New` function returns, as its function result, a new empty attribute set. If `Q3AttributeSet_New` fails, it returns `NULL`.

Q3AttributeSet_Add

You can use the `Q3AttributeSet_Add` function to add an attribute to an attribute set.

```
TQ3Status Q3AttributeSet_Add (
                    TQ3AttributeSet attributeSet,
                    TQ3AttributeType type,
                    const void *data);
```

`attributeSet` An attribute set.

`type` An attribute type.

`data` A pointer to the attribute's data.

DESCRIPTION

The `Q3AttributeSet_Add` function adds the attribute specified by the `type` and `data` parameters to the attribute set specified by the `attributeSet` parameter. The attribute set must already exist when you call `Q3AttributeSet_Add`. If that

attribute set already contains an attribute of the specified type, `Q3AttributeSet_Add` replaces that attribute with the one specified by the `type` and `data` parameters. Note that the attribute data is copied into the attribute set. Accordingly, you can reuse the `data` parameter once you have called `Q3AttributeSet_Add`.

Q3AttributeSet_Contains

You can use the `Q3AttributeSet_Contains` function to determine whether an attribute set contains an attribute of a specific type.

```
TQ3Boolean Q3AttributeSet_Contains (
                    TQ3AttributeSet attributeSet,
                    TQ3AttributeType attributeType);
```

`attributeSet` An attribute set.

`attributeType` An attribute type.

DESCRIPTION

The `Q3AttributeSet_Contains` function returns, as its function result, a Boolean value that indicates whether the attribute set specified by the `attributeSet` parameter contains (`kQ3True`) or does not contain (`kQ3False`) an attribute of the type specified by the `attributeType` parameter.

Q3AttributeSet_Get

You can use the `Q3AttributeSet_Get` function to get the data associated with an attribute in an attribute set.

```
TQ3Status Q3AttributeSet_Get (
                    TQ3AttributeSet attributeSet,
                    TQ3AttributeType type,
                    void *data);
```

`attributeSet` An attribute set.

`type`	An attribute type.
`data`	On entry, a pointer to a structure large enough to hold the attribute data associated with attributes of the specified type. On exit, a pointer to the attribute data of the attribute having the specified type.

DESCRIPTION

The `Q3AttributeSet_Get` function returns, in the `data` parameter, the data currently associated with the attribute whose type is specified by the `type` parameter in the attribute set specified by the `attributeSet` parameter. If no attribute of that type is in the attribute set, `Q3AttributeSet_Get` returns `kQ3Failure` and posts the error `kQ3ErrorAttributeNotContained`.

If you pass the value `NULL` in the `data` parameter, no data is copied back to your application.

ERRORS

`kQ3ErrorAttributeNotContained`

Q3AttributeSet_GetNextAttributeType

You can use the `Q3AttributeSet_GetNextAttributeType` function to iterate through all the attributes in an attribute set.

```
TQ3Status Q3AttributeSet_GetNextAttributeType (
                    TQ3AttributeSet source,
                    TQ3AttributeType *type);
```

`source`	An attribute set.
`type`	On entry, an attribute type. On exit, the attribute type of the attribute that immediately follows that attribute in the attribute set.

DESCRIPTION

The `Q3AttributeSet_GetNextAttributeType` function returns, in the `type` parameter, the attribute type of the attribute that immediately follows the attribute having the type specified by the `type` parameter in the attribute set specified by the `source` parameter. To get the type of the first attribute in the attribute set, pass `kQ3AttributeTypeNone` in the `type` parameter. `Q3AttributeSet_GetNextAttributeType` returns `kQ3AttributeTypeNone` when it has reached then end of the list of attributes.

Q3AttributeSet_Empty

You can use the `Q3AttributeSet_Empty` function to empty an attribute set of all its attributes.

```
TQ3Status Q3AttributeSet_Empty (TQ3AttributeSet target);
```

`target` An attribute set.

DESCRIPTION

The `Q3AttributeSet_Empty` function removes all the attributes currently in the attribute set specified by the `target` parameter.

Q3AttributeSet_Clear

You can use the `Q3AttributeSet_Clear` function to remove an attribute of a certain type from an attribute set.

```
TQ3Status Q3AttributeSet_Clear (
                    TQ3AttributeSet attributeSet,
                    TQ3AttributeType type);
```

`attributeSet` An attribute set.

`type` An attribute type.

DESCRIPTION

The `Q3AttributeSet_Clear` function removes the attribute whose type is specified by the `type` parameter from the attribute set specified by the `attributeSet` parameter.

Q3AttributeSet_Submit

You can use the `Q3AttributeSet_Submit` function to submit an attribute set in immediate mode.

```
TQ3Status Q3AttributeSet_Submit (
                    TQ3AttributeSet attributeSet,
                    TQ3ViewObject view);
```

`attributeSet` An attribute set.

`view` A view.

DESCRIPTION

The `Q3AttributeSet_Submit` function submits the attribute set specified by the `attributeSet` parameter into the view specified by the `view` parameter.

SPECIAL CONSIDERATIONS

You should call `Q3AttributeSet_Submit` only in a submitting loop.

Q3AttributeSet_Inherit

You can use the `Q3AttributeSet_Inherit` function to configure an attribute set so that it contains all the attributes of a child set together with all the attributes inherited from a parent set.

```
TQ3Status Q3AttributeSet_Inherit (
                    TQ3AttributeSet parent,
                    TQ3AttributeSet child,
                    TQ3AttributeSet result);
```

`parent` An attribute set.

`child` An attribute set.

`result` On entry, an attribute set. On exit, an attribute set that contains all the attributes in the specified child set together with all the attributes inherited from the specified parent set.

DESCRIPTION

The `Q3AttributeSet_Inherit` function returns, in the `result` parameter, an attribute set that merges attributes from the attribute sets specified by the `child` and `parent` parameters. The resulting set contains all the attributes in the child set together with all those in the parent set having an attribute type that is *not* contained in the child attribute set.

If the specified child and parent attribute sets contain any custom attribute types, `Q3AttributeSet_Inherit` uses the custom type's `kQ3MethodTypeAttributeCopyInherit` custom method. See the chapter "QuickDraw 3D Objects" for complete information on custom element types.

Registering Custom Attributes

You can add a custom attribute type by calling the `Q3AttributeClass_Register` function. If necessary, you can delete an application-defined attribute type by calling the `Q3XObjectHierarchy_UnregisterClass` function.

Note
For complete details on adding custom attribute types, see "Defining Custom Attribute Types," beginning on page 522. ◆

Q3AttributeClass_Register

You can use the `Q3AttributeClass_Register` function to register an application-defined attribute type.

```
TQ3ObjectClass Q3AttributeClass_Register (
                    TQ3AttributeType attributeType,
                    const char *creatorName,
                    unsigned long sizeOfElement,
                    TQ3MetaHandler metaHandler);
```

`attributeType` The type of your custom attribute.

`creatorName` A pointer to a null-terminated string containing the name of the attribute's creator and the name of the type of attribute being registered.

`sizeOfElement` The size of the data associated with the specified custom attribute type.

`metaHandler` A pointer to an application-defined metahandler that QuickDraw 3D calls to handle the new custom attribute type.

DESCRIPTION

The `Q3AttributeClass_Register` function returns, as its function result, an object class reference for a new custom attribute type having a type specified by the `attributeType` parameter and a name specified by the `creatorName` parameter. The `metaHandler` parameter is a pointer to the metahandler for your custom attribute type. See the chapter "QuickDraw 3D Objects" for information on writing a metahandler. If `Q3AttributeClass_Register` cannot create a new attribute type, it returns the value `NULL`.

The `creatorName` parameter should be a pointer to null-terminated C string that contains your (or your company's) name and the name of the type of attribute you are defining. Use the colon character (:) to delimit fields within this string. The string should not contain any spaces or punctuation other than the colon character, and it cannot end with a colon. Here are some examples of valid creator names:

```
"MyCompany:SurfDraw:Wavelength"
"MyCompany:SurfWorks:VRModule:WaterTemperature"
```

The `sizeOfElement` parameter specifies the fixed size of the data associated with your custom attribute type. You can associate dynamically sized data with your attribute type by putting a pointer to a dynamically sized block of data into the attribute set and having your handler's copy method duplicate the data. (In this case, you would set the `sizeOfElement` parameter to `sizeof(Ptr)`.) Your handler's dispose method must also deallocate any dynamically sized blocks.

Adding Application-Defined Attribute and Element Types

You can add new application-defined attribute and element types by using `Q3XAttributeClass_Register` and `Q3XElementClass_Register`. These functions let an application add a new attribute or element type by registering its methods. A unique object type of `TQ3AttributeType` or `TQ3ElementType` is returned, which you can use to set and get the custom attribute or element:

```
typedef TQ3ElementType TQ3AttributeType;
```

Q3XAttributeClass_Register

You can use the `Q3XAttributeClass_Register` function to add a new attribute type to QuickDraw 3D.

```
TQ3XObjectClass Q3XAttributeClass_Register(
                    TQ3AttributeType    *attributeType,
                    const char          *name,
                    unsigned long       sizeOfElement,
                    TQ3XMetaHandler     metaHandler);
```

`attributeType` A new attribute type, which is returned to you.

`name` The new object name, for use in the text metafile.

`sizeOfElement` The size, in bytes, required for the new object.

`metaHandler` A metahandler that returns nonvirtual methods. This value may be `NULL` for some classes.

DESCRIPTION

The `Q3XAttributeClass_Register` function returns, in the `attributeType` parameter, a new attribute with name `name` and size `sizeOfElement` for the metahandler specified by `metaHandler`.

Q3XElementClass_Register

You can use the `Q3XElementClass_Register` function to add a new element type to QuickDraw 3D.

```
TQ3XObjectClass Q3XElementClass_Register(
                    TQ3ElementType     *elementType,
                    const char         *name,
                    unsigned long      sizeOfElement,
                    TQ3XMetaHandler    metaHandler);
```

`elementType` A new element type, which QuickDraw 3D assigns and returns to you.

`name` The new object name, for use in the text metafile.

`sizeOfElement` The size, in bytes, required for the new object.

`metaHandler` A metahandler that returns nonvirtual methods. This value may be `NULL` for some classes.

DESCRIPTION

The `Q3XElementClass_Register` function returns, in the `elementType` parameter, a new element with name `name` and size `sizeOfElement` for the metahandler specified by `metaHandler`.

Copy Methods

An application-defined attribute or element type may support any of four copying methods. The reason for defining four different methods is to let you customize the semantics of your `Q3AttributeSet_Add` and `Q3AttributeSet_Get` calls to allow for various meanings of the `data` parameter.

TQ3XElementCopyAddMethod

The `TQ3XElementCopyAddMethod` functionality is optional. If included, it supports adding a new attribute or element type to an attribute set. If it is not included, the default action is a memory copy of `sizeOfElement` bytes.

```
#define kQ3XMethodTypeElementCopyAdd Q3_METHOD_TYPE('e','c','p','a')

typedef TQ3Status (*TQ3XElementCopyAddMethod)(
                    const void    *fromAPIElement,
                    void          *toInternalElement);
```

`fromAPIElement`
API element to copy from.

`toInternalElement`
Internal element to copy to.

DESCRIPTION

The `TQ3XElementCopyAddMethod` method is called when adding a new attribute type to an attribute set, using `Q3AttributeSet_Add`. The application-defined object is passed whatever pointer was passed in the `Q3AttributeSet_Add` call as the `fromAPIElement` parameter, and in `toInternalElement` a pointer to an empty, uninitialized block of size `sizeOfElement` (obtained from the registration call).

TQ3XElementCopyReplaceMethod

The `TQ3XElementCopyReplaceMethod` functionality is optional. If included, it supports replacing an existing attribute or element type. If it is not included, the default action is a memory copy of `sizeOfElement` bytes.

```
#define kQ3XMethodTypeElementCopyReplace Q3_METHOD_TYPE('e','c','p','r')

typedef TQ3Status (*TQ3XElementCopyReplaceMethod)(
                    const void    *fromAPIElement,
                    void          *ontoInternalElement);
```

`fromAPIElement`
API element to copy from.

`ontoInternalElement`
Internal element to replace with copy.

DESCRIPTION

The `TQ3XElementCopyReplaceMethod` method is called when replacing an existing attribute type in an attribute set, using `Q3AttributeSet_Add`.The application-defined object is passed whatever pointer was passed in the `Q3AttributeSet_Add` call as the `fromAPIElement` parameter, and in `ontoInternalElement` a pointer to an existing block of size `sizeOfElement` (obtained from the registration call) that has previously been initialized with a `CopyAdd` call.

TQ3XElementCopyGetMethod

The `TQ3XElementCopyGetMethod` functionality is optional. If included, it supports fetching an attribute or element from an attribute set. If it is not included, the default action is a memory copy of `sizeOfElement` bytes.

```
#define kQ3XMethodTypeElementCopyGet Q3_METHOD_TYPE('e','c','p','g')

typedef TQ3Status (*TQ3XElementCopyGetMethod)(
                    const void    *fromInternalElement,
                    void          *toAPIElement);
```

`fromInternalElement`
Internal element to copy from.

`toAPIElement`
API element to copy to.

DESCRIPTION

The `TQ3XElementCopyGetMethod` method is called when obtaining an attribute from an attribute set by means of `Q3AttributeSet_Get`. The application-defined object is passed in `fromInternalElement` a pointer to an existing block of size `sizeOfElement`

(determined by the registration call) that has been initialized with a `CopyAdd` call. It is passed in `toAPIElement` whatever pointer was passed in the `Q3AttributeSet_Get` call.

TQ3XElementCopyDuplicateMethod

The `TQ3XElementCopyDuplicateMethod` functionality is optional. If included, it supports internal duplication of attributes or elements. If it is not included, the default action is a memory copy of `sizeOfElement` bytes.

```
#define kQ3XMethodTypeElementCopyDuplicate
                    Q3_METHOD_TYPE('e','c','p','d')

typedef TQ3Status (*TQ3XElementCopyDuplicateMethod)(
                    const void    *fromInternalElement,
                    void          *toInternalElement);
```

`fromInternalElement`
Internal element to copy from.

`toInternalElement`
Internal element to copy to.

DESCRIPTION

The `TQ3XElementCopyDuplicateMethod` method is called when `Q3Object_Duplicate` is called for an attribute set or element set. It copies data from a valid block of size `sizeOfElement` (determined by the registration call) to another block of size `sizeOfElement`. The block copied into must be invalid.

Deletion Method

An application-defined attribute or element type may support a deletion method.

TQ3XElementDeleteMethod

The `TQ3XElementDeleteMethod` functionality is optional. If included, it deletes an attribute or element.

```
#define kQ3XMethodTypeElementDelete Q3_METHOD_TYPE('e','c','p','l')

typedef TQ3Status (*TQ3XElementDeleteMethod)(
                    void    *internalElement);
```

`internalElement`
Internal element to delete.

DESCRIPTION

The `TQ3XElementDeleteMethod` method deletes the element pointed to by `internalElement`.

Getting the Size of an Attribute or Element

The `Q3XElementType_GetElementSize` routine returns the size of an attribute or element.

Q3XElementType_GetElementSize

The `Q3XElementType_GetElementSize` function returns the size in bytes of an asttribute or element type.

```
TQ3Status Q3XElementType_GetElementSize(
                    TQ3ElementType    elementType,
                    unsigned long     *sizeOfElement);
```

`elementType` An attribute or element type.

`sizeOfElement`
The size in bytes of the type.

DESCRIPTION

The `Q3XElementType_GetElementSize` function returns, in the `sizeOfElement` parameter, the size in bytes of the attribute or element indicated by the `elementType` parameter.

Inheritance Control and Copying

Custom attributes may be inherited or not, as determined by the metahandler. If inheritance is supported, then the metahandler should return `kQ3True` from the `kQ3XMethodTypeAttributeInherit` method and `kQ3False` otherwise.

If inheritance is supported, the inheritance function called internally just copies the data representing the attribute, using the size parameter specified in the registration call. If more complex behavior is desired, however, the custom attribute must be supplied with a `TQ3XAttributeCopyInheritMethod` method to be called at inheritance time. This might be the case, for example, if there were a variable-size array in the data structure.

TQ3XAttributeInheritMethod

The `TQ3XAttributeInheritMethod` functionality reports whether an application-defined attribute supports inheritance.

```
#define kQ3XMethodTypeAttributeInherit Q3_METHOD_TYPE('i','n','h','t')

typedef TQ3Boolean TQ3XAttributeInheritMethod;
```

return value `kQ3True` or `kQ3False`.

DESCRIPTION

The `TQ3XAttributeInheritMethod` method returns `kQ3True` if inheritance is supported and `kQ3False` otherwise.

TQ3XAttributeCopyInheritMethod

The `TQ3XAttributeCopyInheritMethod` functionality performs inheritance of an attribute in an application-defined object.

```
#define kQ3XMethodTypeAttributeCopyInherit
                    Q3_METHOD_TYPE('a','c','p','i')

typedef TQ3Status (*TQ3XAttributeCopyInheritMethod)(
                    const void    *fromInternalAttribute,
                    void          *toInternalAttribute);
```

`fromInternalAttribute`
Internal attribute to be copied from.

`toInternalAttribute`
Internal attribute to be copied to.

DESCRIPTION

The `TQ3XAttributeCopyInheritMethod` method provides inheritance from the attribute designated by `fromInternalAttribute` to the attribute designated by `toInternalAttribute`.

Attribute Errors

The following is a list of errors that attribute routines can return. A list of general QuickDraw 3D errors is given in “QuickDraw 3D Errors, Warnings, and Notices” (page 87).

```
kQ3ErrorAttributeNotContained
kQ3ErrorAttributeInvalidType
```

CHAPTER 6

Style Objects

This chapter describes style objects (or styles) and the functions you can use to manipulate them. You use styles to specify some of the basic characteristics of a renderer. For example, one renderer style determines whether an object is drawn as a solid filled object or as a set of edges. Another renderer style determines whether a surface is drawn smoothly or as a set of polygonal facets.

To use this chapter, you should already be familiar with the QuickDraw 3D class hierarchy, described in the chapter "QuickDraw 3D Objects" earlier in this book. For information about renderers, see the chapter "Renderer Objects" in this book. You do not, however, need to know how to create or manipulate renderers to read this chapter.

This chapter begins by describing style objects and their features. Then it shows how to specify the current rendering styles of a model. The section "Style Objects Reference," beginning on page 555 provides a complete description of style objects and the routines you can use to create and manipulate them.

About Style Objects

A **style object** (or, more briefly, a **style**) is a type of QuickDraw 3D object that determines some of the basic characteristics of the renderer used to draw the geometric objects in a scene. A style is of type `TQ3StyleObject`, which is a subclass of a shape object.

You can apply a style to a model by creating a style object and then submitting it to the model. QuickDraw 3D provides functions that allow both retained and immediate style submitting. Alternatively, you can create a style object and then add it to a group. Then, when the group is submitted for rendering, the style is applied to all objects in the group (if it's an ordered display group) or to all objects in the group following the style (if it's a display group).

Note
See the chapter "Group Objects" for complete information on how styles are applied to the objects in a group. ◆

QuickDraw 3D defines these types of styles that affect the rendering or picking of a scene:

- backfacing styles
- interpolation styles
- fill styles
- highlight styles
- subdivision styles
- orientation styles
- shadow-receiving styles
- picking ID styles
- picking parts styles

Unlike attributes, which define characteristics of the appearances of individual surfaces and can be applied to only part of a model, styles define characteristics of a renderer and are generally (but not always) applied to a model as a whole.

IMPORTANT
Some renderers might not support all types of styles, and some renderers might not be able to apply a given style to all geometric objects. For example, not all renderers can draw shadows; such renderers therefore ignore the shadow-receiving style. ▲

If you apply a style to an object and then apply a different style of the same type to that object, the style applied second replaces the style applied first.

Backfacing Styles

A model's **backfacing style** determines whether or not a renderer draws shapes (typically polygons) that face away from a view's camera. QuickDraw 3D defines constants for the backfacing styles that are currently available.

```
typedef enum TQ3BackfacingStyle {
    kQ3BackfacingStyleBoth,
    kQ3BackfacingStyleRemove,
    kQ3BackfacingStyleFlip
} TQ3BackfacingStyle;
```

The default value, `kQ3BackfacingStyleBoth`, specifies that the renderer should draw shapes that face either toward or away from the camera. The backfacing shapes may be illuminated only dimly or not at all, because their face normals point away from the camera.

The constant `kQ3BackfacingStyleRemove` specifies that the renderer should not draw or otherwise process shapes that face away from the camera. (This process is called **backface culling.**) This rendering style is likely to be significantly faster than the other two backfacing styles (because up to half the shapes are not rendered) but can cause holes to appear in visible backfacing objects.

Note
An object that faces away from the camera might still be visible. Accordingly, backface culling is not the same as hidden surface removal. ◆

The constant `kQ3BackfacingStyleFlip` specifies that the renderer should draw shapes that face toward or away from the camera. The face normals of backfacing shapes are inverted so that they face toward the camera.

Interpolation Styles

A model's **interpolation style** determines the method of interpolation a renderer uses when applying lighting or other shading effects to a surface. QuickDraw 3D defines constants for the interpolation styles that are currently available.

```
typedef enum TQ3InterpolationStyle {
    kQ3InterpolationStyleNone,
    kQ3InterpolationStyleVertex,
    kQ3InterpolationStylePixel
} TQ3InterpolationStyle;
```

The constant `kQ3InterpolationStyleNone` specifies that no interpolation is to occur. When a renderer applies an effect (such as illumination) to a surface, it

calculates a single intensity value for an entire polygon. This style results in a model's surfaces having a faceted appearance.

To render surfaces smoothly, you can specify one of two interpolation styles. The constant `kQ3InterpolationStyleVertex` specifies that the renderer is to interpolate values linearly across a polygon, using the values at the vertices. The constant `kQ3InterpolationStylePixel` specifies that the renderer is to apply an effect at every pixel in the image. For example, a renderer will calculate illumination based on the surface normal of every pixel in the image. This rendering style is likely to be computation-intensive.

Fill Styles

A model's **fill style** determines whether an object is drawn as a solid filled object or is drawn as a set of edges or points. QuickDraw 3D defines constants for the fill styles that are currently available.

```
typedef enum TQ3FillStyle {
    kQ3FillStyleFilled,
    kQ3FillStyleEdges,
    kQ3FillStylePoints
} TQ3FillStyle;
```

The default value, `kQ3FillStyleFilled`, specifies that the renderer should draw shapes as solid filled objects. The constant `kQ3FillStyleEdges` specifies that the renderer should draw shapes as the sets of lines that define the edges of the surfaces rather than as filled shapes. The constant `kQ3FillStylePoints` specifies that the renderer should draw shapes as the sets of points that define the vertices of the surfaces. This fill style is used primarily to accelerate the rendering of very complex shapes.

Highlight Styles

A model's **highlight style** determines the material attributes of a geometric object (or a group of geometric objects) that override the normal attributes of the object (or group of objects). For example, it is often useful during interaction with the objects in a model to highlight a selected shape by changing its color. You can define the specific highlight style to be applied to a selected object, thus avoiding the need to edit the geometric description of the object simply to change its color or other attributes.

If a highlight style is defined for a model, any renderers that support highlighting will use the attributes in that style to override the material attributes defined for any geometric objects in the model. However, the highlight style is used for a particular geometric object only if the object's **highlight state** (that is, an attribute of type `kQ3AttributeTypeHighlightState` that has data of type `TQ3Boolean`) is set to `kQ3True`. For example, suppose that the attribute set of a box contains an attribute of type `kQ3AttributeTypeHighlightState`, which is set to `kQ3True`. Further, suppose that the face attribute sets of the box do not contain any attributes of that type. In this case, the attribute set of the current highlight style is used during rendering.

Subdivision Styles

A model's **subdivision style** determines how a renderer decomposes smooth curves and surfaces into polylines and polygonal meshes for display purposes. You can control the fineness of the decomposition by changing either the subdivision style or the parameters associated with a particular style. QuickDraw 3D defines constants for the subdivision styles that are currently available.

```
typedef enum TQ3SubdivisionMethod {
    kQ3SubdivisionMethodConstant,
    kQ3SubdivisionMethodWorldSpace,
    kQ3SubdivisionMethodScreenSpace
} TQ3SubdivisionMethod;
```

The value `kQ3SubdivisionMethodConstant` specifies **constant subdivision:** the renderer should subdivide a curve into some given number of polyline segments and a surface into a certain-sized mesh of polygons.

The value `kQ3SubdivisionMethodWorldSpace` specifies **world-space subdivision:** the renderer should subdivide a curve (or surface) into polylines (or polygons) whose sides have a world-space length that is at most as large as a given value.

The value `kQ3SubdivisionMethodScreenSpace` specifies **screen-space subdivision:** the renderer should subdivide a curve (or surface) into polylines (or polygons) whose sides have a length that is at most as large as some number of pixels.

A full specification of a subdivision style requires both a **subdivision method** (which is specified by one of the three subdivision style constants) together with

one or two **subdivision method specifiers.** For a curve rendered with constant subdivision, for example, the subdivision method specifier indicates the number of polylines into which the curve is to be subdivided. A subdivision method specifier is passed either as a parameter to a routine or as a field in a subdivision style data structure. See page 555 for complete details on the meaning of subdivision method specifiers for each of the three subdivision methods.

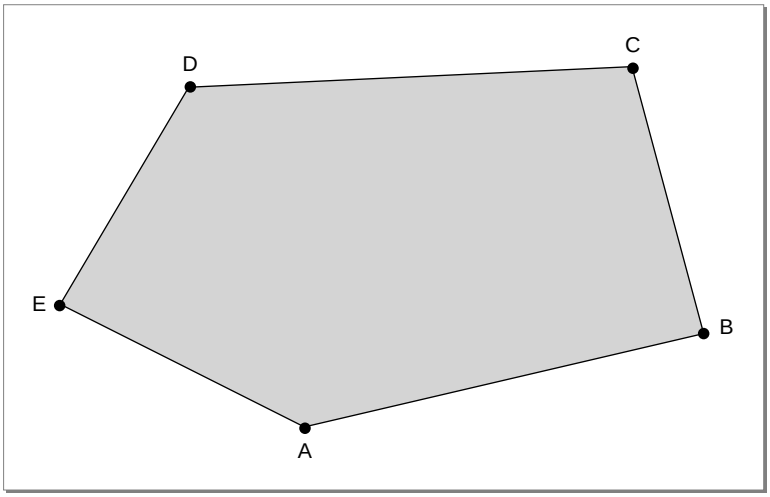
Orientation Styles

A model's **orientation style** determines which side of a planar surface is considered to be the "front" side. QuickDraw 3D defines constants for the orientation styles that are currently available.

```
typedef enum TQ3OrientationStyle {
    kQ3OrientationStyleCounterClockwise,
    kQ3OrientationStyleClockwise
} TQ3OrientationStyle;
```

The default value, `kQ3OrientationStyleCounterClockwise`, specifies that the front face of a polygonal shape is that face whose vertices are listed in counterclockwise order. The constant `kQ3OrientationStyleClockwise` specifies that the front face of a polygonal shape is that face whose vertices are listed in clockwise order. Figure 6-1 shows the front of a polygonal face.

Figure 6-1 The front side of a polygon



The cross product of the vectors formed by the first two edges (that is, by the segments from A to B and from B to C) points straight out of the page, indicating that this is the front side of the polygon. The renderer will use this information for operations such as culling and shading. If you change the orientation style to clockwise, you must make sure that the polygonal shape corresponds.

Note
The orientation style affects only explicitly polygonal geometric primitives, such as triangles, simple and general polygons, and polyhedral primitives. It does not affect the appearance of other primitives, such as cyclinders and NURB patches, and these primitives need not be converted to polygonal shapes for rendering, picking, bounding, or similar operations. ◆

Shadow-Receiving Styles

A model's **shadow-receiving style** determines whether or not objects in a model receive shadows cast by other objects in the model. The shadow-receiving style is defined by a Boolean value. If a renderer's

shadow-receiving style is set to `kQ3True`, objects in the scene receive shadows. If a renderer's shadow-receiving style is set to `kQ3False`, objects in the scene do not receive shadows.

Picking ID Styles

A **picking ID style** determines the picking ID of an object in a model. A **picking ID** is an arbitrary 32-bit integer that you can use to determine which object was selected by a pick operation. For example, you can assign different picking IDs to the eight corners of a cube; when the user selects a corner, you can inspect the corner's picking ID (by looking at the `pickID` field of the hit data structure associated with that corner) to determine which corner was selected.

Note
See the chapter "Pick Objects" for complete information about picking. ◆

You assign a picking ID to a geometric object by creating a picking ID style having the desired picking ID and then submitting that style object before submitting the geometric object. See "Managing Picking ID Styles," beginning on page 576 for a description of the functions you can use to create and manipulate picking ID styles.

IMPORTANT
QuickDraw 3D does not perform any validation to ensure that the picking IDs you assign to objects in a model are unique. It is your application's responsibility to generate unique picking IDs. ▲

Picking Parts Styles

A model's **picking parts style** determines the kinds of objects that are eligible for placement in a hit list during a pick operation. Currently, you can use the picking parts style to limit your attention to certain parts of a mesh. The picking parts style is specified by a value defined using one or more pick parts masks, which are defined by these constants:

```
typedef enum TQ3PickPartsMasks {
    kQ3PickPartsObject                                  = 0,
    kQ3PickPartsMaskFace                                = 1 << 0,
```

```
    kQ3PickPartsMaskEdge                                    = 1 << 1,
    kQ3PickPartsMaskVertex                                  = 1 << 2
} TQ3PickPartsMasks;
```

The default picking parts style is `kQ3PickPartsObject`, which indicates that the hit list is to contain only whole objects. You can add in the other masks to select parts of a mesh for picking. For instance, to pick edges and vertices, you would draw a pick parts style using the value:

```
kQ3PickPartsMaskEdge | kQ3PickPartsMaskVertex
```

Note
For a description of mesh parts, see the chapter "Geometric Objects." For complete information about picking parts, see the chapter "Pick Objects." ◆

Anti-Alias Style

Many renderers implement scan-conversion and rasterization algorithms by displaying pixels at the borders of lines and polygons either completely on or completely off. This may result in "jaggies," a staircase-like appearance at edges that are not perfectly horizontal or vertical. Such artifacts are visually unpleasant, both in static scenes and in dynamically updated sequences of frames, where they can produce apparent motion across lines or the edges of polygons.

Many renderers support **anti-aliasing** techniques, which reduce or eliminate this problem. For more complete information about anti-aliasing, consult a book such as *Computer Graphics: Principles and Practice* by Foley and Van Dam.

QuickDraw 3D provides an anti-alias style to invoke and control anti-aliasing in renderers. Because different renderers support a variety of anti-aliasing algorithms, QuickDraw 3D provides control at an abstract level, with its interpretation left to the renderer. The control structures for anti-alias style are the following:

```
typedef enum TQ3AntiAliasModeMasks {
    kQ3AntiAliasModeMaskEdges       = 1 << 0,
    kQ3AntiAliasModeMaskFilled      = 1 << 1
} TQ3AntiAliasModeMasks;

typedef unsigned long TQ3AntiAliasMode;
```

```
typedef struct TQ3AntiAliasStyleData {
    TQ3Switch               state;
    TQ3AntiAliasMode        mode;
    float                   quality;
} TQ3AntiAliasStyleData;
```

You can use the `state` field, of type `TQ3Switch`, to turn anti-aliasing on and off. It lets you leave the other state variables set to desired values (either in the data structure or in a style object) and turn anti-aliasing on and off without needing to reinitialize the rest of the state.

You can use the `mode` field to control which primitives the anti-aliasing techniques affect. If the field is set to `kQ3AntiAliasModeMaskEdges`, then lines, polylines, and ellipses are anti-aliased, plus all other primitives if you are using an edge fill style. Filled primitives (triangles, NURB surfaces, polyhedra, etc.) are anti-aliased if the mode is set to `kQ3AntiAliasModeMaskFilled`. Both classes of primitives are anti-aliased if the field is set to `kQ3AntiAliasModeMaskEdges | kQ3AntiAliasModeMaskFilled`.

Setting the `quality` field to a value between 0 and 1 provides general control over the level of anti-aliasing. The effect of this value depends on the anti-aliasing algorithm and how the renderer implements it, but in general 0 means a very low level of anti-aliasing and 1 means a very high level. A `quality` value of 0 does not necessarily mean that anti-aliasing is off, but rather that it is at the lowest level implemented by the renderer.

Routines that help you control anti-aliasing are described in "Managing the Anti-Alias Style," beginning on page 581.

Using Style Objects

You apply styles either by submitting them during rendering or picking or by including a style object in a group. See Listing 1-11 (page 70) in the chapter "Introduction to QuickDraw 3D" for examples of submitting styles during retained mode rendering. See Listing 15-3 (page 960) in the chapter "Pick Objects" for an example of submitting a style during immediate mode picking.

Style Objects Reference

This section describes the data structures and routines you can use to manage style objects.

Data Structures

This section describes the data structures supplied by QuickDraw 3D for managing style objects.

Subdivision Style Data Structure

You use a **subdivision style data structure** to get or set information about the type of subdivision of curves and surfaces used by a renderer. A subdivision style data structure is defined by the `TQ3SubdivisionStyleData` data type.

```
typedef struct TQ3SubdivisionStyleData {
    TQ3SubdivisionMethod                method;
    float                               c1;
    float                               c2;
} TQ3SubdivisionStyleData;
```

Field descriptions

`method` The method of curve and surface subdivision used by the renderer. This field must contain one of these constants:

```
kQ3SubdivisionMethodConstant
kQ3SubdivisionMethodWorldSpace
kQ3SubdivisionMethodScreenSpace
```

The constant `kQ3SubdivisionMethodConstant` indicates that the renderer subdivides a curve into a number (specified in the `c1` field) of polyline segments and a surface into a mesh (whose dimensions are specified by the `c1` and `c2` fields) of polygons. The constant `kQ3SubdivisionMethodWorldSpace` indicates that the renderer subdivides a curve (or surface)

into polylines (or polygons) whose sides have a world-space length that is at most as large as the value specified in the `c1` field. The constant `kQ3SubdivisionMethodScreenSpace` indicates that the renderer subdivides a curve (or surface) into polylines (or polygons) whose sides have a length that is at most as large as the number of pixels specified in the `c1` field.

`c1` For constant subdivision, the number of polylines into which a curve should be subdivided, or the number of vertices in the *u* parametric direction of the polygonal mesh into which a surface is subdivided. For world-space subdivision, the maximum length of a polyline segment (or polygon side) into which a curve (or surface) is subdivided. For screen-space subdivision, the maximum number of pixels in a polyline segment (or polygon side) into which a curve (or surface) is subdivided; for a NURB curve or surface, however, `c1` specifies the maximum allowable distance between the curve or surface and the polylines or polygons into which it is subdivided. The value in this field should be an integer greater than 0 for constant subdivision, and greater than 0.0 for world-space or screen-space subdivision.

`c2` For constant subdivision, the number of vertices in the *v* parametric direction of the polygonal mesh into which a surface is subdivided. The value in this field should be an integer greater than 0. For world-space and screen-space subdivision, this field is unused.

Style Objects Routines

This section describes the routines you can use to manage a renderer's styles.

Managing Styles

QuickDraw 3D provides general routines for operating with style objects.

Q3Style_GetType

You can use the `Q3Style_GetType` function to get the type of a style object.

```
TQ3ObjectType Q3Style_GetType (TQ3StyleObject style);
```

`style` A style object.

DESCRIPTION

The `Q3Style_GetType` function returns, as it s function result, the type of the style object specified by the `style` parameter. The types of style objects currently supported by QuickDraw 3D are defined by these constants:

```
kQ3StyleTypeBackfacing
kQ3StyleTypeFill
kQ3StyleTypeHighlight
kQ3StyleTypeInterpolation
kQ3StyleTypeOrientation
kQ3StyleTypePickID
kQ3StyleTypePickParts
kQ3StyleTypeReceiveShadows
kQ3StyleTypeSubdivision
```

If the specified style object is invalid or is not one of these types, `Q3Style_GetType` returns the value `kQ3ObjectTypeInvalid`.

Q3Style_Submit

You can use the `Q3Style_Submit` function to submit a style in retained mode.

```
TQ3Status Q3Style_Submit (
                    TQ3StyleObject style,
                    TQ3ViewObject view);
```

`style` A style object.

`view` A view.

DESCRIPTION

The `Q3Style_Submit` function submits the style specified by the `style` parameter to the view specified by the `view` parameter.

SPECIAL CONSIDERATIONS

You should call `Q3Style_Submit` only in a submitting loop.

Managing Backfacing Styles

QuickDraw 3D provides routines that you can use to manage backfacing styles.

Q3BackfacingStyle_New

You can use the `Q3BackfacingStyle_New` function to create a new backfacing style object.

```
TQ3StyleObject Q3BackfacingStyle_New (
                    TQ3BackfacingStyle backfacingStyle);
```

`backfacingStyle`
A backfacing style value.

DESCRIPTION

The `Q3BackfacingStyle_New` function returns, as its function result, a new style object having the backfacing style specified by the `backfacingStyle` parameter. The `backfacingStyle` parameter should be one of these values:

```
kQ3BackfacingStyleBoth
kQ3BackfacingStyleRemove
kQ3BackfacingStyleFlip
```

If a new style object could not be created, `Q3BackfacingStyle_New` returns the value `NULL`.

To change the current backfacing style, you must actually draw the style object. You can call `Q3Style_Submit` to draw the style in retained mode or

Q3BackfacingStyle_Submit (described next) to draw the style in immediate mode.

SEE ALSO

See "Backfacing Styles" (page 546) for a description of the available backfacing styles.

Q3BackfacingStyle_Submit

You can use the Q3BackfacingStyle_Submit function to submit a backfacing style for drawing in immediate mode.

```
TQ3Status Q3BackfacingStyle_Submit (
                    TQ3BackfacingStyle backfacingStyle,
                    TQ3ViewObject view);
```

backfacingStyle
: A backfacing style value.

view
: A view.

DESCRIPTION

The Q3BackfacingStyle_Submit function sets the backfacing style of the view specified by the view parameter to the style specified in the backfacingStyle parameter.

SPECIAL CONSIDERATIONS

You should call Q3BackfacingStyle_Submit only in a submitting loop.

Q3BackfacingStyle_Get

You can use the Q3BackfacingStyle_Get function to get the backfacing style value of a backfacing style.

```
TQ3Status Q3BackfacingStyle_Get (
                    TQ3StyleObject backfacingObject,
                    TQ3BackfacingStyle *backfacingStyle);
```

`backfacingObject`
A backfacing style object.

`backfacingStyle`
On exit, a pointer to the backfacing style value of the specified backfacing style object.

DESCRIPTION

The `Q3BackfacingStyle_Get` function returns, in the `backfacingStyle` parameter, a pointer to the current backfacing style value of the backfacing style object specified by the `backfacingObject` parameter.

Q3BackfacingStyle_Set

You can use the `Q3BackfacingStyle_Set` function to set the backfacing style value of a backfacing style.

```
TQ3Status Q3BackfacingStyle_Set (
                    TQ3StyleObject backfacingObject,
                    TQ3BackfacingStyle backfacingStyle);
```

`backfacingObject`
A backfacing style object.

`backfacingStyle`
A backfacing style value.

DESCRIPTION

The `Q3BackfacingStyle_Set` function sets the backfacing style value of the style object specified by the `backfacingObject` parameter to the value specified in the `backfacingStyle` parameter.

Managing Interpolation Styles

QuickDraw 3D provides routines that you can use to manage interpolation styles.

Q3InterpolationStyle_New

You can use the `Q3InterpolationStyle_New` function to create a new interpolation style object.

```
TQ3StyleObject Q3InterpolationStyle_New (
                    TQ3InterpolationStyle interpolationStyle);
```

`interpolationStyle`
An interpolation style value.

DESCRIPTION

The `Q3InterpolationStyle_New` function returns, as its function result, a new style object having the interpolation style specified by the `interpolationStyle` parameter. The `interpolationStyle` parameter should be one of these values:

```
kQ3InterpolationStyleNone
kQ3InterpolationStyleVertex
kQ3InterpolationStylePixel
```

If a new style object could not be created, `Q3InterpolationStyle_New` returns the value `NULL`.

To change the current interpolation style, you must actually draw the style object. You can call `Q3Style_Submit` to draw the style in retained mode or `Q3InterpolationStyle_Submit` (described next) to draw the style in immediate mode.

SEE ALSO

See "Interpolation Styles" (page 547) for a description of the available interpolation styles.

Q3InterpolationStyle_Submit

You can use the `Q3InterpolationStyle_Submit` function to submit an interpolation style in immediate mode.

```
TQ3Status Q3InterpolationStyle_Submit (
                    TQ3InterpolationStyle interpolationStyle,
                    TQ3ViewObject view);
```

`interpolationStyle`
An interpolation style value.

`view` A view.

DESCRIPTION

The `Q3InterpolationStyle_Submit` function sets the interpolation style of the view specified by the `view` parameter to the style specified in the `interpolationStyle` parameter.

SPECIAL CONSIDERATIONS

You should call `Q3InterpolationStyle_Submit` only in a submitting loop.

Q3InterpolationStyle_Get

You can use the `Q3InterpolationStyle_Get` function to get the interpolation style value of an interpolation style.

```
TQ3Status Q3InterpolationStyle_Get (
                    TQ3StyleObject interpolationObject,
                    TQ3InterpolationStyle *interpolationStyle);
```

`interpolationObject`
An interpolation style object.

`interpolationStyle`
On exit, a pointer to the interpolation style value of the specified interpolation style object.

DESCRIPTION

The `Q3InterpolationStyle_Get` function returns, in the `interpolationStyle` parameter, a pointer to the current interpolation style value of the interpolation style object specified by the `interpolationObject` parameter.

Q3InterpolationStyle_Set

You can use the `Q3InterpolationStyle_Set` function to set the interpolation style value of an interpolation style.

```
TQ3Status Q3InterpolationStyle_Set (
                    TQ3StyleObject interpolationObject,
                    TQ3InterpolationStyle interpolationStyle);
```

`interpolationObject`
An interpolation style object.

`interpolationStyle`
An interpolation style value.

DESCRIPTION

The `Q3InterpolationStyle_Set` function sets the interpolation style value of the style object specified by the `interpolationObject` parameter to the value specified in the `interpolationStyle` parameter.

Managing Fill Styles

QuickDraw 3D provides routines that you can use to manage fill styles.

Q3FillStyle_New

You can use the `Q3FillStyle_New` function to create a new fill style object.

```
TQ3StyleObject Q3FillStyle_New (TQ3FillStyle fillStyle);
```

`fillStyle` A fill style value.

DESCRIPTION

The `Q3FillStyle_New` function returns, as its function result, a new style object having the fill style specified by the `fillStyle` parameter. The `fillStyle` parameter should be one of these values:

```
kQ3FillStyleFilled
kQ3FillStyleEdges
kQ3FillStylePoints
```

If a new style object could not be created, `Q3FillStyle_New` returns the value `NULL`.

To change the current fill style, you must actually draw the style object. You can call `Q3Style_Submit` to draw the style in retained mode or `Q3FillStyle_Submit` (described next) to draw the style in immediate mode.

SEE ALSO

See "Fill Styles" (page 548) for a description of the available fill styles.

Q3FillStyle_Submit

You can use the `Q3FillStyle_Submit` function to submit a fill style in immediate mode.

```
TQ3Status Q3FillStyle_Submit (
                    TQ3FillStyle fillStyle,
                    TQ3ViewObject view);
```

`fillStyle` A fill style value.

`view` A view.

DESCRIPTION

The `Q3FillStyle_Submit` function sets the fill style of the view specified by the `view` parameter to the style specified in the `fillStyle` parameter.

SPECIAL CONSIDERATIONS

You should call `Q3FillStyle_Submit` only in a submitting loop.

Q3FillStyle_Get

You can use the `Q3FillStyle_Get` function to get the fill style value of a fill style.

```
TQ3Status Q3FillStyle_Get (
                    TQ3StyleObject styleObject,
                    TQ3FillStyle *fillStyle);
```

`styleObject` A fill style object.

`fillStyle` On exit, a pointer to the fill style value of the specified fill style object.

DESCRIPTION

The `Q3FillStyle_Get` function returns, in the `fillStyle` parameter, a pointer to the current fill style value of the fill style object specified by the `styleObject` parameter.

Q3FillStyle_Set

You can use the `Q3FillStyle_Set` function to set the fill style value of a fill style.

```
TQ3Status Q3FillStyle_Set (
                    TQ3StyleObject styleObject,
                    TQ3FillStyle fillStyle);
```

`styleObject` A fill style object.

`fillStyle` A fill style value.

DESCRIPTION

The `Q3FillStyle_Set` function sets the fill style value of the style object specified by the `styleObject` parameter to the value specified in the `fillStyle` parameter.

Managing Highlight Styles

QuickDraw 3D provides routines that you can use to manage highlight styles.

Q3HighlightStyle_New

You can use the `Q3HighlightStyle_New` function to create a new highlight style object.

```
TQ3StyleObject Q3HighlightStyle_New (
                    TQ3AttributeSet highlightAttribute);
```

`highlightAttribute`
An attribute set.

DESCRIPTION

The `Q3HighlightStyle_New` function returns, as its function result, a new style object having the highlight style specified by the `highlightAttribute` parameter. The `highlightAttribute` parameter should be a reference to an attribute set.

If a new style object could not be created, `Q3HighlightStyle_New` returns the value `NULL`.

To change the current highlight style, you must actually draw the style object. You can call `Q3Style_Submit` to draw the style in retained mode or `Q3HighlightStyle_Submit` (described next) to draw the style in immediate mode.

SEE ALSO

See "Highlight Styles" (page 548) for a description of highlight styles.

Q3HighlightStyle_Submit

You can use the `Q3HighlightStyle_Submit` function to submit a highlight style in immediate mode.

```
TQ3Status Q3HighlightStyle_Submit (
                    TQ3AttributeSet highlightAttribute,
                    TQ3ViewObject view);
```

`highlightAttribute`
An attribute set.

`view` A view.

DESCRIPTION

The `Q3HighlightStyle_Submit` function sets the highlight style of the view specified by the `view` parameter to the style specified in the `highlightAttribute` parameter.

SPECIAL CONSIDERATIONS

You should call `Q3HighlightStyle_Submit` only in a submitting loop.

Q3HighlightStyle_Get

You can use the `Q3HighlightStyle_Get` function to get the highlight style value of a highlight style.

```
TQ3Status Q3HighlightStyle_Get (
                    TQ3StyleObject highlight,
                    TQ3AttributeSet *highlightAttribute);
```

`highlight` A highlight style object.

`highlightAttribute`
On exit, a pointer to the attribute set of the specified highlight style object.

DESCRIPTION

The `Q3HighlightStyle_Get` function returns, in the `highlightAttribute` parameter, a pointer to the current attribute set of the style object specified by the `highlight` parameter.

Q3HighlightStyle_Set

You can use the `Q3HighlightStyle_Set` function to set the highlight style value of a highlight style.

```
TQ3Status Q3HighlightStyle_Set (
                    TQ3StyleObject highlight,
                    TQ3AttributeSet highlightAttribute);
```

`highlight` A highlight style object.

`highlightAttribute`
An attribute set.

DESCRIPTION

The `Q3HighlightStyle_Set` function sets the highlight style value of the style object specified by the `highlight` parameter to the attribute set specified in the `highlightAttribute` parameter.

Managing Subdivision Styles

QuickDraw 3D provides routines that you can use to manage subdivision styles.

Q3SubdivisionStyle_New

You can use the `Q3SubdivisionStyle_New` function to create a new subdivision style object.

```
TQ3StyleObject Q3SubdivisionStyle_New (
                    const TQ3SubdivisionStyleData *data);
```

`data` A pointer to a subdivision style data structure.

DESCRIPTION

The `Q3SubdivisionStyle_New` function returns, as its function result, a new style object having the subdivision style specified by the `data` parameter. The `method` field of the subdivision style data structure pointed to by the `data` parameter should be one of these values:

```
kQ3SubdivisionMethodConstant
kQ3SubdivisionMethodWorldSpace
kQ3SubdivisionMethodScreenSpace
```

The meaning of the `c1` and `c2` fields depends on the value of the `method` field. See "Subdivision Style Data Structure" (page 555) for details.

If a new style object could not be created, `Q3SubdivisionStyle_New` returns the value `NULL`.

To change the current subdivision style, you must actually draw the style object. You can call `Q3Style_Submit` to draw the style in retained mode or `Q3SubdivisionStyle_Submit` to draw the style in immediate mode.

SEE ALSO

See "Subdivision Styles" (page 549) for a description of subdivision styles.

Q3SubdivisionStyle_Submit

You can use the `Q3SubdivisionStyle_Submit` function to submit a subdivision style in immediate mode.

```
TQ3Status Q3SubdivisionStyle_Submit (
                    const TQ3SubdivisionStyleData *data,
                    TQ3ViewObject view);
```

`data`	A pointer to a subdivision style data structure.
`view`	A view.

DESCRIPTION

The `Q3SubdivisionStyle_Submit` function sets the subdivision style of the view specified by the `view` parameter to the style specified by the `data` parameter.

SPECIAL CONSIDERATIONS

You should call `Q3SubdivisionStyle_Submit` only in a submitting loop.

Q3SubdivisionStyle_GetData

You can use the `Q3SubdivisionStyle_GetData` function to get the subdivision style method and specifiers of a subdivision style.

```
TQ3Status Q3SubdivisionStyle_GetData (
                    TQ3StyleObject subdiv,
                    TQ3SubdivisionStyleData *data);
```

`subdiv`	A subdivision style object.
`data`	On exit, a pointer to a subdivision style data structure.

DESCRIPTION

The `Q3SubdivisionStyle_GetData` function returns, in the `data` parameter, a pointer to a subdivision style data structure for the style object specified by the `subdiv` parameter.

Q3SubdivisionStyle_SetData

You can use the `Q3SubdivisionStyle_SetData` function to set the subdivision style method and specifiers of a subdivision style.

```
TQ3Status Q3SubdivisionStyle_SetData (
                    TQ3StyleObject subdiv,
                    const TQ3SubdivisionStyleData *data);
```

`subdiv` A subdivision style object.

`data` A pointer to a subdivision style data structure.

DESCRIPTION

The `Q3SubdivisionStyle_SetData` function sets the subdivision style values of the style object specified by the `subdiv` parameter to the values specified in the `data` parameter.

Managing Orientation Styles

QuickDraw 3D provides routines that you can use to manage orientation styles.

Q3OrientationStyle_New

You can use the `Q3OrientationStyle_New` function to create a new orientation style object.

```
TQ3StyleObject Q3OrientationStyle_New (
                    TQ3OrientationStyle frontFacingDirection);
```

`frontFacingDirection`
An orientation style value.

DESCRIPTION

The `Q3OrientationStyle_New` function returns, as its function result, a new style object having the orientation style specified by the `frontFacingDirection` parameter. The `frontFacingDirection` parameter should be one of these values:

```
kQ3OrientationStyleCounterClockwise
kQ3OrientationStyleClockwise
```

If a new style object could not be created, `Q3OrientationStyle_New` returns the value `NULL`.

To change the current orientation style, you must actually draw the style object. You can call `Q3Style_Submit` to draw the style in retained mode or `Q3OrientationStyle_Submit` (described next) to draw the style in immediate mode.

SEE ALSO

See "Orientation Styles" (page 550) for a description of orientation styles.

Q3OrientationStyle_Submit

You can use the `Q3OrientationStyle_Submit` function to submit a orientation style in immediate mode.

```
TQ3Status Q3OrientationStyle_Submit (
                    TQ3OrientationStyle frontFacingDirection,
                    TQ3ViewObject view);
```

`frontFacingDirection`
An orientation style value.

`view` A view.

DESCRIPTION

The `Q3OrientationStyle_Submit` function sets the orientation style of the view specified by the `view` parameter to the style specified by the `frontFacingDirection` parameter.

SPECIAL CONSIDERATIONS

You should call `Q3OrientationStyle_Submit` only in a submitting loop.

Q3OrientationStyle_Get

You can use the `Q3OrientationStyle_Get` function to get the orientation style value of an orientation style.

```
TQ3Status Q3OrientationStyle_Get (
                    TQ3StyleObject frontFacingDirectionObject,
                    TQ3OrientationStyle *frontFacingDirection);
```

`frontFacingDirectionObject`
An orientation style object.

`frontFacingDirection`
On exit, a pointer to the orientation style value of the specified orientation style object.

DESCRIPTION

The `Q3OrientationStyle_Get` function returns, in the `frontFacingDirection` parameter, a pointer to the current orientation style value of the style object specified by the `frontFacingDirectionObject` parameter.

Q3OrientationStyle_Set

You can use the `Q3OrientationStyle_Set` function to set the orientation style value of a orientation style.

```
TQ3Status Q3OrientationStyle_Set (
                    TQ3StyleObject frontFacingDirectionObject,
                    TQ3OrientationStyle frontFacingDirection);
```

`frontFacingDirectionObject`
An orientation style object.

`frontFacingDirection`
An orientation style value.

DESCRIPTION

The `Q3OrientationStyle_Set` function sets the orientation style value of the style object specified by the `frontFacingDirectionObject` parameter to the value specified in the `frontFacingDirection` parameter.

Managing Shadow-Receiving Styles

QuickDraw 3D provides routines that you can use to manage shadow-receiving styles.

Q3ReceiveShadowsStyle_New

You can use the `Q3ReceiveShadowsStyle_New` function to create a new shadow-receiving style object.

```
TQ3StyleObject Q3ReceiveShadowsStyle_New (TQ3Boolean receives);
```

`receives` A Boolean value that determines whether the new style object specifies that objects in the scene receive shadows (`kQ3True`) or do not receive shadows (`kQ3False`).

DESCRIPTION

The `Q3ReceiveShadowsStyle_New` function returns, as its function result, a new style object having the shadow-receiving style specified by the `receives` parameter.

If a new style object could not be created, `Q3ReceiveShadowsStyle_New` returns the value `NULL`.

To change the current shadow-receiving style, you must actually draw the style object. You can call `Q3Style_Submit` to draw the style in retained mode or `Q3ReceiveShadowsStyle_Submit` (described next) to draw the style in immediate mode.

SEE ALSO

See "Shadow-Receiving Styles" (page 551) for a description of shadow-receiving styles.

Q3ReceiveShadowsStyle_Submit

You can use the `Q3ReceiveShadowsStyle_Submit` function to submit a shadow-receiving style in immediate mode.

```
TQ3Status Q3ReceiveShadowsStyle_Submit (
                    TQ3Boolean receives,
                    TQ3ViewObject view);
```

`receives` A Boolean value that determines whether objects in the scene receive shadows (`kQ3True`) or do not receive shadows (`kQ3False`).

`view` A view.

DESCRIPTION

The `Q3ReceiveShadowsStyle_Submit` function sets the shadow-receiving style of the view specified by the `view` parameter to the style specified by the `receives` parameter.

SPECIAL CONSIDERATIONS

You should call `Q3ReceiveShadowsStyle_Submit` only in a submitting loop.

Q3ReceiveShadowsStyle_Get

You can use the `Q3ReceiveShadowsStyle_Get` function to get the shadow-receiving style value of a shadow-receiving style.

```
TQ3Status Q3ReceiveShadowsStyle_Get (
                    TQ3StyleObject styleObject,
                    TQ3Boolean *receives);
```

styleObject	A shadow-receiving style object.
receives	On exit, a pointer to the shadow-receiving style value of the specified shadow-receiving style object.

DESCRIPTION

The Q3ReceiveShadowsStyle_Get function returns, in the receives parameter, a pointer to the current shadow-receiving style value of the style object specified by the styleObject parameter.

Q3ReceiveShadowsStyle_Set

You can use the Q3ReceiveShadowsStyle_Set function to set the shadow-receiving style value of a shadow-receiving style.

```
TQ3Status Q3ReceiveShadowsStyle_Set (
                    TQ3StyleObject styleObject,
                    TQ3Boolean receives);
```

styleObject	A shadow-receiving style object.
receives	A Boolean value that determines whether objects in the scene receive shadows (kQ3True) or do not receive shadows (kQ3False).

DESCRIPTION

The Q3ReceiveShadowsStyle_Set function sets the shadow-receiving style value of the style object specified by the styleObject parameter to the value specified in the receives parameter.

Managing Picking ID Styles

QuickDraw 3D provides routines that you can use to manage picking ID styles.

Q3PickIDStyle_New

You can use the `Q3PickIDStyle_New` function to create a new picking ID style object.

```
TQ3StyleObject Q3PickIDStyle_New (unsigned long id);
```

`id` A picking ID.

DESCRIPTION

The `Q3PickIDStyle_New` function returns, as its function result, a new style object having the picking ID specified by the `id` parameter. If a new style object could not be created, `Q3PickIDStyle_New` returns the value `NULL`.

SEE ALSO

See "Picking ID Styles" (page 552) for a description of picking ID styles.

Q3PickIDStyle_Submit

You can use the `Q3PickIDStyle_Submit` function to submit a picking ID style in immediate mode.

```
TQ3Status Q3PickIDStyle_Submit (
                    unsigned long id,
                    TQ3ViewObject view);
```

`id` A picking ID.

`view` A view.

DESCRIPTION

The `Q3PickIDStyle_Submit` function sets the picking ID of the view specified by the `view` parameter to the value specified by the `id` parameter.

SPECIAL CONSIDERATIONS

You should call `Q3PickIDStyle_Submit` only in a submitting loop.

Q3PickIDStyle_Get

You can use the `Q3PickIDStyle_Get` function to get the picking ID style value of a picking ID style.

```
TQ3Status Q3PickIDStyle_Get (
                    TQ3StyleObject pickIDObject,
                    unsigned long *id);
```

`pickIDObject` A picking ID style object.

`id` On exit, the picking ID of the specified picking ID style object.

DESCRIPTION

The `Q3PickIDStyle_Get` function returns, in the `id` parameter, the current picking ID of the style object specified by the `pickIDObject` parameter.

Q3PickIDStyle_Set

You can use the `Q3PickIDStyle_Set` function to set the picking ID of a picking ID style.

```
TQ3Status Q3PickIDStyle_Set (
                    TQ3StyleObject pickIDObject,
                    unsigned long id);
```

`pickIDObject` A picking ID style object.

`id` A picking ID.

DESCRIPTION

The `Q3PickIDStyle_Set` function sets the picking ID of the style object specified by the `pickIDObject` parameter to the value specified in the `id` parameter.

Managing Picking Parts Styles

QuickDraw 3D provides routines that you can use to manage picking parts styles.

Q3PickPartsStyle_New

You can use the `Q3PickPartsStyle_New` function to create a new picking parts style object.

```
TQ3StyleObject Q3PickPartsStyle_New (TQ3PickParts parts);
```

`parts` A picking parts style value.

DESCRIPTION

The `Q3PickPartsStyle_New` function returns, as its function result, a new style object having the picking parts style specified by the `parts` parameter. See page 552 for a list of masks you can use to construct a picking parts style value.

If a new style object could not be created, `Q3PickPartsStyle_New` returns the value `NULL`.

To change the current picking parts style, you must actually draw the style object. You can call `Q3Style_Submit` to draw the style in retained mode or `Q3PickPartsStyle_Submit` (described next) to draw the style in immediate mode.

SEE ALSO

See “Picking Parts Styles” (page 552) for a description of picking parts styles.

Q3PickPartsStyle_Submit

You can use the `Q3PickPartsStyle_Submit` function to submit a picking parts style in immediate mode.

```
TQ3Status Q3PickPartsStyle_Submit (
                    TQ3PickParts parts,
                    TQ3ViewObject view);
```

`parts` A picking parts style value.

`view` A view.

DESCRIPTION

The `Q3PickPartsStyle_Submit` function sets the picking parts style of the view specified by the `view` parameter to the style specified by the `parts` parameter.

SPECIAL CONSIDERATIONS

You should call `Q3PickPartsStyle_Submit` only in a submitting loop.

Q3PickPartsStyle_Get

You can use the `Q3PickPartsStyle_Get` function to get the picking parts style value of a picking parts style.

```
TQ3Status Q3PickPartsStyle_Get (
                    TQ3StyleObject pickPartsObject,
                    TQ3PickParts *parts);
```

`pickPartsObject`
A picking parts style object.

`parts` On entry, a pointer to a variable of type `TQ3PickParts`. On exit, the current picking parts style value of the specified style object.

DESCRIPTION

The `Q3PickPartsStyle_Get` function returns, in the `parts` parameter, a pointer to the current picking parts value of the style object specified by the `pickPartsObject` parameter. See page 552 for a list of masks used to construct a picking parts value.

Q3PickPartsStyle_Set

You can use the `Q3PickPartsStyle_Set` function to set the picking parts style value of a picking parts style.

```
TQ3Status Q3PickPartsStyle_Set (
                    TQ3StyleObject pickPartsObject,
                    TQ3PickParts parts);
```

`pickPartsObject`
A picking parts style object.

`parts` A picking parts style value.

DESCRIPTION

The `Q3PickPartsStyle_Set` function sets the picking parts style value of the style object specified by the `pickPartsObject` parameter to the value specified in the `parts` parameter.

Managing the Anti-Alias Style

QuickDraw 3D provides routines that you can use to manage the anti-alias style.

Q3AntiAliasStyle_New

You can use the `Q3AntiAliasStyle_New` function to create a `TQ3StyleObject` object. When the object is rendered, it sets the renderer's anti-aliasing style.

```
TQ3StyleObject Q3AntiAliasStyle_New (
                    TQ3AntiAliasStyleData *aaStyleData);
```

`aaStyleData` A `TQ3AntiAliasStyleData` structure.

return value An anti-alias style object.

DESCRIPTION

The `Q3AntiAliasStyle_New` function returns, in `TQ3StyleObject`, the anti-alias style object determined by the structure pointed to by `aaStyleData`.

Q3AntiAliasStyle_Submit

You can use the `Q3AntiAliasStyle_Submit` function to set the current anti-aliasing mode state in the view.

```
TQ3Status Q3AntiAliasStyle_Submit (
                    const TQ3AntiAliasStyleData    *aaStyleData,
                    TQ3ViewObject                  view);
```

`aaStyleData` A `TQ3AntiAliasStyleData` structure.

`view` A view.

DESCRIPTION

The `Q3AntiAliasStyle_Submit` function sets the current anti-aliasing mode state determined by `aaStyleData` in the view designated by `view`.

SPECIAL CONSIDERATIONS

You can call the `Q3AntiAliasStyle_Submit` function while rendering a frame, between the `Q3View_StartRendering` and `Q3View_EndRendering` calls. However, many renderers cannot change the state of anti-aliasing while rendering a frame, so only the first call to `Q3AntiAliasStyle_Submit` may have an effect.

Q3AntiAliasStyle_GetData

You can use the `Q3AntiAliasStyle_GetData` function to fetch the `TQ3AntiAliasStyleData` structure currently associated with an anti-alias style object.

```
TQ3Status Q3AntiAliasStyle_GetData (
                    TQ3StyleObject          styleObject,
                    TQ3AntiAliasStyleData   *aaStyleData);
```

`styleObject` An anti-alias style object.

`aaStyleData` A `TQ3AntiAliasStyleData` structure.

DESCRIPTION

The `Q3AntiAliasStyle_GetData` function returns, in the `aaStyleData` parameter, the current `TQ3AntiAliasStyleData` structure for the anti-alias style object designated by `styleObject`.

Q3AntiAliasStyle_SetData

You can use the `Q3AntiAliasStyle_SetData` function to determine the `TQ3AntiAliasStyleData` structure currently associated with an anti-alias style object.

```
TQ3Status Q3AntiAliasStyle_SetData (
                    TQ3StyleObject              styleObject,
                    const TQ3AntiAliasStyleData *aaStyleData);
```

`styleObject` An anti-alias style object.

`aaStyleData` A `TQ3AntiAliasStyleData` structure.

DESCRIPTION

The `Q3AntiAliasStyle_SetData` function sets the `TQ3AntiAliasStyleData` structure as the current source of anti-alias style control for the anti-alias style object designated by `styleObject`.

CHAPTER 6

Style Objects

CHAPTER 7

Transform Objects

This chapter describes transform objects (or transforms) and the functions you can use to create and manipulate them. You can use transforms to change the position, size, or orientation of a geometric object. QuickDraw 3D uses numerous transforms internally, for example, when creating a two-dimensional image of a three-dimensional model. QuickDraw 3D supports a number of types of transforms, including translate, scaling, rotation, and arbitrary affine transforms.

You should read this chapter for general information about the types of transforms supported by QuickDraw 3D and for specific information about applying transforms to objects in your models. See the chapter "View Objects" for routines that you can use to get information about the transforms that QuickDraw 3D uses internally when rendering a model.

This chapter begins by describing transform objects and their features. It also describes the various coordinate systems or spaces supported by QuickDraw 3D. The section "Transform Objects Reference," beginning on page 599 provides a complete description of transform objects and the routines you can use to create and manipulate them.

About Transform Objects

A **transform object** (or, more briefly, a **transform**) is an object that you can use to modify or transform the appearance or behavior of drawable QuickDraw 3D objects. You use transforms to reposition and reorient geometric shapes in space. Transforms are useful because they do not alter the geometric representation of objects (that is, the vertices or other values that define a geometric object); rather, they are applied as matrices at rendering time, temporarily "moving" an object in space. Thus you can reference a single object

multiple times with different transforms and can place an object in many different locations within a model.

A transform is of type `TQ3TransformObject`, which is a type of shape object. QuickDraw 3D defines these basic types of transforms:

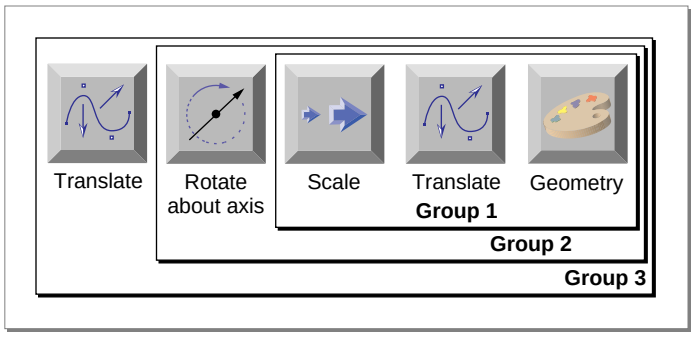
- matrix transforms
- translate transforms
- scale transforms
- rotate transforms
- rotate-about-point transforms
- rotate-about-axis transforms
- quaternion transforms

No matter how you specify a transform, QuickDraw 3D maintains its data in that form until you begin to render an image, at which time it converts the data to a temporary matrix that is applied to the objects it governs. Because transforms are a type of shape object, you apply a transform by drawing it into a view or by putting it into a group. If you draw a transform in a view, you can use either retained or immediate transforms.

When you apply several transforms to a vector, the transform matrices are premultiplied to the vector. For example, in the multiplication v[A][B]...[M] of the vector v by the matrices A, B,..., M, matrix A is first applied to the vector, then B, and so forth. Accordingly, you should specify transforms to be concatenated in the reverse order that you want to apply them. This scheme is consistent with the application of matrices in a hierarchy, in which matrices at the top of a hierarchy are applied last.

For example, consider the very simple model illustrated in Figure 7-1, which consists of three separate groups. A geometric object is first grouped with a scale and a translate transform (the translate transform was added to the group before the scale transform was added); the resulting group is then grouped with a rotate-about-axis transform, and that group is finally grouped with a second translate transform.

Figure 7-1 A simple model illustrating the order in which transforms are applied



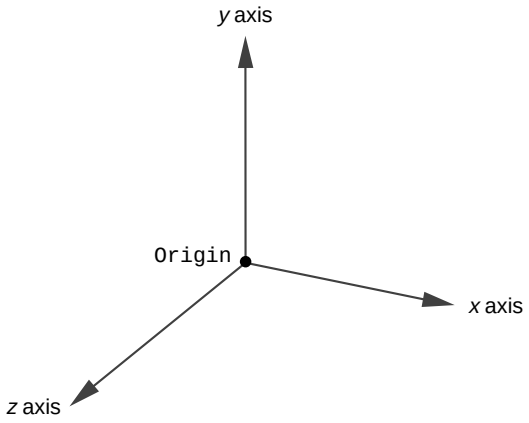
When this model is rendered, the transforms are applied to the geometric object in this order: scale, translate (group 1), rotate-about-axis (group 2), translate (group 3). Your application should add transforms to a group in the reverse order they are to be rendered. That is, in the example, you would first add the translate transform to Group 1 and then add the scale transform.

Note
For information about creating groups of QuickDraw 3D objects, see the chapter "Group Objects." ◆

Spaces

A **coordinate system** (or **space**) is any system of assigning planar or spatial positions to objects. In general, QuickDraw 3D operates with rectilinear or **Cartesian coordinate systems,** in which the position of a point in a plane or in space is determined by projecting the point onto the **coordinate axes,** which are mutually perpendicular lines that intersect at a point called the *origin*. By convention, the **origin** is the planar point (0, 0) or the spatial point (0, 0, 0). Figure 7-2 shows a Cartesian coordinate system that is **right-handed** (that is, if the thumb of the right hand points in the direction of the positive x axis and the index finger points in the direction of the positive y axis, then the middle finger points in the direction of the positive z axis).

Figure 7-2 A right-handed Cartesian coordinate system



Note
You can, for certain purposes, specify positions using other types of coordinate systems, such as the **polar coordinate system** (a system of assigning planar positions to objects in terms of their distances *r* from the origin along a ray that forms a given angle θ with a fixed coordinate line) or the **spherical coordinate system** (a system of assigning spatial positions to objects in terms of their distances *r* from the origin along a ray that forms a given angle θ with a fixed coordinate line and another angle ϕ with another fixed coordinate line). QuickDraw 3D provides routines you can use to convert among these three types of coordinate systems. See the chapter "Mathematical Utilities" for details. Unless noted differently, this book always uses Cartesian coordinate systems. ◆

QuickDraw 3D, like virtually all other 3D graphics systems, defines several distinct coordinate systems and maintains transforms that it uses to convert one coordinate system into another.

Because it's often useful to define an object once and then to create multiple copies of that object for placement at different positions and orientations, QuickDraw 3D supports a **local coordinate system** for each object you define. An object's local coordinate system is simply the coordinate system in which it

is specified (that is, that determines the values you specify in the relevant data structure). Any given object can be defined using any of infinitely many local coordinate systems. Usually, you'll pick a local coordinate system whose origin coincides with some part of the object. For instance, it's quite natural to define a box using a local coordinate system whose origin is at the box's origin, and whose axes coincide with the box's axes.

Note
A local coordinate system is sometimes called an **object coordinate system** or a **modeling coordinate system,** and the space it defines is the **object space** or **modeling space.** ◆

The **world coordinate system** (or **world space**) defines the locations of all geometric objects as they exist at rendering or picking time, with all applicable transforms acting on them. It's important to note that world space is relevant only within a submitting loop, because the transforms that relocate or reorient an object must be applied to the object to determine its position and orientation in world coordinates.

Note
The world coordinate system is sometimes called the **global coordinate system** or the **application coordinate system,** and the space it defines is the **global space** or **application space.** ◆

You can create copies of an object and place them at different locations by applying different transforms to each copy. A transform changes an object's position or orientation in world coordinates, but not its local coordinates. In other words, if you use the function `Q3Box_GetOrigin` with two copies of a single box, the function always returns the same origin for each box, whether or not transforms have been applied to one or both of the copies.

The relationship between an object's local coordinate system and the world coordinate system is specified by that object's **local-to-world transform.** For objects that have no transforms applied to them at rendering time, the local-to-world transform can be represented by the identity matrix, in which case the local coordinate system of that object and the world coordinate system coincide. If one or more transforms is applied to the object at rendering time, the world space location of the object is determined by taking its local space position and applying the transforms to it.

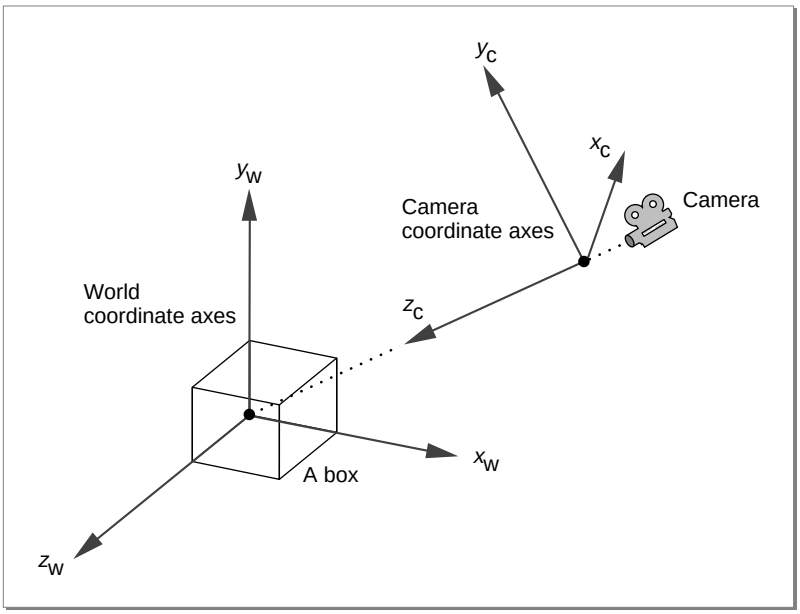
A world coordinate system defines the relative positions and sizes of geometric objects. When an object is rendered in a view, the view's camera specifies yet another coordinate system, the **camera coordinate system** (or **camera space**). A camera coordinate system is defined by the camera placement structure associated with the camera, which is defined like this:

```
typedef struct TQ3CameraPlacement {
    TQ3Point3D                          cameraLocation;
    TQ3Point3D                          pointOfInterest;
    TQ3Vector3D                         upVector;
} TQ3CameraPlacement;
```

Note
See the chapter "Camera Objects" for complete information about the camera placement structure. ◆

The `cameraLocation` field specifies the origin of the camera coordinate system. The `pointOfInterest` field specifies the *z* axis of the camera coordinate system, and the `upVector` field specifies the *y* axis of the camera coordinate system. The *x* axis of the camera coordinate system is determined by the left-hand rule. Figure 7-3 shows a camera coordinate system and its relation to the world coordinate system. In this figure, the camera is set to take an isometric view of the box whose origin is at the origin of the world coordinate system.

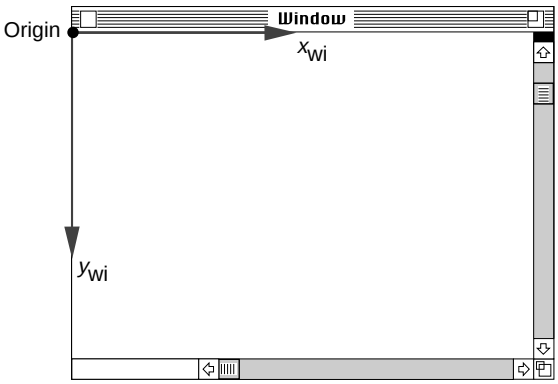
Figure 7-3 A camera coordinate system



As you know, a camera specifies a method of projecting a three-dimensional model onto a two-dimensional plane, called the **view plane.** The camera, the view plane, and the hither and yon clipping planes together define the part of the model that is projected onto that view plane. As you can see in Figure 9-7 (page 679), these objects define a rectangular frustum known as the **viewing box.** When perspective camera is used, the camera, the view plane, and the hither and yon clipping planes define a pyramidal frustum known as the **viewing frustum** (see Figure 9-5 (page 677)). Because a camera and its camera coordinate system determine a unique view frustum, camera space is also called **frustum space.**

The final step in creating an image of a model is to map the two-dimensional image projected onto the view plane into the draw context associated with a view. In general, the draw context specifies a window on a screen or other display device that is to contain all or part of the view plane image. Accordingly, QuickDraw 3D maintains, for each draw context, a **window coordinate system** (or **window space**) that defines the position of objects in the draw context. Figure 7-4 shows a window coordinate system.

Figure 7-4 A window coordinate system



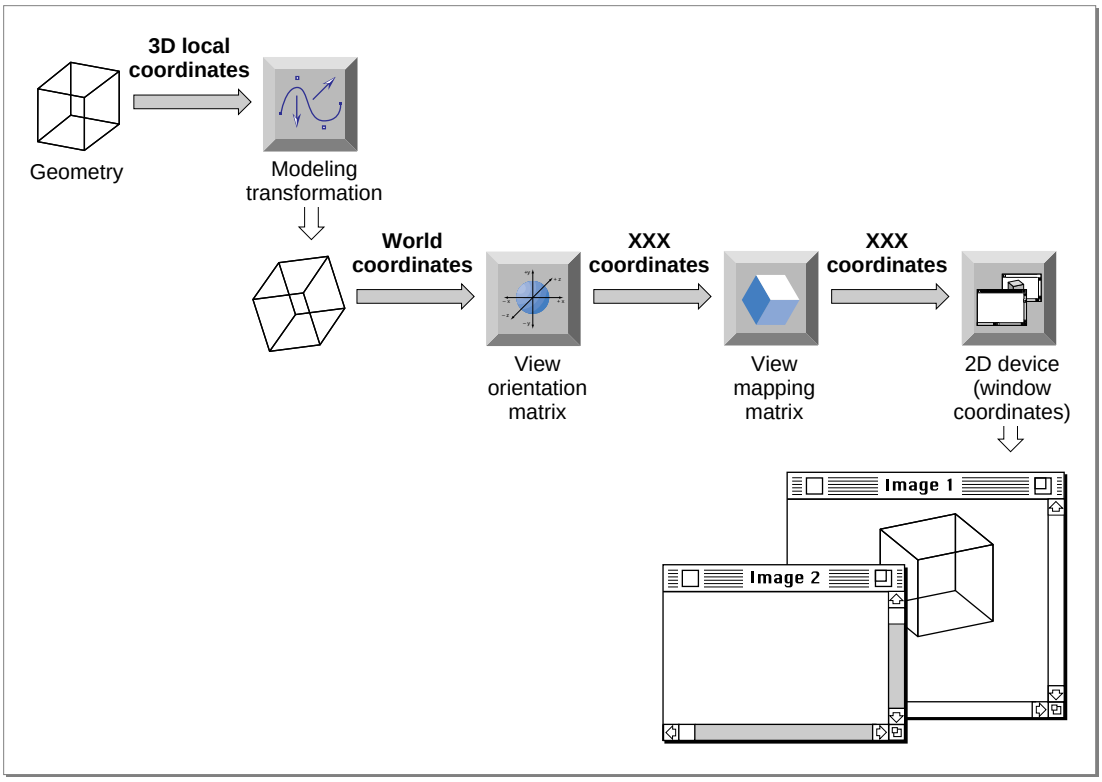
Note
A window coordinate system is sometimes called a **screen coordinate system** or a **draw context coordinate system,** and the space it defines is the **screen space** or **draw context space.** ◆

In addition to the local-to-world transform (which defines the relationship between an object's local coordinate system and the world coordinate system), QuickDraw 3D also maintains a **world-to-frustum transform** (which defines the relationship between the world coordinate system and the frustum coordinate system) and a **frustum-to-window transform** (which defines the relationship between a frustum coordinate system and a window coordinate system). See Figure 7-5. You can, if necessary, get a matrix representation of these three transforms. See the chapter "View Objects" for details.

The world-to-frustum transform is actually the product of two transforms specified by matrices, the view orientation matrix and the view mapping matrix. The **view orientation matrix** rotates and translates the view's camera so that it is pointing down the negative z axis. The **view mapping matrix** transforms the viewing frustum into a standard rectangular solid. This standard rectangular solid is a box containing x values from –1 to 1, y values from –1 to 1, and z values from 0 to –1. The far clipping plane is the plane defined by the equation $z = -1$, and the near clipping plane is the plane defined by the equation $z = 0$.

With a perspective camera, the view mapping matrix performs most of the work of projection. The objects transformed by the world-to-frustum transform are still 3D, but it's easy to get the 2D projection onto the view plane by simply dropping the *z* coordinate of each rendered point.

Figure 7-5 View state transformations



Types of Transforms

QuickDraw 3D supports a number of different ways of transforming geometric objects. Equivalently, these transforms are ways of transforming coordinate systems containing geometric objects.

Matrix Transforms

A **matrix transform** is any transform specified by an affine, invertible 4-by-4 matrix. QuickDraw 3D does not check that the matrix you specify is affine or invertible, so it is your responsibility to ensure that the matrix has these qualities.

A matrix transform is the most general type of transform and can be used to represent any of the other kinds of transforms. If, however, you just want to apply a translation to an object, it's better to use a translate transform instead of a matrix transform. By using the more specific type of transform object, you allow renderers and shaders to apply optimizations that might not apply to a more general transform.

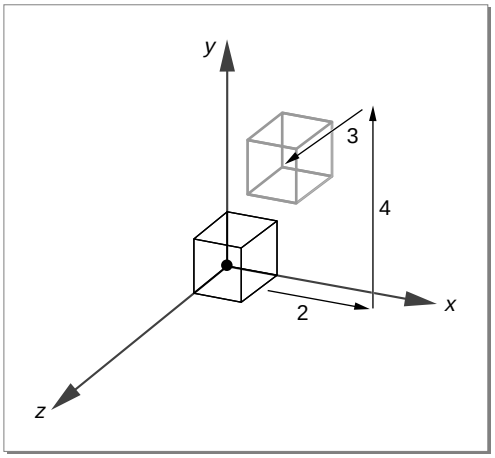
Translate Transforms

A **translate transform** translates an object along the *x*, *y*, and *z* axes by specified values. You specify the desired translation values using a vector. For example, to translate an object by 2 units along the positive *x* axis, by 4 units along the positive *y* axis, and by 3 units along the positive *z* axis, you could define a vector like this:

```
TQ3Vector3D                     myVector;
TQ3TransformObject              myTransform;

Q3Vector3D_Set(&myVector, 2.0, 4.0, 3.0);
myTransform = Q3TranslateTransform_New(&myVector);
```

Figure 7-6 shows a unit cube before and after a translate transform is applied.

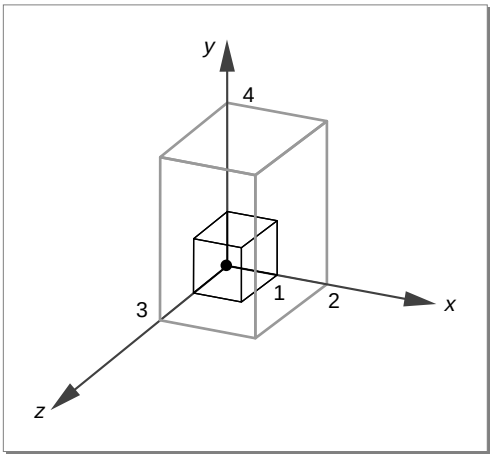
Figure 7-6 A translate transform



Scale Transforms

A **scale transform** scales an object along the *x*, *y*, and *z* axes by specified values. Figure 7-7 shows a unit cube before and after applying a scale transform.

Figure 7-7 A scale transform



As with a translate transform, you specify the desired scale transform by using a vector. For example, to scale an object by a factor of 2 along the positive *x* axis, by a factor of 4 along the positive *y* axis, and by a factor of 3 along the positive *z* axis, you could define a vector like this:

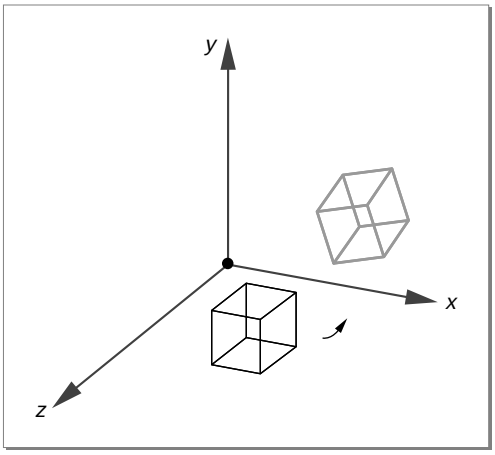
```
TQ3Vector3D             myVector;

Q3Vector3D_Set(&myVector, 2.0, 4.0, 3.0);
```

Rotate Transforms

A **rotate transform** rotates an object about the *x*, *y*, or *z* axis by a specified number of radians at the origin.

To specify a rotate transform, you fill in the fields of a **rotate transform data structure,** which specifies the axis of rotation and the number of radians to rotate. You can use QuickDraw 3D macros to convert degrees to radians, if you prefer to work with degrees. (See the chapter "Mathematical Utilities" for details.) Figure 7-8 shows a unit cube before and after applying a rotate transform.

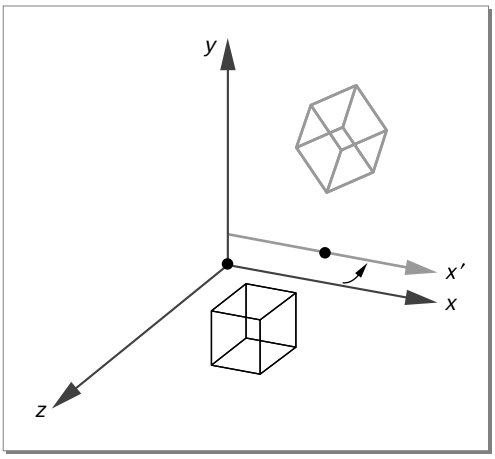
Figure 7-8 A rotate transform



Rotate-About-Point Transforms

A **rotate-about-point transform** rotates an object about the *x*, *y*, or *z* axis by a specified number of radians at an arbitrary point in space. To specify a rotate-about-point transform, you fill in the fields of a rotate-about-point transform data structure, which specifies the axis of rotation, the point of rotation, and the number of radians to rotate. Figure 7-9 shows a unit cube before and after applying a rotate-about-point transform.

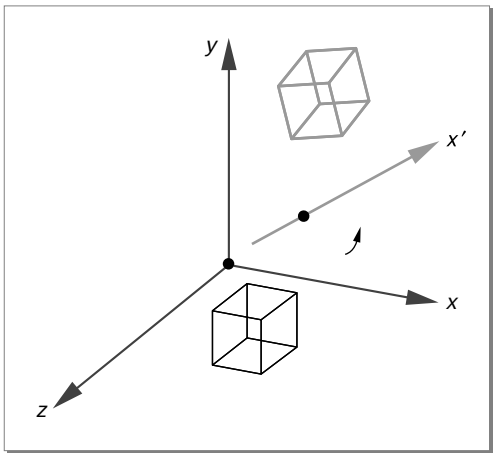
Figure 7-9 A rotate-about-point transform



Rotate-About-Axis Transforms

A **rotate-about-axis transform** rotates an object about an arbitrary axis in space by a specified number of radians at an arbitrary point in space. To specify a rotate-about-axis transform, you fill in the fields of a rotate-about-axis transform data structure, which specifies the axis of rotation, the point of rotation, and the number of radians to rotate. Figure 7-10 shows a unit cube before and after applying a rotate-about-axis transform.

Figure 7-10 A rotate-about-axis transform



Quaternion Transforms

A **quaternion transform** rotates and twists an object according to the mathematical properties of quaternions.

The Reset Transform

When transforms are submitted to a view, either directly through an immediate-mode call such as `Q3ScaleTransform_Submit` or a retained-mode call such as `Q3Transform_Submit`, or indirectly by inclusion in a group, the view's current transformation matrix is concatenated with the submitted matrix to form the new current transformation matrix.

An application can take advantage of this transformation stacking behavior in hierarchical modelling; the view can maintain a stack of transformations that the application may push and pop. The application can do this explicitly, using `Q3View_Push` and `Q3View_Pop`, or implicitly by using groups, which push and pop transformations when they are entered and exited. In a push, the current transformation is pushed onto a stack, but a copy remains as the current transfomation. Subsequent transformation submissions concatenate their matrices with the current transformation.

However, this behavior is not always desirable. Suppose an application traverses a hierarchy, either in mixed or immediate mode, using sequences of

push-transform-polygon-pop actions. If the application wants to draw a shape untransformed in the middle of such a sequence, (because, for example, the shape is already drawn in world space coordinates), then the application would have to get the current transformation, invert it, and submit the inverted transformation, thereby resolving the current transformation in the view to the identity matrix. Such a sequence of actions would usually be bracketed by a push-pop sequence. Because obtaining the current matrix, inverting it, and submitting the inverted matrix require significant amounts of processing time, and because inversion cannot be perfectly precise (because of floating-point approximations), QuickDraw 3D includes a **reset transform.** It resets the current transformation to identity (that is, equivalent to the 4x4 identity matrix).

The routines that implement the reset transform are described in "Creating and Submitting the Reset Transform," beginning on page 629.

Transform Objects Reference

This section describes the QuickDraw 3D data structures and routines that you can use to manage transforms.

Data Structures

QuickDraw 3D defines a number of data structures that you can use to specify the various kinds of transform objects.

Rotate Transform Data Structure

You can use a rotate transform data structure to specify a rotate transform (for example, when calling the `Q3RotateTransform_NewData` function). The rotate transform data structure is defined by the `TQ3RotateTransformData` data type.

```
typedef struct TQ3RotateTransformData {
    TQ3Axis                             axis;
    float                               radians;
} TQ3RotateTransformData;
```

Field descriptions

`axis`	The axis of rotation. You can use the constants `kQ3AxisX`, `kQ3AxisY`, and `kQ3AxisZ` to specify an axis.
`radians`	The number of radians to rotate around the axis of rotation.

Rotate-About-Point Transform Data Structure

You can use a **rotate-about-point transform data structure** to specify a rotate transform about an axis at an arbitrary point in space (for example, when calling the `Q3RotateAboutPointTransform_NewData` function). The rotate-about-point transform data structure is defined by the `TQ3RotateAboutPointTransformData` data type.

```
typedef struct TQ3RotateAboutPointTransformData {
    TQ3Axis                         axis;
    float                           radians;
    TQ3Point3D                      about;
} TQ3RotateAboutPointTransformData;
```

Field descriptions

`axis`	The axis of rotation. You can use the constants `kQ3AxisX`, `kQ3AxisY`, and `kQ3AxisZ` to specify an axis.
`radians`	The number of radians to rotate around the axis of rotation.
`about`	The point at which the rotation is to occur.

Rotate-About-Axis Data Structure

You can use an **rotate-about-axis transform data structure** to specify a rotate transform about an arbitrary axis in space at an arbitrary point in space. The rotate-about-axis transform data structure is defined by the `TQ3RotateAboutAxisTransformData` data type.

```
typedef struct TQ3RotateAboutAxisTransformData {
    TQ3Point3D                      origin;
    TQ3Vector3D                     orientation;
    float                           radians;
} TQ3RotateAboutAxisTransformData;
```

Field descriptions

`origin`	The origin of the axis of rotation.
`orientation`	The orientation of the axis of rotation. This vector must be normalized or the results will be unpredictable.
`radians`	The number of radians to rotate around the axis of rotation.

Transform Objects Routines

This section describes the routines you can use to manage transforms.

Managing Transforms

QuickDraw 3D provides routines that you can use to manage transforms.

Q3Transform_GetType

You can use the `Q3Transform_GetType` function to get the type of a transform object.

```
TQ3ObjectType Q3Transform_GetType (TQ3TransformObject transform);
```

`transform` A transform.

DESCRIPTION

The `Q3Transform_GetType` function returns, as its function result, the type of the transform object specified by the `transform` parameter. The types of transform objects currently supported by QuickDraw 3D are defined by these constants:

```
kQ3TransformTypeMatrix
kQ3TransformTypeQuaternion
kQ3TransformTypeRotate
kQ3TransformTypeRotateAboutAxis
kQ3TransformTypeRotateAboutPoint
kQ3TransformTypeScale
kQ3TransformTypeTranslate
```

If the specified transform object is invalid or is not one of these types, `Q3Transform_GetType` returns the value `kQ3ObjectTypeInvalid`.

Q3Transform_GetMatrix

You can use the `Q3Transform_GetMatrix` function to get the matrix representation of a transform.

```
TQ3Matrix4x4 *Q3Transform_GetMatrix (
                    TQ3TransformObject transform,
                    TQ3Matrix4x4 *matrix);
```

`transform` A transform.

`matrix` On exit, a pointer to the matrix that represents the transform specified in the `transform` parameter.

DESCRIPTION

The `Q3Transform_GetMatrix` function returns, in the `matrix` parameter and as its function result, the matrix that represents the transform specified by the `transform` parameter. The caller is responsible for allocating the memory pointed to by `matrix`.

Q3Transform_Submit

You can use the `Q3Transform_Submit` function to submit a transform.

```
TQ3Status Q3Transform_Submit (
                    TQ3TransformObject transform,
                    TQ3ViewObject view);
```

`transform` A transform.

`view` A view.

DESCRIPTION

The `Q3Transform_Submit` function pushes the transform specified by the `transform` parameter onto the view transform stack of the specified view. `Q3Transform_Submit` returns `kQ3Success` if the operation succeeds and `kQ3Failure` otherwise.

SPECIAL CONSIDERATIONS

You should call this function only in a submitting loop.

Creating and Manipulating Matrix Transforms

QuickDraw 3D provides routines that you can use to create and manipulate matrix transforms.

Q3MatrixTransform_New

You can use the `Q3MatrixTransform_New` function to create a new matrix transform.

```
TQ3TransformObject Q3MatrixTransform_New (
                    const TQ3Matrix4x4 *matrix);
```

`matrix` On entry, a pointer to a 4-by-4 matrix that defines the desired new transform.

DESCRIPTION

The `Q3MatrixTransform_New` function returns, as its function result, a reference to a new transform object of type `kQ3TransformTypeMatrix` using the data passed in the `matrix` parameter. The data you pass in the `matrix` parameter is copied into internal QuickDraw 3D data structures. If QuickDraw 3D cannot allocate memory for those structures, `Q3MatrixTransform_New` returns the value `NULL`.

It is your responsibility to ensure that the matrix specified by the `matrix` parameter is affine and invertible. QuickDraw 3D does not check for these qualities.

Q3MatrixTransform_Submit

You can use the `Q3MatrixTransform_Submit` function to submit a matrix transform without creating an object or allocating memory.

```
TQ3Status Q3MatrixTransform_Submit (
                    const TQ3Matrix4x4 *matrix,
                    TQ3ViewObject view);
```

`matrix` A pointer to a 4-by-4 matrix.

`view` A view.

DESCRIPTION

The `Q3MatrixTransform_Submit` function pushes the matrix transform specified by the `matrix` parameter on the view transform stack of the view specified by the `view` parameter. The function returns `kQ3Success` if the operation succeeds and `kQ3Failure` otherwise.

SPECIAL CONSIDERATIONS

You should call this function only in a submitting loop.

Q3MatrixTransform_Get

You can use the `Q3MatrixTransform_Get` function to query the private data stored in a matrix transform.

```
TQ3Status Q3MatrixTransform_Get (
                    TQ3TransformObject transform,
                    TQ3Matrix4x4 *matrix);
```

`transform` A transform.

`matrix` On exit, a pointer to the matrix associated with the transform specified in the `transform` parameter.

DESCRIPTION

The `Q3MatrixTransform_Get` function returns, in the `matrix` parameter, information about the matrix transform specified by the `transform` parameter. You should use `Q3MatrixTransform_Get` only with transforms of type `kQ3TransformTypeMatrix`.

Q3MatrixTransform_Set

You can use the `Q3MatrixTransform_Set` function to set new private data for a matrix transform.

```
TQ3Status Q3MatrixTransform_Set (
                    TQ3TransformObject transform,
                    const TQ3Matrix4x4 *matrix);
```

`transform` A transform.

`matrix` A pointer to the new matrix to be associated with the transform specified in the `transform` parameter.

DESCRIPTION

The `Q3MatrixTransform_Set` function sets the matrix transform specified by the `transform` parameter to the matrix passed in the `matrix` parameter. You should use `Q3MatrixTransform_Set` only with transforms of type `kQ3TransformTypeMatrix`.

Creating and Manipulating Rotate Transforms

QuickDraw 3D provides routines that you can use to create and manipulate rotate transforms. A rotate transform rotates an object about the *x*, *y*, or *z* axis by a specified number of radians. You can use macros to convert radians to degrees if you prefer to work with degrees instead of radians. See the chapter "Mathematical Utilities" for more information.

Q3RotateTransform_New

You can use the `Q3RotateTransform_New` function to create a new rotate transform.

```
TQ3TransformObject Q3RotateTransform_New (
                    const TQ3RotateTransformData *data);
```

`data` A pointer to a rotate transform data structure.

DESCRIPTION

The `Q3RotateTransform_New` function returns, as its function result, a reference to a new transform object of type `kQ3TransformTypeRotate` using the data passed in the `data` parameter. The data you pass is copied into internal QuickDraw 3D data structures. If QuickDraw 3D cannot allocate memory for those structures, `Q3RotateTransform_New` returns the value `NULL`.

Q3RotateTransform_Submit

You can use the `Q3RotateTransform_Submit` function to submit a rotate transform without creating an object or allocating memory.

```
TQ3Status Q3RotateTransform_Submit (
                    const TQ3RotateTransformData *data,
                    TQ3ViewObject view);
```

`data` A pointer to a rotate transform data structure.

`view` A view.

DESCRIPTION

The `Q3RotateTransform_Submit` function pushes the rotate transform specified by the `data` parameter onto the view transform stack of the view specified by the `view` parameter. The function returns `kQ3Success` if the operation succeeds and `kQ3Failure` otherwise.

SPECIAL CONSIDERATIONS

You should call this function only in a submitting loop.

Q3RotateTransform_GetData

You can use the `Q3RotateTransform_GetData` function to query the private data stored in a rotate transform.

```
TQ3Status Q3RotateTransform_GetData (
                    TQ3TransformObject transform,
                    TQ3RotateTransformData *data);
```

`transform` A rotate transform.

`data` A pointer to a rotate transform data structure.

DESCRIPTION

The `Q3RotateTransform_GetData` function returns, in the `data` parameter, information about the rotate transform specified by the `transform` parameter. You should use `Q3RotateTransform_GetData` only with transforms of type `kQ3TransformTypeRotate`.

Q3RotateTransform_SetData

You can use the `Q3RotateTransform_SetData` function to set new private data for a rotate transform.

```
TQ3Status Q3RotateTransform_SetData (
                    TQ3TransformObject transform,
                    const TQ3RotateTransformData *data);
```

`transform` A rotate transform.

`data` A pointer to a rotate transform data structure.

DESCRIPTION

The `Q3RotateTransform_SetData` function sets the rotate transform specified by the `transform` parameter to the data passed in the `data` parameter. You should use `Q3RotateTransform_SetData` only with transforms of type `kQ3TransformTypeRotate`.

Q3RotateTransform_GetAxis

You can use the `Q3RotateTransform_GetAxis` function to get the axis of a rotate transform.

```
TQ3Status Q3RotateTransform_GetAxis (
                    TQ3TransformObject transform,
                    TQ3Axis *axis);
```

`transform` A rotate transform.

`axis` On exit, the axis of the specified rotate transform.

DESCRIPTION

The `Q3RotateTransform_GetAxis` function returns, in the `axis` parameter, the current axis of rotation of the rotate transform specified by the `transform` parameter.

Q3RotateTransform_SetAxis

You can use the `Q3RotateTransform_SetAxis` function to set the axis of a rotate transform.

```
TQ3Status Q3RotateTransform_SetAxis (
                    TQ3TransformObject transform,
                    TQ3Axis axis);
```

`transform` A rotate transform.

`axis` The desired axis of the specified rotate transform.

DESCRIPTION

The `Q3RotateTransform_SetAxis` function sets the axis of rotation for the rotate transform specified by the `transform` parameter to the value passed in the `axis` parameter.

Q3RotateTransform_GetAngle

You can use the `Q3RotateTransform_GetAngle` function to get the angle of a rotate transform.

```
TQ3Status Q3RotateTransform_GetAngle (
                    TQ3TransformObject transform,
                    float *radians);
```

`transform` A rotate transform.

`radians` On exit, the angle, in radians, of the specified rotate transform.

DESCRIPTION

The `Q3RotateTransform_GetAngle` function returns, in the `radians` parameter, the current angle of rotation (in radians) of the rotate transform specified by the `transform` parameter.

Q3RotateTransform_SetAngle

You can use the `Q3RotateTransform_SetAngle` function to set the angle of a rotate transform.

```
TQ3Status Q3RotateTransform_SetAngle (
                    TQ3TransformObject transform,
                    float radians);
```

`transform` A rotate transform.

`radians` The desired angle, in radians, of the specified rotate transform.

DESCRIPTION

The `Q3RotateTransform_SetAngle` function sets the angle of rotation for the rotate transform specified by the `transform` parameter to the value passed in the `radians` parameter.

Creating and Manipulating Rotate-About-Point Transforms

QuickDraw 3D provides routines that you can use to create and manipulate rotate transforms about a point. A rotate-about-point transform rotates an object about the *x*, *y*, or *z* axis by a specified number of radians at an arbitrary point in space. You can use macros to convert radians to degrees if you prefer to work with degrees instead of radians. See the chapter "Mathematical Utilities" for more information.

Q3RotateAboutPointTransform_New

You can use the `Q3RotateAboutPointTransform_New` function to create a new rotate-about-point transform.

```
TQ3TransformObject Q3RotateAboutPointTransform_New (
                     const TQ3RotateAboutPointTransformData *data);
```

`data` A pointer to a `TQ3RotateAboutPointTransformData` structure.

DESCRIPTION

The `Q3RotateAboutPointTransform_New` function returns, as its function result, a reference to a new transform object of type `kQ3TransformTypeRotateAboutPoint` using the data passed in the `data` parameter. The data you pass is copied into internal QuickDraw 3D data structures. If QuickDraw 3D cannot allocate memory for those structures, `Q3RotateAboutPointTransform_New` returns the value `NULL`.

Q3RotateAboutPointTransform_Submit

You can use the `Q3RotateAboutPointTransform_Submit` function to submit a rotate-about-point transform without creating an object or allocating memory.

```
TQ3Status Q3RotateAboutPointTransform_Submit (
                    const TQ3RotateAboutPointTransformData *data,
                    TQ3ViewObject view);
```

`data` A pointer to a `TQ3RotateAboutPointTransformData` structure.

`view` A view.

DESCRIPTION

The `Q3RotateAboutPointTransform_Submit` function pushes the rotate-about-point transform specified by the `data` parameter onto the view transform stack of the view specified by the `view` parameter. The function returns `kQ3Success` if the operation succeeds and `kQ3Failure` otherwise.

SPECIAL CONSIDERATIONS

You should call this function only in a submitting loop.

Q3RotateAboutPointTransform_GetData

You can use the `Q3RotateAboutPointTransform_GetData` function to query the private data stored in a rotate-about-point transform.

```
TQ3Status Q3RotateAboutPointTransform_GetData (
                    TQ3TransformObject transform,
                    TQ3RotateAboutPointTransformData *data);
```

`transform` A transform.

`data` A pointer to a rotate-about-point data structure.

DESCRIPTION

The `Q3RotateAboutPointTransform_GetData` function returns, in the `data` parameter, information about the rotate-about-point transform specified by the `transform` parameter. You should use `Q3RotateAboutPointTransform_GetData` only with transforms of type `kQ3TransformTypeRotateAboutPoint`.

Q3RotateAboutPointTransform_SetData

You can use the `Q3RotateAboutPointTransform_SetData` function to set new private data for a rotate-about-point transform.

```
TQ3Status Q3RotateAboutPointTransform_SetData (
                    TQ3TransformObject transform,
                    const TQ3RotateAboutPointTransformData *data);
```

`transform` A transform.

`data` A pointer to a rotate-about-point data structure.

DESCRIPTION

The `Q3RotateAboutPointTransform_SetData` function sets the rotate-about-point transform specified by the `transform` parameter to the data passed in the `data` parameter. You should use `Q3RotateAboutPointTransform_SetData` only with transforms of type `kQ3TransformTypeRotateAboutPoint`.

Q3RotateAboutPointTransform_GetAxis

You can use the `Q3RotateAboutPointTransform_GetAxis` function to get the axis of a rotate-about-point transform.

```
TQ3Status Q3RotateAboutPointTransform_GetAxis (
                    TQ3TransformObject transform,
                    TQ3Axis *axis);
```

`transform` A rotate-about-point transform.

`axis` On exit, the axis of the specified rotate-about-point transform.

DESCRIPTION

The `Q3RotateAboutPointTransform_GetAxis` function returns, in the `axis` parameter, the current axis of rotation of the rotate-about-point transform specified by the `transform` parameter.

Q3RotateAboutPointTransform_SetAxis

You can use the `Q3RotateAboutPointTransform_SetAxis` function to set the axis of a rotate-about-point transform.

```
TQ3Status Q3RotateAboutPointTransform_SetAxis (
                    TQ3TransformObject transform,
                    TQ3Axis axis);
```

`transform` A rotate-about-point transform.

`axis` The desired axis of the specified rotate-about-point transform.

DESCRIPTION

The `Q3RotateAboutPointTransform_SetAxis` function sets the axis of rotation for the rotate-about-point transform specified by the `transform` parameter to the value passed in the `axis` parameter.

Q3RotateAboutPointTransform_GetAngle

You can use the `Q3RotateAboutPointTransform_GetAngle` function to get the angle of a rotate-about-point transform.

```
TQ3Status Q3RotateAboutPointTransform_GetAngle (
                    TQ3TransformObject transform,
                    float *radians);
```

`transform`	A rotate-about-point transform.
`radians`	On exit, the angle, in radians, of the specified rotate-about-point transform.

DESCRIPTION

The `Q3RotateAboutPointTransform_GetAngle` function returns, in the `radians` parameter, the current angle of rotation (in radians) of the rotate-about-point transform specified by the `transform` parameter.

Q3RotateAboutPointTransform_SetAngle

You can use the `Q3RotateAboutPointTransform_SetAngle` function to set the angle of a rotate-about-point transform.

```
TQ3Status Q3RotateAboutPointTransform_SetAngle (
                    TQ3TransformObject transform,
                    float radians);
```

`transform`	A rotate-about-point transform.
`radians`	The desired angle, in radians, of the specified rotate-about-point transform.

DESCRIPTION

The `Q3RotateAboutPointTransform_SetAngle` function sets the angle of rotation for the rotate-about-point transform specified by the `transform` parameter to the value passed in the `radians` parameter.

Q3RotateAboutPointTransform_GetAboutPoint

You can use the `Q3RotateAboutPointTransform_GetAboutPoint` function to get the point of rotation of a rotate-about-point transform.

```
TQ3Status Q3RotateAboutPointTransform_GetAboutPoint (
                    TQ3TransformObject transform,
                    TQ3Point3D *about);
```

`transform` A rotate-about-point transform.

`about` On exit, the point of rotation of the specified rotate-about-point transform.

DESCRIPTION

The `Q3RotateAboutPointTransform_GetAboutPoint` function returns, in the `about` parameter, the current point of rotation of the rotate-about-point transform specified by the `transform` parameter.

Q3RotateAboutPointTransform_SetAboutPoint

You can use the `Q3RotateAboutPointTransform_SetAboutPoint` function to set the point of rotation of a rotate-about-point transform.

```
TQ3Status Q3RotateAboutPointTransform_SetAboutPoint (
                    TQ3TransformObject transform,
                    const TQ3Point3D *about);
```

`transform` A rotate-about-point transform.

`about` The desired point of rotation of the specified rotate-about-point transform.

DESCRIPTION

The `Q3RotateAboutPointTransform_SetAboutPoint` function sets the point of rotation for the rotate-about-point transform specified by the `transform` parameter to the value passed in the `about` parameter.

Creating and Manipulating Rotate-About-Axis Transforms

QuickDraw 3D provides routines that you can use to create and manipulate rotate-about-axis transforms. An rotate-about-axis transform rotates an object

about an arbitrary axis in space by a specified number of radians. You can use macros to convert radians to degrees if you prefer to work with degrees instead of radians. See the chapter "Mathematical Utilities" for more information.

Q3RotateAboutAxisTransform_New

You can use the `Q3RotateAboutAxisTransform_New` function to create a new rotate-about-axis transform.

```
TQ3TransformObject Q3RotateAboutAxisTransform_New (
                    const TQ3RotateAboutAxisTransformData *data);
```

`data` A pointer to a `TQ3RotateAboutAxisTransformData` structure.

DESCRIPTION

The `Q3RotateAboutAxisTransform_New` function returns, as its function result, a reference to a new transform object of type `kQ3TransformTypeRotateAboutAxis` using the data passed in the `data` parameter. The data you pass is copied into internal QuickDraw 3D data structures. If QuickDraw 3D cannot allocate memory for those structures, `Q3RotateAboutAxisTransform_New` returns the value `NULL`.

Q3RotateAboutAxisTransform_Submit

You can use the `Q3RotateAboutAxisTransform_Submit` function to submit a rotate-about-axis transform without creating an object or allocating memory.

```
TQ3Status Q3RotateAboutAxisTransform_Submit (
                    const TQ3RotateAboutAxisTransformData *data,
                    TQ3ViewObject view);
```

`data` A pointer to a `TQ3RotateAboutAxisTransformData` structure.

`view` A view.

DESCRIPTION

The `Q3RotateAboutAxisTransform_Submit` function pushes the rotate-about-axis transform specified by the `data` parameter onto the view transform stack of the view specified by the `view` parameter. The function returns `kQ3Success` if the operation succeeds and `kQ3Failure` otherwise.

SPECIAL CONSIDERATIONS

You should call this function only in a submitting loop.

Q3RotateAboutAxisTransform_GetData

You can use the `Q3RotateAboutAxisTransform_GetData` function to query the private data stored in a rotate-about-axis transform.

```
TQ3Status Q3RotateAboutAxisTransform_GetData (
                    TQ3TransformObject transform,
                    TQ3RotateAboutAxisTransformData *data);
```

`transform` A rotate-about-axis transform.

`data` A pointer to a rotate-about-axis data structure.

DESCRIPTION

The `Q3RotateAboutAxisTransform_GetData` function returns, in the `data` parameter, information about the rotate-about-axis transform specified by the `transform` parameter. You should use `Q3RotateAboutAxisTransform_GetData` only with transforms of type `kQ3TransformTypeRotateAboutAxis`.

Q3RotateAboutAxisTransform_SetData

You can use the `Q3RotateAboutAxisTransform_SetData` function to set new private data for a rotate-about-axis transform.

```
TQ3Status Q3RotateAboutAxisTransform_SetData (
                    TQ3TransformObject transform,
                    const TQ3RotateAboutAxisTransformData *data);
```

`transform` A rotate-about-axis transform.

`data` A pointer to a rotate-about-axis data structure.

DESCRIPTION

The `Q3RotateAboutAxisTransform_SetData` function sets the rotate-about-axis transform specified by the `transform` parameter to the data passed in the `data` parameter. You should use `Q3RotateAboutAxisTransform_SetData` only with transforms of type `kQ3TransformTypeRotateAboutAxis`.

Q3RotateAboutAxisTransform_GetOrigin

You can use the `Q3RotateAboutAxisTransform_GetOrigin` function to get the origin of the axis of rotation of a rotate-about-axis transform.

```
TQ3Status Q3RotateAboutAxisTransform_GetOrigin (
                    TQ3TransformObject transform,
                    TQ3Point3D *origin);
```

`transform` A rotate-about-axis transform.

`origin` On exit, the origin of the axis of rotation of the specified rotate-about-axis transform.

DESCRIPTION

The `Q3RotateAboutAxisTransform_GetOrigin` function returns, in the `origin` parameter, the current origin of the axis of rotation of the rotate-about-axis transform specified by the `transform` parameter.

Q3RotateAboutAxisTransform_SetOrigin

You can use the `Q3RotateAboutAxisTransform_SetOrigin` function to set the origin of the axis of rotation of a rotate-about-axis transform.

```
TQ3Status Q3RotateAboutAxisTransform_SetOrigin (
                    TQ3TransformObject transform,
                    const TQ3Point3D *origin);
```

`transform` A rotate-about-axis transform.

`origin` The desired origin of the axis of rotation of the specified rotate-about-axis transform.

DESCRIPTION

The `Q3RotateAboutAxisTransform_SetOrigin` function sets the origin of the axis of rotation for the rotate-about-axis transform specified by the `transform` parameter to the value passed in the `origin` parameter.

Q3RotateAboutAxisTransform_GetOrientation

You can use the `Q3RotateAboutAxisTransform_GetOrientation` function to get the orientation of the axis of rotation of a rotate-about-axis transform.

```
TQ3Status Q3RotateAboutAxisTransform_GetOrientation (
                    TQ3TransformObject transform,
                    TQ3Vector3D *axis);
```

`transform` A rotate-about-axis transform.

`axis` On exit, the orientation of the axis of the specified rotate-about-axis transform. This vector is normalized.

DESCRIPTION

The `Q3RotateAboutAxisTransform_GetOrientation` function returns, in the `axis` parameter, the current orientation of the axis of rotation of the rotate-about-axis transform specified by the `transform` parameter.

Q3RotateAboutAxisTransform_SetOrientation

You can use the `Q3RotateAboutAxisTransform_SetOrientation` function to set the orientation of the axis of rotation of a rotate-about-axis transform.

```
TQ3Status Q3RotateAboutAxisTransform_SetOrientation (
                    TQ3TransformObject transform,
                    const TQ3Vector3D *axis);
```

`transform` A rotate-about-axis transform.

`axis` The desired orientation of the axis of the specified rotate-about-axis transform. This vector must be normalized.

DESCRIPTION

The `Q3RotateAboutAxisTransform_SetOrientation` function sets orientation of the axis of rotation for the rotate-about-axis transform specified by the `transform` parameter to the value passed in the `axis` parameter.

Q3RotateAboutAxisTransform_GetAngle

You can use the `Q3RotateAboutAxisTransform_GetAngle` function to get the angle of a rotate-about-axis transform.

```
TQ3Status Q3RotateAboutAxisTransform_GetAngle (
                    TQ3TransformObject transform,
                    float *radians);
```

`transform` A rotate-about-axis transform.

`radians` On exit, the angle, in radians, of the specified rotate-about-axis transform.

DESCRIPTION

The `Q3RotateAboutAxisTransform_GetAngle` function returns, in the `radians` parameter, the current angle of rotation (in radians) of the rotate-about-axis transform specified by the `transform` parameter.

Q3RotateAboutAxisTransform_SetAngle

You can use the `Q3RotateAboutAxisTransform_SetAngle` function to set the angle of a rotate-about-axis transform.

```
TQ3Status Q3RotateAboutAxisTransform_SetAngle (
                    TQ3TransformObject transform,
                    float radians);
```

`transform` A rotate-about-axis transform.

`radians` The desired angle, in radians, of the specified rotate-about-axis transform.

DESCRIPTION

The `Q3RotateAboutAxisTransform_SetAngle` function sets the angle of rotation for the rotate-about-axis transform specified by the `transform` parameter to the value passed in the `radians` parameter.

Creating and Manipulating Scale Transforms

QuickDraw 3D provides routines that you can use to create and manipulate scale transforms. A scale transform scales an object along the *x*, *y*, and *z* axes by specified values. You are responsible for ensuring that an object is at the correct location and in the proper orientation for the scaling to have the desired effect.

IMPORTANT

A scale factor can be negative. You should, however, exercise caution when using negative scale factors. In addition, when two or three of the scale factors are 0, nothing is drawn. ▲

Q3ScaleTransform_New

You can use the `Q3ScaleTransform_New` function to create a new scale transform.

```
TQ3TransformObject Q3ScaleTransform_New (
                    const TQ3Vector3D *scale);
```

`scale` A vector whose three fields specify the desired scaling along each coordinate axis.

DESCRIPTION

The `Q3ScaleTransform_New` function returns, as its function result, a reference to a new transform object of type `kQ3TransformTypeScale` using the data passed in the `scale` parameter. The scale transform scales an object by the values in `scale->x`, `scale->y`, and `scale->z`, respectively. The data you pass in the `scale` parameter is copied into internal QuickDraw 3D data structures. If QuickDraw 3D cannot allocate memory for those structures, `Q3ScaleTransform_New` returns the value `NULL`.

Q3ScaleTransform_Submit

You can use the `Q3ScaleTransform_Submit` function to submit a scale transform without creating an object or allocating memory.

```
TQ3Status Q3ScaleTransform_Submit (
                    TQ3Vector3D *scale,
                    TQ3ViewObject view);
```

`scale` A vector whose three fields specify the desired scaling along each coordinate axis.

`view` A view.

DESCRIPTION

The `Q3ScaleTransform_Submit` function pushes the scale transform specified by the `scale` parameter on the view transform stack of the view specified by the `view` parameter. The function returns `kQ3Success` if the operation succeeds and `kQ3Failure` otherwise.

SPECIAL CONSIDERATIONS

You should call this function only in a submitting loop.

Q3ScaleTransform_Get

You can use the `Q3ScaleTransform_Get` function to query the private data stored in a scale transform.

```
TQ3Status Q3ScaleTransform_Get (
                    TQ3TransformObject transform,
                    TQ3Vector3D *scale);
```

`transform` A transform.

`scale` A vector whose three fields specify the scaling along each coordinate axis.

DESCRIPTION

The `Q3ScaleTransform_Get` function returns, in the `scale` parameter, information about the scale transform specified by the `transform` parameter. You should use `Q3ScaleTransform_Get` only with transforms of type `kQ3TransformTypeScale`.

Q3ScaleTransform_Set

You can use the `Q3ScaleTransform_Set` function to set new private data for a scale transform.

```
TQ3Status Q3ScaleTransform_Set (
                    TQ3TransformObject transform,
                    const TQ3Vector3D *scale);
```

`transform` A transform.

`scale` A vector whose three fields specify the desired scaling along each coordinate axis.

DESCRIPTION

The `Q3ScaleTransform_Set` function sets the scale transform specified by the `transform` parameter to the data passed in the `scale` parameter. You should use `Q3ScaleTransform_Set` only with transforms of type `kQ3TransformTypeScale`.

Creating and Manipulating Translate Transforms

QuickDraw 3D provides routines that you can use to create and manipulate translate transforms. A translate transform translates an object along the *x*, *y*, and *z* axes by specified values.

Q3TranslateTransform_New

You can use the `Q3TranslateTransform_New` function to create a new translate transform.

```
TQ3TransformObject Q3TranslateTransform_New (
                    const TQ3Vector3D *translate);
```

`translate` A vector whose three fields specify the desired translation along each coordinate axis.

DESCRIPTION

The `Q3TranslateTransform_New` function returns, as its function result, a reference to a new transform object of type `kQ3TransformTypeTranslate` using the data passed in the `translate` parameter. The transform translates an object by the values in `translate->x`, `translate->y`, and `translate->z`, respectively. The data you pass in the `translate` parameter is copied into internal QuickDraw 3D data structures. If QuickDraw 3D cannot allocate memory for those structures, `Q3TranslateTransform_New` returns the value `NULL`.

Q3TranslateTransform_Submit

You can use the `Q3TranslateTransform_Submit` function to submit a translate transform without creating an object or allocating memory.

```
TQ3Status Q3TranslateTransform_Submit (
                    const TQ3Vector3D *translate,
                    TQ3ViewObject view);
```

`translate` A vector whose three fields specify the desired translation along each coordinate axis.

`view` A view.

DESCRIPTION

The `Q3TranslateTransform_Submit` function pushes the translate transform specified by the `translate` parameter on the view transform stack of the view specified by the `view` parameter. The function returns `kQ3Success` if the operation succeeds and `kQ3Failure` otherwise.

SPECIAL CONSIDERATIONS

You should call this function only in a submitting loop.

Q3TranslateTransform_Get

You can use the `Q3TranslateTransform_Get` function to query the private data stored in a translate transform.

```
TQ3Status Q3TranslateTransform_Get (
                    TQ3TransformObject transform,
                    TQ3Vector3D *translate);
```

`transform` A transform.

`translate` On entry, a pointer to a vector. On exit, a pointer to a vector whose three fields specify the current translation along each coordinate axis.

DESCRIPTION

The `Q3TranslateTransform_Get` function returns, in the `translate` parameter, information about the translate transform specified by the `transform` parameter. You should use `Q3TranslateTransform_Get` only with transforms of type `kQ3TransformTypeTranslate`.

Q3TranslateTransform_Set

You can use the `Q3TranslateTransform_Set` function to set new private data for a translate transform.

```
TQ3Status Q3TranslateTransform_Set (
                    TQ3TransformObject transform,
                    const TQ3Vector3D *translate);
```

`transform` A transform.

`translate` A vector whose three fields specify the desired translation along each coordinate axis.

DESCRIPTION

The `Q3TranslateTransform_Set` function sets the translate transform specified by the `transform` parameter to the data passed in the `translate` parameter. You should use `Q3TranslateTransform_Set` only with transforms of type `kQ3TransformTypeTranslate`.

Creating and Manipulating Quaternion Transforms

QuickDraw 3D provides routines that you can use to create and manipulate quaternion transforms. A quaternion transform rotates and twists an object according to the mathematical properties of quaternions.

Q3QuaternionTransform_New

You can use the `Q3QuaternionTransform_New` function to create a new quaternion transform.

```
TQ3TransformObject Q3QuaternionTransform_New (TQ3Quaternion *quaternion);
```

`quaternion` A quaternion.

DESCRIPTION

The `Q3QuaternionTransform_New` function returns, as its function result, a reference to a new transform object of type `kQ3TransformTypeQuaternion` using the data passed in the `quaternion` parameter. The data you pass in the `quaternion` parameter is copied into internal QuickDraw 3D data structures. If QuickDraw 3D cannot allocate memory for those structures, `Q3QuaternionTransform_New` returns the value `NULL`.

Q3QuaternionTransform_Submit

You can use the `Q3QuaternionTransform_Submit` function to submit a quaternion transform without creating an object or allocating memory.

```
TQ3Status Q3QuaternionTransform_Submit (
                    TQ3Quaternion *quaternion,
                    TQ3ViewObject view);
```

`quaternion` A quaternion.

`view` A view.

DESCRIPTION

The `Q3QuaternionTransform_Submit` function pushes the quaternion transform specified by the `quaternion` parameter on the view transform stack of the view specified by the `view` parameter. The function returns `kQ3Success` if the operation succeeds and `kQ3Failure` otherwise.

SPECIAL CONSIDERATIONS

You should call this function only in a submitting loop.

Q3QuaternionTransform_Get

You can use the `Q3QuaternionTransform_Get` function to query the private data stored in a quaternion transform.

```
TQ3Status Q3QuaternionTransform_Get (
                    TQ3TransformObject transform,
                    TQ3Quaternion *quaternion);
```

`transform` A transform.

`quaternion` A quaternion.

DESCRIPTION

The `Q3QuaternionTransform_Get` function returns, in the `quaternion` parameter, information about the quaternion transform specified by the `transform` parameter. You should use `Q3QuaternionTransform_Get` only with transforms of type `kQ3TransformTypeQuaternion`.

Q3QuaternionTransform_Set

You can use the `Q3QuaternionTransform_Set` function to set new private data for a quaternion transform.

```
TQ3Status Q3QuaternionTransform_Set (
                    TQ3TransformObject transform,
                    TQ3Quaternion *quaternion);
```

`transform` A transform.

`quaternion` A quaternion.

DESCRIPTION

The `Q3QuaternionTransform_Set` function sets the quaternion transform specified by the `transform` parameter to the data passed in the `quaternion` parameter. You should use `Q3QuaternionTransform_Set` only with transforms of type `kQ3TransformTypeQuaternion`.

Creating and Submitting the Reset Transform

QuickDraw 3D provides routines that you can use to implement the identity transformation described in "The Reset Transform," beginning on page 598.

Q3ResetTransform_New

You can use the `Q3ResetTransform_New` function to create a reset transform object.

```
TQ3TransformObject Q3ResetTransform_New (void);
```

DESCRIPTION

The `Q3ResetTransform_New` function returns a reset transform object that resets the current transform object to the identity transformation.

Q3ResetTransform_Submit

You can use the `Q3ResetTransform_Submit` function to rest the transformation of a view to identity.

```
TQ3Status Q3ResetTransform_Submit (TQ3ViewObject view);
```

`view` A view.

DESCRIPTION

The `Q3ResetTransform_Submit` function resets the current transformation of the view designated by `view` to the identity transformation.

Transform Errors, Warnings, and Notices

Transform operations may return the following errors, warnings, and notices. A list of general QuickDraw 3D errors is given in “QuickDraw 3D Errors, Warnings, and Notices” (page 87).

```
kQ3ErrorScaleOfZero
kQ3WarningScaleEntriesAllZero
kQ3WarningScaleContainsNegativeEntries
kQ3NoticeScaleContainsZeroEntries
```

CHAPTER 8

Light Objects

This chapter describes light objects (or lights) and the functions you can use to manipulate them. You use lights to provide illumination on the objects in a model. A group of lights is associated with every view, along with camera information and other settings that affect the rendering of a model.

To use this chapter, you should already be familiar with the QuickDraw 3D class hierarchy, described in the chapter "QuickDraw 3D Objects" earlier in this book. For information about grouping lights into a light group, see the chapter "Group Objects." For information about associating a light group with a view, see the chapter "View Objects." You do not, however, need to know how to create light groups or attach them to views to read this chapter.

For the lights associated with a view to have any effect, there must also be an illumination shader associated with the view. See the chapter "Shader Objects" for information on creating illumination shaders and attaching them to views.

This chapter begins by describing light objects and their features. Then it shows how to create and manipulate lights. The section "Light Objects Reference," beginning on page 637 provides a complete description of light objects and the routines you can use to create and manipulate them.

About Light Objects

A **light object** (or, more briefly, a **light**) is a type of QuickDraw 3D object that you can use to provide illumination to the surfaces in a scene. A light is of type `TQ3LightObject`.

In general, the illumination of a surface in a scene is affected by multiple light sources. As a result, a view is associated with a **light group,** which is simply a group of lights. To illuminate the objects in the scene, you need to create a light

group and attach it to a view (for example, by calling `Q3LightGroup_New` and `Q3View_SetLightGroup`).

Note
If you do not attach a group of lights to a view, the results are renderer-specific. ◆

QuickDraw 3D supports multiple light sources and multiple types of lights in a given scene. QuickDraw 3D defines four types of lights:

- ambient lights
- directional lights
- point lights
- spot lights

All four types of lights share some basic properties, which are maintained in a **light data structure,** defined by the `TQ3LightData` data structure.

```
typedef struct TQ3LightData {
    TQ3Boolean                      isOn;
    float                           brightness;
    TQ3ColorRGB                     color;
} TQ3LightData;
```

These fields specify the brightness (that is, the intensity) and color of the light and the current state (active or inactive) of the light. You can turn a light on and off by toggling the `isOn` field of a light data structure.

As you will see, an ambient light is completely described by a light data structure. All other types of lights contain additional information, such as the location and direction of the light source. Those kinds of lights are defined by data structures that include a light data structure.

Ambient Light

Ambient light is an amount of light of a specific color that is added to the illumination of all surfaces in a scene. QuickDraw 3D supports at most *one* active source of ambient light per view, which is therefore called the ambient light object (or the ambient light). An ambient light has no location and cannot therefore cast shadows or become attenuated by distance of the light source from a surface. In effect, ambient light is light that is applied equally

everywhere in a scene. In the absence of any other light sources, an ambient light illuminates a scene with a flat, uniform light. An ambient light is defined by the `TQ3LightData` data structure.

Directional Lights

A **directional light** is a light source that emits parallel rays of light in a specific direction. You can think of a directional light as a light source that is infinitely far away from the surfaces it is illuminating. For example, for scenes on the surface of the Earth, the sun is effectively a directional light.

Note
Directional lights are therefore sometimes also called *infinite lights.* ◆

A directional light has no location. As a result, you specify the direction of the light as a vector equivalent to the direction of the light. In addition, a directional light cannot suffer attenuation (that is, a loss of intensity over distance). It can, however, cast shadows.

Point Lights

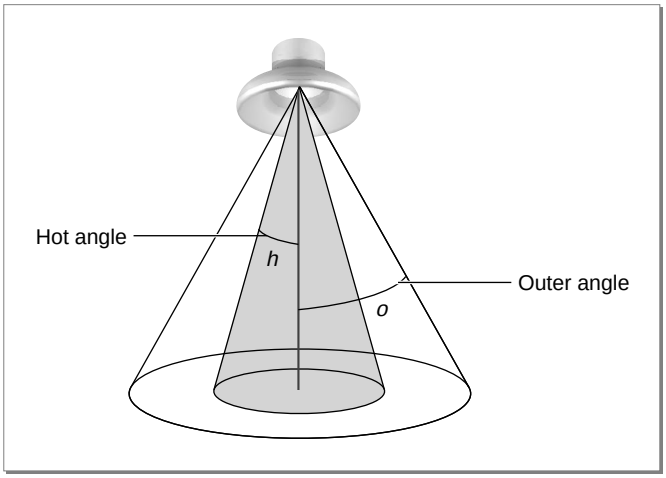
A **point light** is a light source that emits rays of light in all directions from a specific location. The illumination that a point light contributes to a surface depends on the basic properties of the light source (its intensity and color) together with the orientation of the surface and its distance from the light source.

A point light can suffer attenuation, in which case objects closer to the light source receive more illumination than objects farther away. QuickDraw 3D allows you to specify one of several attenuation values that determine the precise amount by which the intensity of a point light decays over distance. For example, you can use the constant `kQ3AttenuationTypeInverseDistance` to have the intensity of a point light be inversely proportional to the distance between the illuminated surface and the light source. See “Light Attenuation Values” (page 638) for a complete list of the available attenuation values.

Spot Lights

A **spot light** is a light source that emits a circular cone of light in a specific direction from a specific location. Figure 8-1 shows the geometry of a spot light. Every spot light has a hot angle and an outer angle that together define the shape of the cone of light and the amount of attenuation, if any, that occurs from the center of the cone to the outer edge of the cone.

Figure 8-1 A spot light

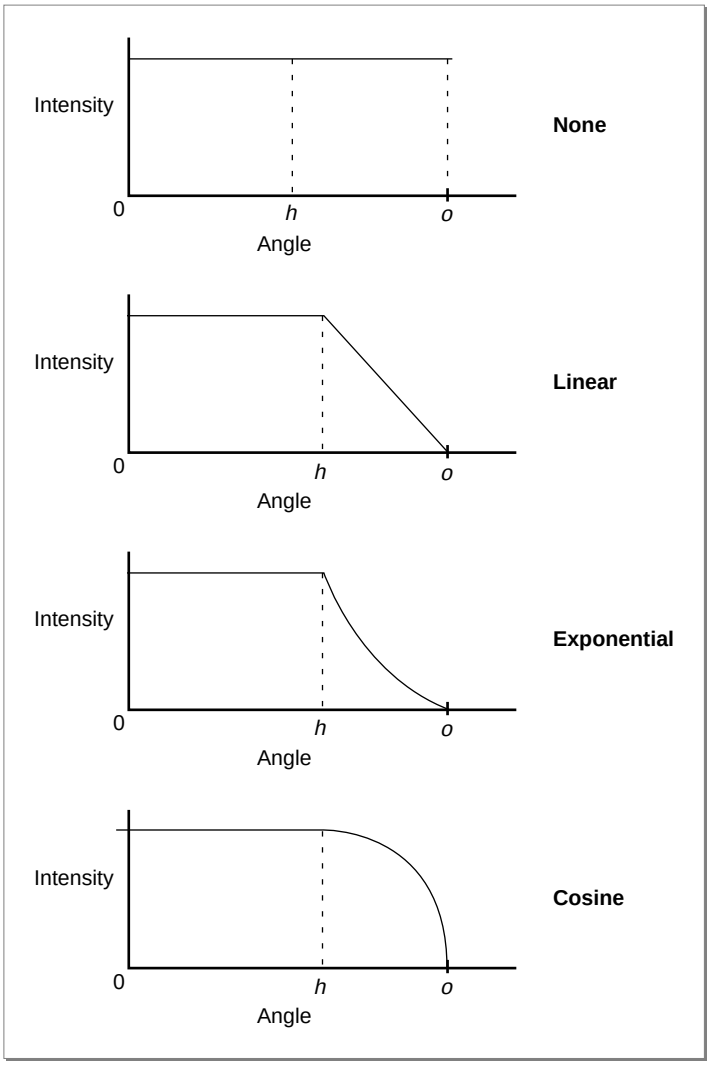


A spot light's **hot angle** is the half-angle (specified in radians) from the center of the cone of light within which the light remains at constant full intensity. In Figure 8-1, *h* is the hot angle. A spot light's **outer angle** is the half angle (specified in radians) from the center of the cone to the edge of the cone. In Figure 8-1, *o* is the outer angle.

The attenuation of the light's intensity from the edge of the hot angle to the edge of the outer angle is determined by the light's **fall-off value.** QuickDraw 3D allows you to specify no fall-off, a linear fall-off, an exponential fall-off, and a fall-off that is proportional to the cosine of the angle. The available fall-off algorithms are illustrated in Figure 8-2.

See "Light Fall-Off Values" (page 638) for a description of the constants you can use to specify a spot light's fall-off value.

Figure 8-2 Fall-off algorithms



Using Light Objects

QuickDraw 3D supplies routines that you can use to create and manipulate light objects. This section describes how to accomplish these tasks.

Creating a Light

You create a light by filling in the fields of the data structure for the type of light you want to create and then by calling a QuickDraw 3D function to create the light. For example, to create a point light, you fill in a data structure of type `TQ3PointLightData` and then call `Q3PointLight_New`, as shown in Listing 8-1.

Listing 8-1 Creating a new point light

```
TQ3LightObject MyNewPointLight (void)
{
    TQ3LightData                    myLightData;
    TQ3PointLightData               myPointLightData;
    TQ3LightObject                  myPointLight;
    TQ3Point3D                      pointLocation = {-20.0, 0.0, 20.0};
    TQ3ColorRGB                     WhiteLight = { 1.0, 1.0, 1.0 };

    /*Set up light data for a point light.*/
    myLightData.isOn = kQ3True;
    myLightData.brightness = 1.0;
    myLightData.color = WhiteLight;
    myPointLightData.lightData = myLightData;
    myPointLightData.castsShadows = kQ3False;
    myPointLightData.attenuation = kQ3AttenuationTypeNone;
    myPointLightData.location = pointLocation;

    /*Create a point light.*/
    myPointLight = Q3PointLight_New(&myPointLightData);
    return (myPointLight);
}
```

As you can see, the `MyNewPointLight` function defined in Listing 8-1 simply fills in the `myPointLight` structure and then calls `Q3PointLight_New`. `MyNewPointLight` returns to its caller either a reference to the new light (if `Q3PointLight_New` succeeds) or the value `NULL` (if `Q3PointLight_New` fails).

Manipulating Lights

For a light to affect a model in a view, you need to insert the light into the light group associated with the view. You call `Q3LightGroup_New` to create a new (empty) light group and `Q3Group_AddObject` to add lights to that group. Then you need to call `Q3View_SetLightGroup` to attach the light group to a view. Finally, you need to create an illumination shader that specifies the kind of illumination model you want applied to objects in the model. For example, to provide Phong illumination on the objects in a model, you can create an illumination shader by calling `Q3PhongIllumination_New`. The illumination shader is not explicitly associated with the view. Instead, you specify the illumination shader by calling `Q3Shader_Submit` in your rendering loop. See the chapter "Shader Objects" for details.

Light Objects Reference

This section describes the constants, data structures, and routines you can use to create and manipulate light objects.

Constants

This section describes the constants that you use to define light attenuation and fall-off values.

Note
Some renderers might not support all the defined attenuation or fall-off values. ◆

Light Attenuation Values

Most types of lights have an attenuation value that determines how quickly, if at all, the intensity of a light changes as a function of the distance of the illuminated object from the light source. You can use these constants to specify an attenuation value:

```
typedef enum TQ3AttenuationType {
    kQ3AttenuationTypeNone,
    kQ3AttenuationTypeInverseDistance,
    kQ3AttenuationTypeInverseDistanceSquared
} TQ3AttenuationType;
```

Constant descriptions

`kQ3AttenuationTypeNone`
: The intensity of the light is not affected by the distance from the illuminated object.

`kQ3AttenuationTypeInverseDistance`
: The intensity of the light is inversely proportional to the distance from the illuminated object.

`kQ3AttenuationTypeInverseDistanceSquared`
: The intensity of the light is inversely proportional to the square of the distance from the illuminated object.

Light Fall-Off Values

Spot lights have a fall-off value that determines the attenuation of the light from the edge of the hot angle to the edge of the outer angle. You can use these constants to specify a fall-off value:

```
typedef enum TQ3FallOffType {
    kQ3FallOffTypeNone,
    kQ3FallOffTypeLinear,
    kQ3FallOffTypeExponential,
    kQ3FallOffTypeCosine
} TQ3FallOffType;
```

Constant descriptions

`kQ3FallOffTypeNone`
: The intensity of the light is not affected by the distance from the center of the cone to the edge of the cone.

`kQ3FallOffTypeLinear`
: The intensity of the light at the edge of the cone falls off linearly from the intensity of the light at the center of the cone.

`kQ3FallOffTypeExponential`
: The intensity of the light at the edge of the cone falls off exponentially from the intensity of the light at the center of the cone.

`kQ3FallOffTypeCosine`
: The intensity of the light at the edge of the cone falls off as the cosine of the outer angle from the intensity of the light at the center of the cone.

Data Structures

This section describes the data structures supplied by QuickDraw 3D for managing lights. The data structures used to manage lights are all public.

Note

The locations and directions of lights are always specified in world coordinates. ◆

Light Data Structure

You use a light data structure to get or set basic information about a light source of any kind. A light data structure is defined by the `TQ3LightData` data type.

```
typedef struct TQ3LightData {
    TQ3Boolean                      isOn;
    float                           brightness;
    TQ3ColorRGB                     color;
} TQ3LightData;
```

Field descriptions

`isOn`	A Boolean value that indicates whether the light source is active (`kQ3True`) or inactive (`kQ3False`).
`brightness`	The brightness or intensity of the light source. The value in this field is a floating-point number in the range 0.0 to 1.0, inclusive. Some renderers may allow you to specify overbright lights (where the value in this field is greater than 1.0) or lights with negative brightness (where the value in this field is less than 0.0); the effects produced by out-of-range brightness values are renderer-specific.
`color`	The color of the light emitted by a light source.

Directional Light Data Structure

You use a **directional light data structure** to get or set information about a directional light source. A directional light data structure is defined by the `TQ3DirectionalLightData` data type.

```
typedef struct TQ3DirectionalLightData {
    TQ3LightData                        lightData;
    TQ3Boolean                          castsShadows;
    TQ3Vector3D                         direction;
} TQ3DirectionalLightData;
```

Field descriptions

`lightData`	A light data structure specifying basic information about the directional light.
`castsShadows`	A Boolean value that indicates whether the directional light casts shadows (`kQ3True`) or not (`kQ3False`).
`direction`	The direction of the directional light. Note that the direction is defined as a world-space vector *away from* the light source. This vector does not need to be normalized, but its length must be greater than 0.

Point Light Data Structure

You use a **point light data structure** to get or set information about a point light source. A point light data structure is defined by the `TQ3PointLightData` data type.

```
typedef struct TQ3PointLightData {
    TQ3LightData                    lightData;
    TQ3Boolean                      castsShadows;
    TQ3AttenuationType              attenuation;
    TQ3Point3D                      location;
} TQ3PointLightData;
```

Field descriptions

`lightData`	A light data structure specifying basic information about the point light.
`castsShadows`	A Boolean value that indicates whether the point light casts shadows (`kQ3True`) or not (`kQ3False`).
`attenuation`	The type of attenuation of the point light. See "Light Attenuation Values" (page 638) for a description of the constants this field can contain.
`location`	The location of the point light, in world coordinates.

Spot Light Data Structure

You use a **spot light data structure** to get or set information about a spot light source. A spot light data structure is defined by the `TQ3SpotLightData` data type.

```
typedef struct TQ3SpotLightData {
    TQ3LightData                    lightData;
    TQ3Boolean                      castsShadows;
    TQ3AttenuationType              attenuation;
    TQ3Point3D                      location;
    TQ3Vector3D                     direction;
    float                           hotAngle;
    float                           outerAngle;
    TQ3FallOffType                  fallOff;
} TQ3SpotLightData;
```

Field descriptions

`lightData`	A light data structure specifying basic information about the spot light.
`castsShadows`	A Boolean value that indicates whether the spot light casts shadows (`kQ3True`) or not (`kQ3False`).

`attenuation`	The type of attenuation of the spot light. See "Light Attenuation Values" (page 638) for a description of the constants that can be used in this field.
`location`	The location of the spot light, in world coordinates.
`direction`	The direction of the spot light. Note that the direction is defined as a world-space vector *away from* the light source. This vector does not need to be normalized, but vectors returned by QuickDraw 3D in this field might be normalized.
`hotAngle`	The hot angle of the spot light. The hot angle of a spot light is the half-angle, measured in radians, from the center of the cone of light within which the light remains at constant full intensity. The value in this field is a floating-point number in the range 0.0 to $\pi/2$, inclusive.
`outerAngle`	The outer angle of the spot light. The outer angle of a spot light is the half angle, measured in radians, from the center of the cone of light to the edge of the light's influence. The value in this field is a floating-point number in the range 0.0 to $\pi/2$, inclusive, and should always be greater than or equal to the value in the `hotAngle` field.
`fallOff`	The fall-off value for the spot light. See "Light Fall-Off Values" (page 638) for a description of the constants that can be used in this field.

Light Objects Routines

This section describes routines you can use to manage lights.

Managing Lights

QuickDraw 3D provides a number of general routines for managing lights of any kind.

Q3Light_GetType

You can use the `Q3Light_GetType` function to get the type of a light object.

```
TQ3ObjectType Q3Light_GetType (TQ3LightObject light);
```

`light` A light object.

DESCRIPTION

The `Q3Light_GetType` function returns, as its function result, the type of the light object specified by the `light` parameter. The types of light objects currently supported by QuickDraw 3D are defined by these constants:

```
kQ3LightTypeAmbient
kQ3LightTypeDirectional
kQ3LightTypePoint
kQ3LightTypeSpot
```

If the specified light object is invalid or is not one of these types, `Q3Light_GetType` returns the value `kQ3ObjectTypeInvalid`.

Q3Light_GetState

You can use the `Q3Light_GetState` function to get the current state of a light.

```
TQ3Status Q3Light_GetState (
                    TQ3LightObject light,
                    TQ3Boolean *isOn);
```

`light` A light object.

`isOn` On exit, the current state of the light specified by the `light` parameter.

DESCRIPTION

The `Q3Light_GetState` function returns, in the `isOn` parameter, a Boolean value that indicates whether the light specified by the `light` parameter is active (`kQ3True`) or inactive (`kQ3False`).

Q3Light_SetState

You can use the `Q3Light_SetState` function to set the state of a light.

```
TQ3Status Q3Light_SetState (
                    TQ3LightObject light,
                    TQ3Boolean isOn);
```

`light` A light object.

`isOn` The desired state of the specified light.

DESCRIPTION

The `Q3Light_SetState` function sets the state of the light specified by the `light` parameter to the value specified by the `isOn` parameter. If `isOn` is set to `kQ3True`, the light is made active; if `isOn` is set to `kQ3False`, the light is made inactive.

Q3Light_GetBrightness

You can use the `Q3Light_GetBrightness` function to get the current brightness of a light.

```
TQ3Status Q3Light_GetBrightness (
                    TQ3LightObject light,
                    float *brightness);
```

`light` A light object.

`brightness` On exit, the current brightness of the specified light.

DESCRIPTION

The `Q3Light_GetBrightness` function returns, in the `brightness` parameter, a value that indicates the current brightness of the light specified by the `light` parameter. The value should be between 0.0 and 1.0, inclusive. Some renderers may allow you to specify overbright lights (where the value in this field is greater than 1.0) or lights with negative brightness (where the value in this field is less than 0.0). If you have a light that has non-linear distance attenuation, you may need brightness values above 1.0 for realistic modeling (imagine the sun). The effects produced by out-of-range brightness values are renderer-specific.

Q3Light_SetBrightness

You can use the `Q3Light_SetBrightness` function to set the brightness of a light.

```
TQ3Status Q3Light_SetBrightness (
                    TQ3LightObject light,
                    float brightness);
```

`light` A light object.

`brightness` The desired brightness of the specified light.

DESCRIPTION

The `Q3Light_SetBrightness` function sets the brightness of the light specified by the `light` parameter to the value specified by the `brightness` parameter. The value should be between 0.0 and 1.0, inclusive. Some renderers may allow you to specify overbright lights (where the value in this field is greater than 1.0) or lights with negative brightness (where the value in this field is less than 0.0). If you have a light that has non-linear distance attenuation, you may need brightness values above 1.0 for realistic modeling (imagine the sun). The effects produced by out-of-range brightness values are renderer-specific.

Q3Light_GetColor

You can use the `Q3Light_GetColor` function to get the current color of a light.

```
TQ3Status Q3Light_GetColor (
                    TQ3LightObject light,
                    TQ3ColorRGB *color);
```

`light` A light object.

`color` On exit, a pointer to a `TQ3ColorRGB` structure specifying the current color of the specified light.

DESCRIPTION

The `Q3Light_GetColor` function returns, in the `color` parameter, the current color of the light specified by the `light` parameter.

Q3Light_SetColor

You can use the `Q3Light_SetColor` function to set the color of a light.

```
TQ3Status Q3Light_SetColor (
                    TQ3LightObject light,
                    const TQ3ColorRGB *color);
```

`light` A light object.

`color` A pointer to a `TQ3ColorRGB` structure specifying the desired color of the specified light.

DESCRIPTION

The `Q3Light_SetColor` function sets the color of the light specified by the `light` parameter to the value specified by the `color` parameter.

Q3Light_GetData

You can use the `Q3Light_GetData` function to get the basic data associated with a light.

```
TQ3Status Q3Light_GetData (
                    TQ3LightObject light,
                    TQ3LightData *lightData);
```

`light` A light object.

`lightData` On exit, a pointer to a light data structure.

DESCRIPTION

The `Q3Light_GetData` function returns, through the `lightData` parameter, basic information about the light specified by the `light` parameter. See "Light Data Structure" (page 639) for a description of a light data structure.

Q3Light_SetData

You can use the `Q3Light_SetData` function to set the basic data associated with a light.

```
TQ3Status Q3Light_SetData (
                    TQ3LightObject light,
                    const TQ3LightData *lightData);
```

`light` A light object.

`lightData` A pointer to a light data structure.

DESCRIPTION

The `Q3Light_SetData` function sets the data associated with the light specified by the `light` parameter to the data specified by the `lightData` parameter.

Managing Ambient Light

QuickDraw 3D provides routines that you can use to create and edit the ambient light of a view.

Q3AmbientLight_New

You can use the `Q3AmbientLight_New` function to create a new ambient light.

```
TQ3LightObject Q3AmbientLight_New (
                    const TQ3LightData *lightData);
```

`lightData` A pointer to a light data structure.

DESCRIPTION

The `Q3AmbientLight_New` function returns, as its function result, a new ambient light having the characteristics specified by the `lightData` parameter.

Q3AmbientLight_GetData

You can use the `Q3AmbientLight_GetData` function to get the data that defines an ambient light.

```
TQ3Status Q3AmbientLight_GetData (
                    TQ3LightObject light,
                    TQ3LightData *lightData);
```

`light` An ambient light object.

`lightData` On exit, a pointer to a light data structure.

DESCRIPTION

The `Q3AmbientLight_GetData` function returns, through the `lightData` parameter, information about the ambient light specified by the `light` parameter. See "Light Data Structure" (page 639) for a description of a light data structure.

Q3AmbientLight_SetData

You can use the `Q3AmbientLight_SetData` function to set the data that defines an ambient light.

```
TQ3Status Q3AmbientLight_SetData (
                    TQ3LightObject light,
                    const TQ3LightData *lightData);
```

`light` An ambient light object.

`lightData` A pointer to a light data structure.

DESCRIPTION

The `Q3AmbientLight_SetData` function sets the data associated with the ambient light specified by the `light` parameter to the data specified by the `lightData` parameter.

Managing Directional Lights

QuickDraw 3D provides routines that you can use to create and edit directional lights.

Q3DirectionalLight_New

You can use the `Q3DirectionalLight_New` function to create a new directional light.

```
TQ3LightObject Q3DirectionalLight_New (
                    const TQ3DirectionalLightData
                    *directionalLightData);
```

`directionalLightData`
A pointer to a directional light data structure.

DESCRIPTION

The `Q3DirectionalLight_New` function returns, as its function result, a new directional light having the characteristics specified by the `directionalLightData` parameter.

Q3DirectionalLight_GetCastShadowsState

You can use the `Q3DirectionalLight_GetCastShadowsState` function to get the shadow-casting state of a directional light.

```
TQ3Status Q3DirectionalLight_GetCastShadowsState (
                    TQ3LightObject light,
                    TQ3Boolean *castsShadows);
```

`light` A directional light object.

`castsShadows`
On exit, a Boolean value that indicates whether the specified light casts shadows (`kQ3True`) or does not cast shadows (`kQ3False`).

DESCRIPTION

The `Q3DirectionalLight_GetCastShadowsState` function returns, in the `castsShadows` parameter, a Boolean value that indicates whether the light specified by the `light` parameter casts shadows (`kQ3True`) or does not cast shadows (`kQ3False`).

Q3DirectionalLight_SetCastShadowsState

You can use the `Q3DirectionalLight_SetCastShadowsState` function to set the shadow-casting state of a directional light.

```
TQ3Status Q3DirectionalLight_SetCastShadowsState (
                    TQ3LightObject light,
                    TQ3Boolean castsShadows);
```

`light`	A directional light object.
`castsShadows`	A Boolean value that indicates whether the specified light casts shadows (`kQ3True`) or does not cast shadows (`kQ3False`).

DESCRIPTION

The `Q3DirectionalLight_SetCastShadowsState` function sets the shadow-casting state of the directional light specified by the `light` parameter to the Boolean value specified in the `castsShadows` parameter.

Q3DirectionalLight_GetDirection

You can use the `Q3DirectionalLight_GetDirection` function to get the direction of a directional light.

```
TQ3Status Q3DirectionalLight_GetDirection (
                    TQ3LightObject light,
                    TQ3Vector3D *direction);
```

`light`	A directional light object.
`direction`	On exit, the direction of the specified light.

DESCRIPTION

The `Q3DirectionalLight_GetDirection` function returns, in the `direction` parameter, the current direction of the directional light specified by the `light` parameter.

Q3DirectionalLight_SetDirection

You can use the `Q3DirectionalLight_SetDirection` function to set the direction of a directional light.

```
TQ3Status Q3DirectionalLight_SetDirection (
                    TQ3LightObject light,
                    const TQ3Vector3D *direction);
```

`light` A directional light object.

`direction` The desired direction of the specified light.

DESCRIPTION

The `Q3DirectionalLight_SetDirection` function sets the direction of the directional light specified by the `light` parameter to the value passed in the `direction` parameter.

Q3DirectionalLight_GetData

You can use the `Q3DirectionalLight_GetData` function to get the data that defines a directional light.

```
TQ3Status Q3DirectionalLight_GetData (
                    TQ3LightObject light,
                    TQ3DirectionalLightData *directionalLightData);
```

`light` A directional light object.

`directionalLightData`
On exit, a pointer to a directional light data structure.

DESCRIPTION

The `Q3DirectionalLight_GetData` function returns, through the `directionalLightData` parameter, information about the directional light specified by the `light` parameter. See "Directional Light Data Structure" (page 640) for a description of a directional light data structure.

Q3DirectionalLight_SetData

You can use the `Q3DirectionalLight_SetData` function to set the data that defines a directional light.

```
TQ3Status Q3DirectionalLight_SetData (
                    TQ3LightObject light,
                    const TQ3DirectionalLightData
                    *directionalLightData);
```

`light` A directional light object.

`directionalLightData`
A pointer to a directional light data structure.

DESCRIPTION

The `Q3DirectionalLight_SetData` function sets the data associated with the directional light specified by the `light` parameter to the data specified by the `directionalLightData` parameter.

Managing Point Lights

QuickDraw 3D provides routines that you can use to create and edit point lights.

Q3PointLight_New

You can use the `Q3PointLight_New` function to create a new point light.

```
TQ3LightObject Q3PointLight_New (
                    const TQ3PointLightData *pointLightData);
```

`pointLightData`
A pointer to a point light data structure.

DESCRIPTION

The `Q3PointLight_New` function returns, as its function result, a new point light having the characteristics specified by the `pointLightData` parameter.

Q3PointLight_GetCastShadowsState

You can use the `Q3PointLight_GetCastShadowsState` function to get the shadow-casting state of a point light.

```
TQ3Status Q3PointLight_GetCastShadowsState (
                    TQ3LightObject light,
                    TQ3Boolean *castsShadows);
```

`light` A point light object.

`castsShadows`
On exit, a Boolean value that indicates whether the specified light casts shadows (`kQ3True`) or does not cast shadows (`kQ3False`).

DESCRIPTION

The `Q3PointLight_GetCastShadowsState` function returns, in the `castsShadows` parameter, a Boolean value that indicates whether the light specified by the `light` parameter casts shadows (`kQ3True`) or does not cast shadows (`kQ3False`).

Q3PointLight_SetCastShadowsState

You can use the `Q3PointLight_SetCastShadowsState` function to set the shadow-casting state of a point light.

```
TQ3Status Q3PointLight_SetCastShadowsState (
                    TQ3LightObject light,
                    TQ3Boolean castsShadows);
```

`light` A point light object.

`castsShadows`
A Boolean value that indicates whether the specified light casts shadows (`kQ3True`) or does not cast shadows (`kQ3False`).

DESCRIPTION

The `Q3PointLight_SetCastShadowsState` function sets the shadow-casting state of the point light specified by the `light` parameter to the Boolean value specified in the `castsShadows` parameter.

Q3PointLight_GetAttenuation

You can use the `Q3PointLight_GetAttenuation` function to get the attenuation of a point light.

```
TQ3Status Q3PointLight_GetAttenuation (
                    TQ3LightObject light,
                    TQ3AttenuationType *attenuation);
```

`light` A point light object.

`attenuation` On exit, the type of attenuation of the light. See "Light Attenuation Values" (page 638) for a description of the constants that can be returned in this parameter.

DESCRIPTION

The `Q3PointLight_GetAttenuation` function returns, in the `attenuation` parameter, the current attenuation value of the point light specified by the `light` parameter.

Q3PointLight_SetAttenuation

You can use the `Q3PointLight_SetAttenuation` function to set the attenuation of a point light.

```
TQ3Status Q3PointLight_SetAttenuation (
                    TQ3LightObject light,
                    TQ3AttenuationType attenuation);
```

`light` A point light object.

`attenuation` The desired type of attenuation of the light. See "Light Attenuation Values" (page 638) for a description of the constants that can be passed in this parameter.

DESCRIPTION

The `Q3PointLight_SetAttenuation` function sets the attenuation value of the point light specified by the `light` parameter to the value passed in the `attenuation` parameter.

Q3PointLight_GetLocation

You can use the `Q3PointLight_GetLocation` function to get the location of a point light.

```
TQ3Status Q3PointLight_GetLocation (
                    TQ3LightObject light,
                    TQ3Point3D *location);
```

`light` A point light object.

`location` On exit, the location of the point light, in world coordinates.

DESCRIPTION

The `Q3PointLight_GetLocation` function returns, in the `location` parameter, the current location of the point light specified by the `light` parameter.

Q3PointLight_SetLocation

You can use the `Q3PointLight_SetLocation` function to set the location of a point light.

```
TQ3Status Q3PointLight_SetLocation (
                    TQ3LightObject light,
                    const TQ3Point3D *location);
```

`light` A point light object.

`location` The desired location of the point light, in world coordinates.

DESCRIPTION

The `Q3PointLight_SetLocation` function sets the location of the point light specified by the `light` parameter to the value passed in the `location` parameter.

Q3PointLight_GetData

You can use the `Q3PointLight_GetData` function to get the data that defines a point light.

```
TQ3Status Q3PointLight_GetData (
                    TQ3LightObject light,
                    TQ3PointLightData *pointLightData);
```

`light` A point light object.

`pointLightData`
On exit, a pointer to a point light data structure.

DESCRIPTION

The `Q3PointLight_GetData` function returns, through the `pointLightData` parameter, information about the point light specified by the `light` parameter. See “Point Light Data Structure” (page 640) for a description of a point light data structure.

Q3PointLight_SetData

You can use the `Q3PointLight_SetData` function to set the data that defines a point light.

```
TQ3Status Q3PointLight_SetData (
                    TQ3LightObject light,
                    const TQ3PointLightData *pointLightData);
```

`light` A point light object.

`pointLightData`
A pointer to a point light data structure.

DESCRIPTION

The `Q3PointLight_SetData` function sets the data associated with the point light specified by the `light` parameter to the data specified by the `pointLightData` parameter.

Managing Spot Lights

QuickDraw 3D provides routines that you can use to create and edit spot lights.

Q3SpotLight_New

You can use the `Q3SpotLight_New` function to create a new spot light.

```
TQ3LightObject Q3SpotLight_New (
                    const TQ3SpotLightData *spotLightData);
```

`spotLightData`
A pointer to a spot light data structure.

DESCRIPTION

The `Q3SpotLight_New` function returns, as its function result, a new spot light having the characteristics specified by the `spotLightData` parameter.

Q3SpotLight_GetCastShadowsState

You can use the `Q3SpotLight_GetCastShadowsState` function to get the shadow-casting state of a spot light.

```
TQ3Status Q3SpotLight_GetCastShadowsState (
                    TQ3LightObject light,
                    TQ3Boolean *castsShadows);
```

`light` A spot light object.

`castsShadows`
On exit, a Boolean value that indicates whether the specified light casts shadows (`kQ3True`) or does not cast shadows (`kQ3False`).

DESCRIPTION

The `Q3SpotLight_GetCastShadowsState` function returns, in the `castsShadows` parameter, a Boolean value that indicates whether the light specified by the `light` parameter casts shadows (`kQ3True`) or does not cast shadows (`kQ3False`).

Q3SpotLight_SetCastShadowsState

You can use the `Q3SpotLight_SetCastShadowsState` function to set the shadow-casting state of a spot light.

```
TQ3Status Q3SpotLight_SetCastShadowsState (
                    TQ3LightObject light,
                    TQ3Boolean castsShadows);
```

`light` A spot light object.

`castsShadows`
A Boolean value that indicates whether the specified light casts shadows (`kQ3True`) or does not cast shadows (`kQ3False`).

DESCRIPTION

The `Q3SpotLight_SetCastShadowsState` function sets the shadow-casting state of the spot light specified by the `light` parameter to the Boolean value specified in the `castsShadows` parameter.

Q3SpotLight_GetAttenuation

You can use the `Q3SpotLight_GetAttenuation` function to get the attenuation of a spot light.

```
TQ3Status Q3SpotLight_GetAttenuation (
                    TQ3LightObject light,
                    TQ3AttenuationType *attenuation);
```

`light` A spot light object.

`attenuation` On exit, the type of attenuation of the light. See "Light Attenuation Values" (page 638) for a description of the constants that can be returned in this parameter.

DESCRIPTION

The `Q3SpotLight_GetAttenuation` function returns, in the `attenuation` parameter, the current attenuation value of the spot light specified by the `light` parameter.

Q3SpotLight_SetAttenuation

You can use the `Q3SpotLight_SetAttenuation` function to set the attenuation of a spot light.

```
TQ3Status Q3SpotLight_SetAttenuation (
                    TQ3LightObject light,
                    TQ3AttenuationType attenuation);
```

`light` A spot light object.

`attenuation` The desired type of attenuation of the light. See "Light Attenuation Values" (page 638) for a description of the constants that can be passed in this parameter.

DESCRIPTION

The `Q3SpotLight_SetAttenuation` function sets the attenuation value of the spot light specified by the `light` parameter to the value passed in the `attenuation` parameter.

Q3SpotLight_GetLocation

You can use the `Q3SpotLight_GetLocation` function to get the location of a spot light.

```
TQ3Status Q3SpotLight_GetLocation (
                    TQ3LightObject light,
                    TQ3Point3D *location);
```

`light` A spot light object.

`location` On exit, the location of the spot light, in world coordinates.

DESCRIPTION

The `Q3SpotLight_GetLocation` function returns, in the `location` parameter, the current location of the spot light specified by the `light` parameter.

Q3SpotLight_SetLocation

You can use the `Q3SpotLight_SetLocation` function to set the location of a spot light.

```
TQ3Status Q3SpotLight_SetLocation (
                    TQ3LightObject light,
                    const TQ3Point3D *location);
```

`light`	A spot light object.
`location`	The desired location of the spot light, in world coordinates.

DESCRIPTION

The `Q3SpotLight_SetLocation` function sets the location of the spot light specified by the `light` parameter to the value passed in the `location` parameter.

Q3SpotLight_GetDirection

You can use the `Q3SpotLight_GetDirection` function to get the direction of a spot light.

```
TQ3Status Q3SpotLight_GetDirection (
                    TQ3LightObject light,
                    TQ3Vector3D *direction);
```

`light`	A spot light object.
`direction`	On exit, the direction of the specified light.

DESCRIPTION

The `Q3SpotLight_GetDirection` function returns, in the `direction` parameter, the current direction of the spot light specified by the `light` parameter.

Q3SpotLight_SetDirection

You can use the `Q3SpotLight_SetDirection` function to set the direction of a spot light.

```
TQ3Status Q3SpotLight_SetDirection (
                    TQ3LightObject light,
                    const TQ3Vector3D *direction);
```

`light`	A spot light object.

`direction` The desired direction of the specified light.

DESCRIPTION

The `Q3SpotLight_SetDirection` function sets the direction of the spot light specified by the `light` parameter to the value passed in the `direction` parameter.

Q3SpotLight_GetHotAngle

You can use the `Q3SpotLight_GetHotAngle` function to get the hot angle of a spot light.

```
TQ3Status Q3SpotLight_GetHotAngle (
                    TQ3LightObject light,
                    float *hotAngle);
```

`light` A spot light object.

`hotAngle` On exit, the hot angle of the specified light, in radians.

DESCRIPTION

The `Q3SpotLight_GetHotAngle` function returns, in the `hotAngle` parameter, the current hot angle of the spot light specified by the `light` parameter.

Q3SpotLight_SetHotAngle

You can use the `Q3SpotLight_SetHotAngle` function to set the hot angle of a spot light.

```
TQ3Status Q3SpotLight_SetHotAngle (
                    TQ3LightObject light,
                    float hotAngle);
```

`light` A spot light object.

`hotAngle` The desired hot angle of the specified light, in radians.

DESCRIPTION

The `Q3SpotLight_SetHotAngle` function sets the hot angle of the spot light specified by the `light` parameter to the value passed in the `hotAngle` parameter.

Q3SpotLight_GetOuterAngle

You can use the `Q3SpotLight_GetOuterAngle` function to get the outer angle of a spot light.

```
TQ3Status Q3SpotLight_GetOuterAngle (
                    TQ3LightObject light,
                    float *outerAngle);
```

`light` A spot light object.

`outerAngle` On exit, the outer angle of the specified light, in radians.

DESCRIPTION

The `Q3SpotLight_GetOuterAngle` function returns, in the `outerAngle` parameter, the current outer angle of the spot light specified by the `light` parameter.

Q3SpotLight_SetOuterAngle

You can use the `Q3SpotLight_SetOuterAngle` function to set the outer angle of a spot light.

```
TQ3Status Q3SpotLight_SetOuterAngle (
                    TQ3LightObject light,
                    float outerAngle);
```

`light` A spot light object.

`outerAngle` The desired outer angle of the specified light, in radians.

DESCRIPTION

The `Q3SpotLight_SetOuterAngle` function sets the outer angle of the spot light specified by the `light` parameter to the value passed in the `outerAngle` parameter.

Q3SpotLight_GetFallOff

You can use the `Q3SpotLight_GetFallOff` function to get the fall-off value of a spot light.

```
TQ3Status Q3SpotLight_GetFallOff (
                    TQ3LightObject light,
                    TQ3FallOffType *fallOff);
```

`light` A spot light object.

`fallOff` On exit, the fall-off value of the specified spot light. See "Light Fall-Off Values" (page 638) for a description of the constants that can be returned in this parameter.

DESCRIPTION

The `Q3SpotLight_GetFallOff` function returns, in the `fallOff` parameter, the current fall-off value of the spot light specified by the `light` parameter.

Q3SpotLight_SetFallOff

You can use the `Q3SpotLight_SetFallOff` function to set the fall-off value of a spot light.

```
TQ3Status Q3SpotLight_SetFallOff (
                    TQ3LightObject light,
                    TQ3FallOffType fallOff);
```

`light` A spot light object.

`fallOff` The desired fall-off value of the specified spot light. See "Light Fall-Off Values" (page 638) for a description of the constants that can be passed in this parameter.

DESCRIPTION

The `Q3SpotLight_SetFallOff` function sets the fall-off value of the spot light specified by the `light` parameter to the value passed in the `fallOff` parameter.

Q3SpotLight_GetData

You can use the `Q3SpotLight_GetData` function to get the data that defines a spot light.

```
TQ3Status Q3SpotLight_GetData (
                    TQ3LightObject light,
                    TQ3SpotLightData *spotLightData);
```

`light` A spot light object.

`spotLightData`
On exit, a pointer to a spot light data structure.

DESCRIPTION

The `Q3SpotLight_GetData` function returns, through the `spotLightData` parameter, information about the spot light specified by the `light` parameter. See "Spot Light Data Structure" (page 641) for a description of a spot light data structure.

Q3SpotLight_SetData

You can use the `Q3SpotLight_SetData` function to set the data that defines a spot light.

```
TQ3Status Q3SpotLight_SetData (
                    TQ3LightObject light,
                    const TQ3SpotLightData *spotLightData);
```

`light` A spot light object.

`spotLightData`
A pointer to a spot light data structure.

DESCRIPTION

The `Q3SpotLight_SetData` function sets the data associated with the spot light specified by the `light` parameter to the data specified by the `spotLightData` parameter.

Light Notices

The following notices may be returned by light functions. A list of general QuickDraw 3D errors is given in "QuickDraw 3D Errors, Warnings, and Notices" (page 87).

```
kQ3NoticeInvalidAttenuationTypeUsingInternalDefaults
kQ3NoticeBrightnessGreaterThanOne
```

CHAPTER 8

Light Objects

CHAPTER 9

Camera Objects

This chapter describes camera objects (or cameras) and the functions you can use to manipulate them. You use cameras to specify the location of the viewer, the direction of viewing, the portion of the view plane to be rendered, and other information about a scene. A single camera is associated with a view, along with a list of lights and other settings that affect the rendering of a model.

You should already be familiar with the QuickDraw 3D class hierarchy, described in the chapter "QuickDraw 3D Objects." For information about associating a camera with a view, see the chapter "View Objects."

This chapter begins by describing camera objects and their features. Then it shows how to create and manipulate cameras. The section "Camera Objects Reference," beginning on page 683 provides a complete description of camera objects and the routines you can use to create and manipulate them.

About Camera Objects

A **camera object** (or, more briefly, a **camera**) is a type of QuickDraw 3D object that you use to define a point of view, a range of visible objects, and a method of projection for generating a two-dimensional image of those objects from a three-dimensional model. A camera is of type `TQ3CameraObject`, which is a type of shape object.

QuickDraw 3D defines three types of cameras:

- orthographic cameras
- view plane cameras
- aspect ratio cameras

These types of cameras differ in their methods of projection, as explained more fully later in this section. All three types of cameras share some basic properties, which are maintained in a camera data structure, defined by the `TQ3CameraData` data structure.

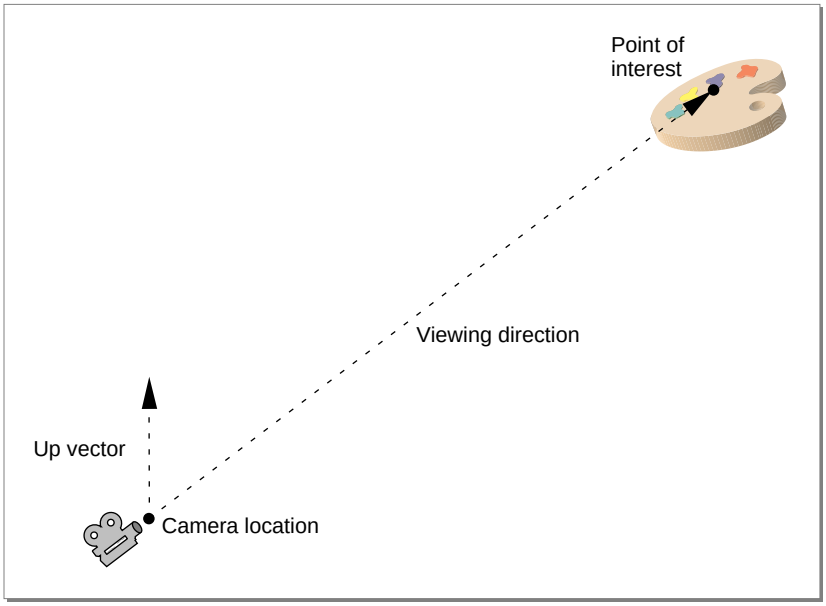
```
typedef struct TQ3CameraData {
    TQ3CameraPlacement                  placement;
    TQ3CameraRange                      range;
    TQ3CameraViewPort                   viewPort;
} TQ3CameraData;
```

These fields specify the location and orientation of the camera, the visible range of interest, and the camera's view port and projection method. The following sections explain these concepts in greater detail.

Camera Placements

A **camera location** is the position, in the world coordinate system, of a camera. A **camera placement** is a camera location together with an orientation and a direction. You specify a camera's orientation by indicating its **up vector,** the vector that defines which direction is up. You specify a camera's direction by indicating a **point of interest,** the point at which the camera is aimed. The vector that is the result of subtracting the camera location from the point of interest is the **viewing direction** or **camera vector.** In general, a camera's up vector should be perpendicular to its viewing direction and should be normalized. You can, however, specify any up vector that isn't colinear with the viewing direction. Figure 9-1 shows the placement of a camera.

Figure 9-1 A camera's placement



Note
Because a camera defines a point of view onto a model, the camera location is also called the *eye point*. ◆

In QuickDraw 3D, you specify a camera's placement by filling in the fields of a **camera placement structure,** defined by the `TQ3CameraPlacement` data type.

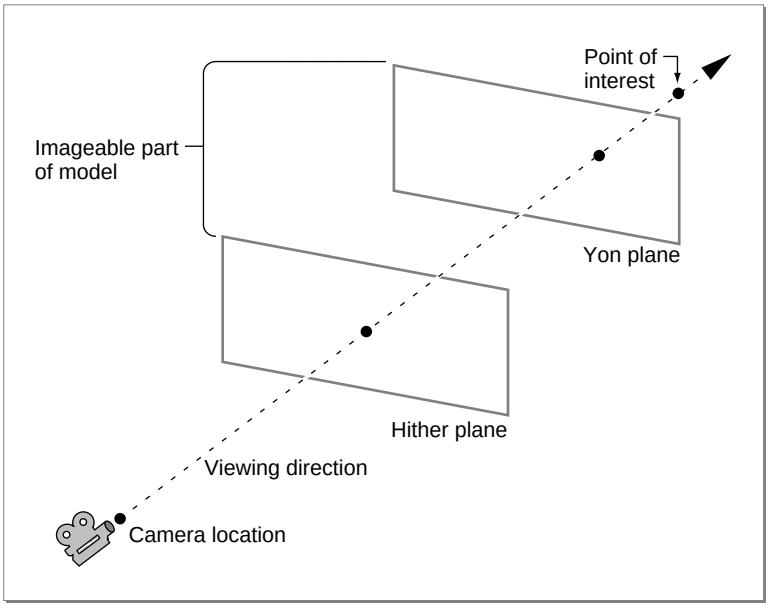
```
typedef struct TQ3CameraPlacement {
    TQ3Point3D                      cameraLocation;
    TQ3Point3D                      pointOfInterest;
    TQ3Vector3D                     upVector;
} TQ3CameraPlacement;
```

See "Camera Placement Structure" (page 683) for complete information about the camera placement structure.

Camera Ranges

Often, you're not interested in all the objects in a model that are visible from the current placement of a camera. Some objects may be too far away from the camera location to create a useful image when projected onto the two-dimensional view plane, and some objects may be so close to the camera that they obscure other important objects. QuickDraw 3D, like most 3D graphics systems, provides a mechanism for ignoring objects that lie outside your current range of interest. You do this by defining two **clipping planes** that delimit the part of a model that is rendered. The **hither plane** is a plane perpendicular to the viewing direction that indicates the clipping range closest to the camera. Any objects or parts of objects that lie between the camera and the hither plane do not appear in a rendered image. Similarly, the **yon plane** is a plane perpendicular to the viewing direction that indicates the clipping range farthest from the camera. Any objects or parts of objects that lie beyond the yon plane do not appear in a rendered image. In short, only objects or parts of objects that lie between the hither and yon planes appear in a rendered image, as shown in Figure 9-2.

Figure 9-2 The hither and yon planes



Note
The hither and yon planes are sometimes called the *near* and *far* planes, respectively. ◆

The extent between the hither and yon planes of a camera is the **camera range,** defined by the `TQ3CameraRange` data structure.

```
typedef struct TQ3CameraRange {
    float                               hither;
    float                               yon;
} TQ3CameraRange;
```

The clipping planes are specified by distances along the viewing direction from the camera location. The distance to the yon plane should always be greater than the distance to the hither plane, and the distance to the hither plane should always be greater than 0.0.

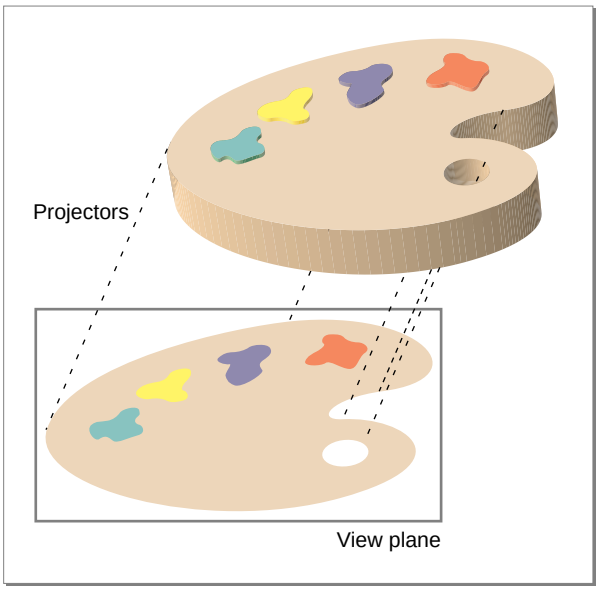
View Planes and View Ports

As you've learned, QuickDraw 3D provides three different types of cameras, which are distinguished from one another by their method of **projection**—that is, by their method of generating a two-dimensional image of the objects in a three-dimensional model. A projection of an object is the set of points in which rays emanating from the object (called **projectors**) intersect a plane (called the **view plane**). The projection created when the projectors are all parallel to one another is called a **parallel projection,** and the projection created when the projectors all intersect in a point is called a **perspective projection.** The point at which the projectors in a perspective projection intersect one another is the **center of projection.**

Note
Currently, QuickDraw 3D provides only normal view planes, where the view plane is perpendicular to the viewing direction. ◆

Figure 9-3 illustrates a parallel projection of an object.

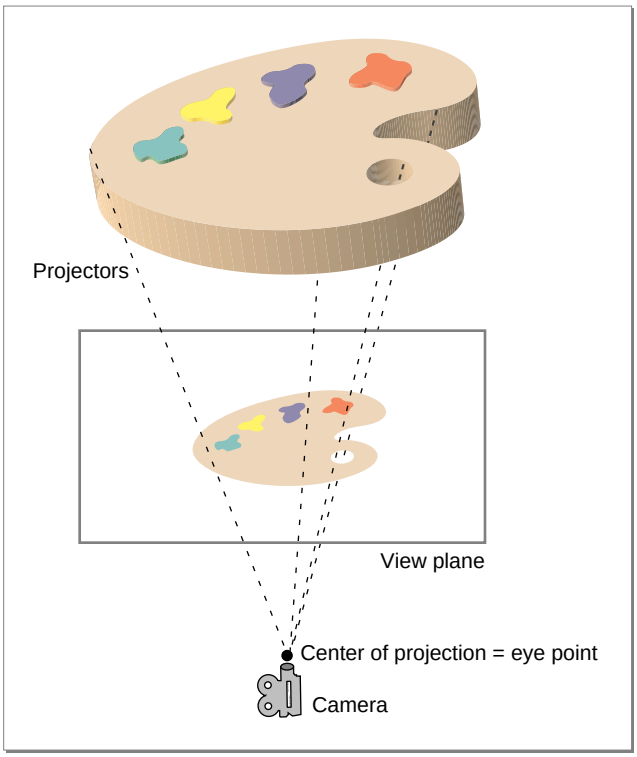
Figure 9-3 A parallel projection of an object



Notice that, because the projectors are parallel, the size of the two-dimensional image corresponds exactly to the size of the three-dimensional object being projected, no matter where the view plane is located. As a result, you do not need to specify the location of the view plane when using parallel projections. See “Orthographic Cameras” (page 677) for details on how to specify a parallel projection.

Figure 9-4 illustrates a perspective projection of an object.

Figure 9-4 A perspective projection of an object



As you can see, the location of the view plane is very important in a perspective projection. When the view plane is close to the camera, the projectors are close together and the image they create is small. Conversely, when the view plane is farther away from the camera, the projectors are farther apart and the image they create is larger. Similarly, no matter where the view plane is located, the size of the projected image of an object is inversely proportional to the distance of the object from the view plane. Objects farther away from the view plane appear smaller than objects of the same size closer to the view plane. This effect is **perspective foreshortening.**

When using perspective projection, you therefore need to specify the location of the view plane. QuickDraw 3D provides two types of perspective cameras, which specify the location of the view plane in different ways. See “View Plane

Cameras" (page 679) and "Aspect Ratio Cameras" (page 681) for complete details on these two types of perspective cameras.

A **camera view port** is the rectangular portion of the view plane that is to be mapped into the area specified by the current draw context. A draw context is usually just a window, so the view port defines the portion of the view plane that appears in the window. By default, a camera's view port is the entire square portion of the view plane bounded by the view volume (either a box, for parallel projections, or a frustum, for perspective projections). Figure 9-5 (page 677) shows the default camera view port for a perspective camera.

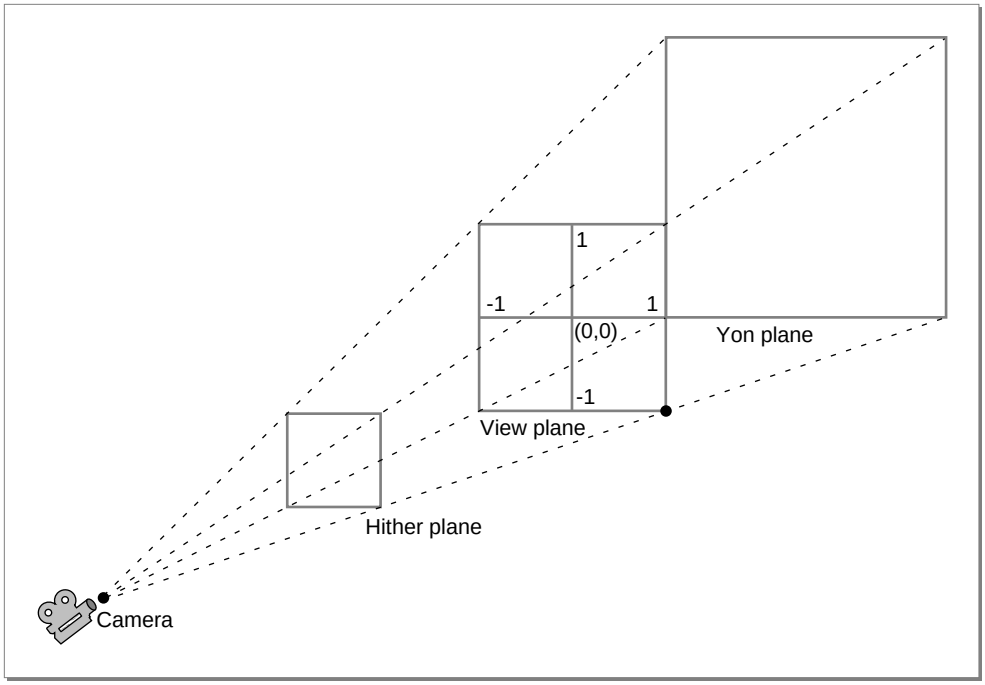
You can select a smaller portion of the view plane by filling in a camera view port structure, defined by the `TQ3CameraViewPort` data type.

```
typedef struct TQ3CameraViewPort {
    TQ3Point2D                          origin;
    float                               width;
    float                               height;
} TQ3CameraViewPort;
```

For example, to display only the right side of the view plane, you would set the `origin` field to the point (0, 1), the `width` field to the value 1.0, and the `height` field to the value 2.0.

Note
The image displayed in a draw context is not necessarily the image drawn on the view port. The view port image is scaled to fit into the draw context pane and then clipped with the draw context mask. See the chapter "Draw Context Objects" for information about draw context panes and masks, and for further details on the relationship between a view port and a draw context. ◆

Figure 9-5 The default camera view port



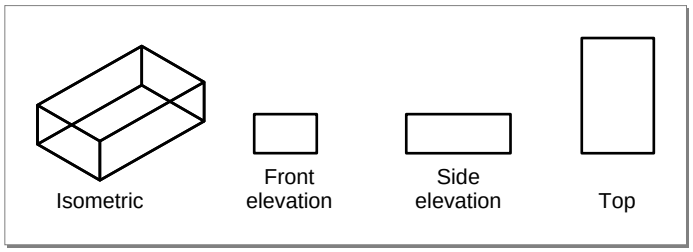
Orthographic Cameras

An **orthographic camera** is a camera that uses parallel projection to generate a two-dimensional image of the objects in a three-dimensional model. In particular, an orthographic camera uses **orthographic projection,** in which the view plane is perpendicular to the viewing direction. Parallel projections are in general less realistic than perspective projections, but they have the advantage that parallel lines in a model remain parallel in the projection, and distances are not distorted by perspective foreshortening.

The two most common types of orthographic projection are isometric projection and elevation projection. An **isometric projection** is an orthographic projection in which the viewing direction makes equal angles with each of the three principal axes of an object. An **elevation projection** is an orthographic projection in which the view plane is perpendicular to one of the principal axes

of the object being projected. Figure 9-6 shows isometric and elevation projections of an object.

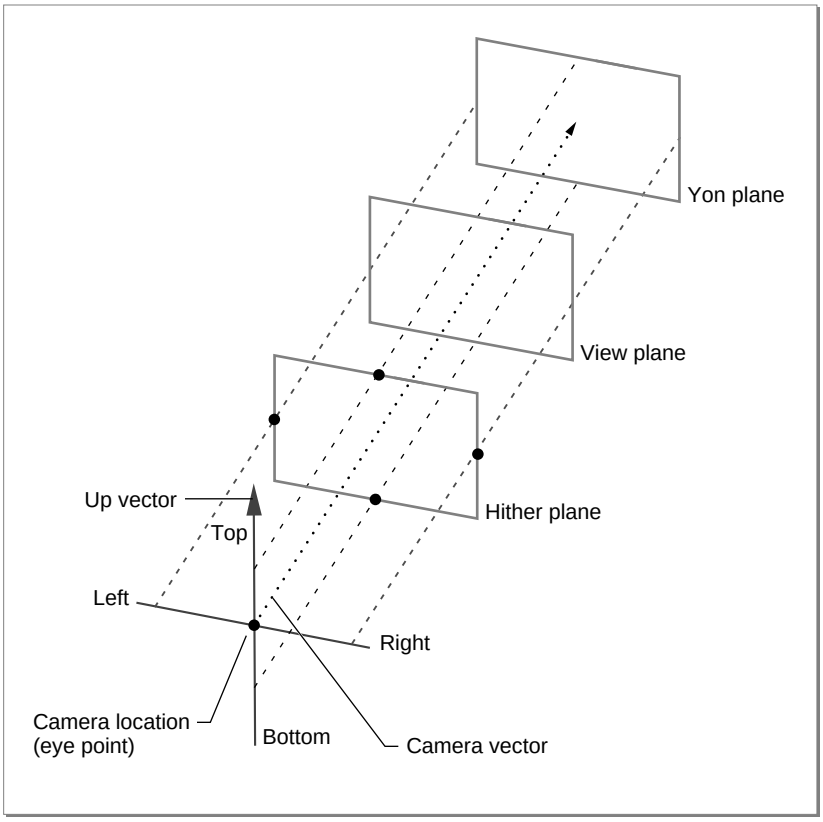
Figure 9-6 Isometric and elevation projections



The view volume associated with an orthographic camera is determined by a box aligned with the viewing direction, as shown in Figure 9-7. To specify the box, you provide the left side, top side, right side, and bottom side. The values you use to specify these sides are relative to the **camera coordinate system** defined by the camera location and the viewing direction. The box defines the four horizontal and vertical clipping planes.

See “Orthographic Camera Data Structure” (page 686) for details on the data you need to provide to define an orthographic camera. See “Managing Orthographic Cameras,” beginning on page 694 for a description of the routines you can use to create and manipulate orthographic cameras.

Figure 9-7 An orthographic camera



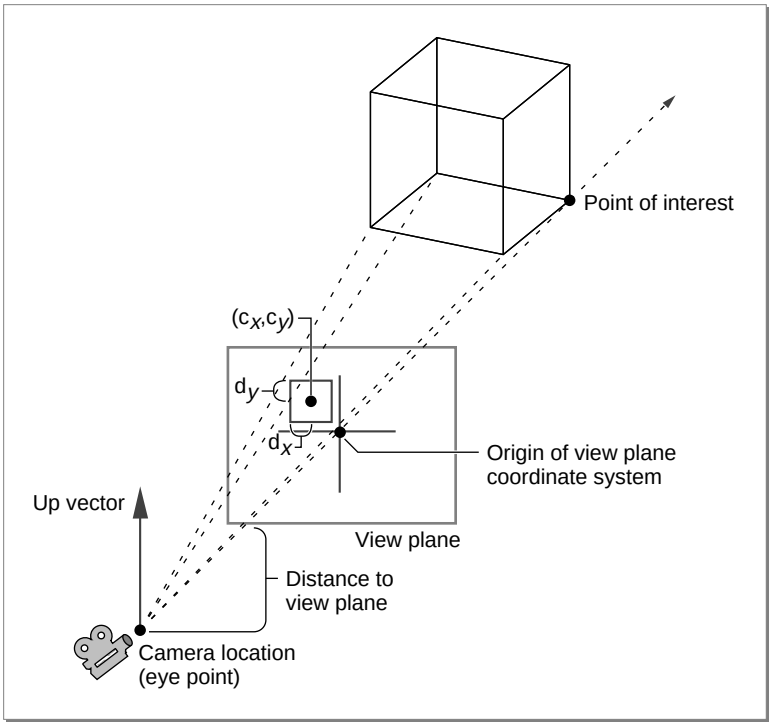
View Plane Cameras

A **view plane camera** is a type of perspective camera defined in terms of an arbitrary view plane. In general, you'll use a view plane camera to create a perspective image of a specific object in a scene. The view plane camera is the only type of perspective camera provided by QuickDraw 3D that allows **off-axis viewing** (that is, viewing where the center of the projected object on the view plane is not on the camera vector), which is convenient when scrolling an image up or down, or left to right.

The view frustum associated with a view plane camera is determined by a view plane (located at a specified distance from the camera) and the rectangular cross

section of an object, as shown in Figure 9-8. The point at which the camera vector intersects the view plane defines the origin of the **view plane coordinate system.** You specify a rectangular cross section of an object by specifying its center (in the view plane coordinate system) and the half-width and half-height of the cross section. In Figure 9-8, the center of the cross section is the point (c_x, c_y), and the half-width and half-height are the distances d_x and d_y, respectively.

Figure 9-8 A view plane camera

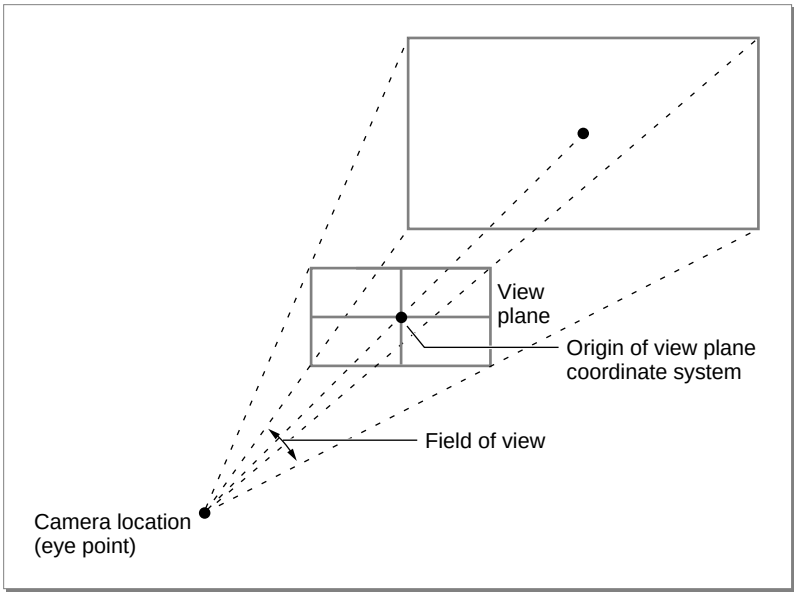


See "View Plane Camera Data Structure" (page 686) for complete details on the data you need to provide to define a view plane camera. See "Managing View Plane Cameras," beginning on page 700 for a description of the routines you can use to create and manipulate view plane cameras.

Aspect Ratio Cameras

An **aspect ratio camera** is a type of perspective camera defined in terms of a viewing angle and a horizontal-to-vertical aspect ratio, as shown in Figure 9-9. With an aspect ratio camera, you don't specify the distance to the view plane directly (as you do with a view plane camera).

Figure 9-9 An aspect ratio camera

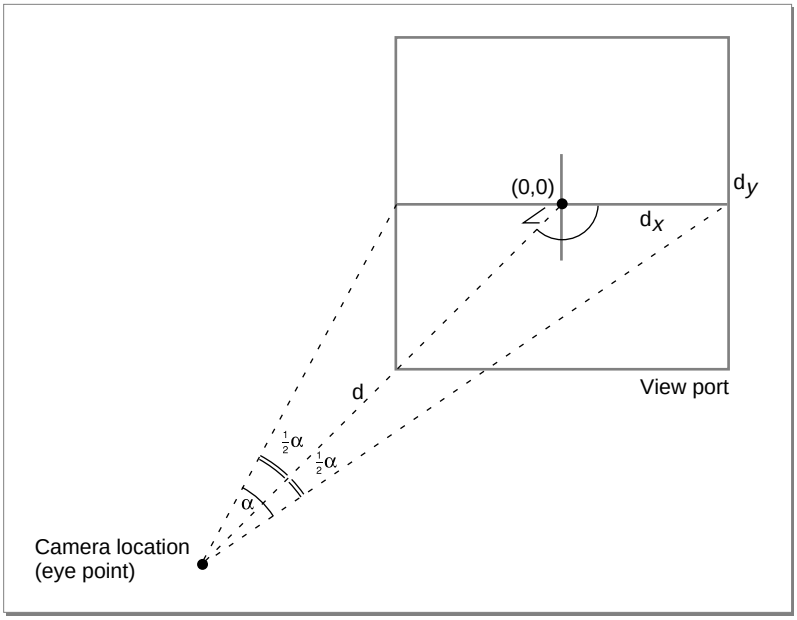


The orientation of the field of view is determined by the specified aspect ratio. If the aspect ratio is greater than 1.0, the field of view is vertical. If the aspect ratio is less than 1.0, the field of view is horizontal. In general, to avoid distortion, the aspect ratio should be the same as the aspect ratio of the camera's view port.

You can easily see that as the field of view increases, the view plane must move closer to camera location for the view port to fit within the field of view, in which case the image size decreases (because of perspective foreshortening). Conversely, as the field of view decreases, the view plane must move away from the camera location, and the image size increases.

Note that you can always find a view plane camera that is projectively identical to any aspect ratio camera. (The converse is not true: it's not always possible to find an aspect ratio camera that is projectively identical to an arbitrary view plane camera.) Consider the aspect ratio camera shown in Figure 9-10. It's easy to specify a view plane camera that creates the same image as that aspect ratio camera. To do this, set the center of the cross section (c_x, c_y) to be the origin $(0, 0)$, and set the half-width d_x to be the quantity $d \tan(\alpha/2)$, where d is the distance from the camera to the view plane and α is the horizontal field of view. (The half-angle applies to the smaller of the two view port dimensions.)

Figure 9-10 The relation between aspect ratio cameras and view plane cameras



See "Aspect Ratio Camera Data Structure" (page 687) for more details on the data you need to provide to define an aspect ratio camera. See "Managing Aspect Ratio Cameras," beginning on page 707 for a description of the routines you can use to create and manipulate aspect ratio cameras.

Using Camera Objects

You create a camera object by filling in the fields of the appropriate data structure (for example, a structure of type `TQ3ViewAngleAspectCameraData` for an aspect ratio camera) and calling an appropriate constructor function (for example, `Q3ViewAngleAspectCamera_New` for an aspect ratio camera). Then, no matter what kind of camera you've created, you need to attach the camera to a view object, by calling the `Q3View_SetCamera` function. See Listing 1-8 (page 66) and Listing 1-9 (page 67) for complete code samples that create a camera and attach it to a view object.

You can change the characteristics of a view's camera by calling camera object editing routines. For example, you can change the aspect ratio of an aspect ratio camera by calling the `Q3ViewAngleAspectCamera_SetAspectRatio` function.

Camera Objects Reference

This section describes the QuickDraw 3D data structures and routines that you can use to create and manage camera objects.

Data Structures

This section describes the data structures supplied by QuickDraw 3D for managing cameras. The data structures used to manage cameras are all public.

Camera Placement Structure

You use a **camera placement structure** to get or set information about the location and orientation of a camera. A camera placement structure is defined by the `TQ3CameraPlacement` data type.

```
typedef struct TQ3CameraPlacement {
    TQ3Point3D                          cameraLocation;
    TQ3Point3D                          pointOfInterest;
    TQ3Vector3D                         upVector;
} TQ3CameraPlacement;
```

Field descriptions

`cameraLocation` The location of the camera, in world-space coordinates.

`pointOfInterest` The camera's point of interest (that is, the point at which the camera is aimed), in world-space coordinates.

`upVector` The up-vector of the camera, which specifies the orientation of the camera. It must be normalized and perpendicular to the viewing direction. The up-vector of a camera is mapped to the *y* axis of the view plane.

Camera Range Structure

You use a **camera range structure** to get or set the hither and yon clipping planes for a camera. A camera range structure is defined by the `TQ3CameraRange` data type.

```
typedef struct TQ3CameraRange {
    float                               hither;
    float                               yon;
} TQ3CameraRange;
```

Field descriptions

`hither` The distance (measured along the camera vector) from the camera's location to the near clipping plane. The value in this field should always be greater than 0.

`yon` The distance (measured along the camera vector) from the camera's location to the far clipping plane. The value in this field should always be greater than the value in the `hither` field.

Camera View Port Structure

You use a **camera view port structure** to get or set information about the view port of a camera. A camera's view port defines the rectangular portion of the

view plane that is to be mapped into the area specified by the current draw context. The default settings for a view port describe the entire view plane, where the origin (–1.0, 1.0) is the upper-left corner and the width and height of the plane are both 2.0. A camera view port structure is defined by the `TQ3CameraViewPort` data type.

```
typedef struct TQ3CameraViewPort {
    TQ3Point2D                          origin;
    float                               width;
    float                               height;
} TQ3CameraViewPort;
```

Field descriptions

`origin`	The origin of the view port. The values of the `x` and `y` fields of this point should be between –1.0 and 1.0.
`width`	The width of the view port. The value in this field should be greater than 0.0 and less than 2.0.
`height`	The height of the view port. The value in this field should be greater than 0.0 and less than 2.0.

Camera Data Structure

You use a **camera data structure** to get or set basic information about a camera of any kind. A camera data structure is defined by the `TQ3CameraData` data type.

```
typedef struct TQ3CameraData {
    TQ3CameraPlacement                  placement;
    TQ3CameraRange                      range;
    TQ3CameraViewPort                   viewPort;
} TQ3CameraData;
```

Field descriptions

`placement`	A camera placement structure that specifies the current placement and orientation of the camera.
`range`	A camera range structure that specifies the current hither and yon clipping planes for the camera.
`viewPort`	A camera view port structure that specifies the current view port of the camera.

Orthographic Camera Data Structure

You use an **orthographic camera data structure** to get or set information about an orthographic camera. An orthographic camera data structure is defined by the `TQ3OrthographicCameraData` data type.

```
typedef struct TQ3OrthographicCameraData {
    TQ3CameraData                   cameraData;
    float                           left;
    float                           top;
    float                           right;
    float                           bottom;
} TQ3OrthographicCameraData;
```

Field descriptions

`cameraData`	A camera data structure specifying basic information about the orthographic camera.
`left`	The left side of the orthographic camera. The value of this field (and the following three fields) is relative to the camera coordinate system.
`top`	The top side of the orthographic camera.
`right`	The right side of the orthographic camera.
`bottom`	The bottom side of the orthographic camera.

View Plane Camera Data Structure

You use a **view plane camera data structure** to get or set information about a view plane camera. A view plane camera data structure is defined by the `TQ3ViewPlaneCameraData` data type.

```
typedef struct TQ3ViewPlaneCameraData {
    TQ3CameraData                   cameraData;
    float                           viewPlane;
    float                           halfWidthAtViewPlane;
    float                           halfHeightAtViewPlane;
    float                           centerXOnViewPlane;
    float                           centerYOnViewPlane;
} TQ3ViewPlaneCameraData;
```

Field descriptions

`cameraData` A camera data structure specifying basic information about the view plane camera.

`viewPlane` The distance to the view plane from the location of the camera. The value in this field must be greater than 0.0. The view plane should be set at the object whose dimensions and location are specified by the following four fields.

`halfWidthAtViewPlane` One half the width of the cross section of an object.

`halfHeightAtViewPlane` The value in the `halfWidthAtViewPlane` field divided by the aspect ratio of the view port.

`centerXOnViewPlane` The *x* coordinate of the center of the object in the view plane.

`centerYOnViewPlane` The *y* coordinate of the center of the object in the view plane.

Aspect Ratio Camera Data Structure

You use an **aspect ratio camera data structure** to get or set information about an aspect ratio camera. An aspect ratio camera data structure is defined by the `TQ3ViewAngleAspectCameraData` data type.

```
typedef struct TQ3ViewAngleAspectCameraData {
    TQ3CameraData                       cameraData;
    float                               fov;
    float                               aspectRatioXToY;
} TQ3ViewAngleAspectCameraData;
```

Field descriptions

`cameraData` A camera data structure specifying basic information about the aspect ratio camera.

`fov` The camera's maximum field of view. This parameter should contain a positive floating-point value specified in radians. If the value in the `aspectRatioXToY` field is greater than 1.0, the field of view is vertical; if the value in the

	`aspectRatioXToY` field is less than 1.0, the field of view is horizontal.
`aspectRatioXToY`	The camera's horizontal-to-vertical aspect ratio. To avoid distortion, this ratio should be the same as the ratio of the width to the height of the camera's view port.

Camera Objects Routines

This section describes the routines you can use to manage cameras.

Managing Cameras

QuickDraw 3D provides a number of general routines for managing cameras of any kind.

Q3Camera_GetType

You can use the `Q3Camera_GetType` function to get type of a camera.

```
TQ3ObjectType Q3Camera_GetType (TQ3CameraObject camera);
```

`camera` A camera object.

DESCRIPTION

The `Q3Camera_GetType` function returns, as its function result, the type of the camera specified by the `camera` parameter. The types of camera currently supported by QuickDraw 3D are defined by these constants:

```
kQ3CameraTypeOrthographic
kQ3CameraTypeViewAngleAspect
kQ3CameraTypeViewPlane
```

If `Q3Camera_GetType` cannot determine the type of the specified camera, it returns `kQ3ObjectTypeInvalid`.

Q3Camera_GetData

You can use the `Q3Camera_GetData` function to get the basic data associated with a camera.

```
TQ3Status Q3Camera_GetData (
                    TQ3CameraObject camera,
                    TQ3CameraData *cameraData);
```

`camera` A camera object.

`cameraData` On exit, a pointer to a camera data structure.

DESCRIPTION

The `Q3Camera_GetData` function returns, through the `cameraData` parameter, basic information about the camera specified by the `camera` parameter. See "Camera Data Structure" (page 685) for a description of a camera data structure.

Q3Camera_SetData

You can use the `Q3Camera_SetData` function to set the basic data associated with a camera.

```
TQ3Status Q3Camera_SetData (
                    TQ3CameraObject camera,
                    const TQ3CameraData *cameraData);
```

`camera` A camera object.

`cameraData` A pointer to a camera data structure.

DESCRIPTION

The `Q3Camera_SetData` function sets the data associated with the camera specified by the `camera` parameter to the data specified by the `cameraData` parameter.

Q3Camera_GetPlacement

You can use the `Q3Camera_GetPlacement` function to get the current placement of a camera.

```
TQ3Status Q3Camera_GetPlacement (
                    TQ3CameraObject camera,
                    TQ3CameraPlacement *placement);
```

`camera` A camera object.

`placement` On exit, a pointer to a camera placement structure.

DESCRIPTION

The `Q3Camera_GetPlacement` function returns, in the `placement` parameter, a pointer to a camera placement structure that describes the current placement of the camera specified by the `camera` parameter.

Q3Camera_SetPlacement

You can use the `Q3Camera_SetPlacement` function to set the placement of a camera.

```
TQ3Status Q3Camera_SetPlacement (
                    TQ3CameraObject camera,
                    const TQ3CameraPlacement *placement);
```

`camera` A camera object.

`placement` A pointer to a camera placement structure.

DESCRIPTION

The `Q3Camera_SetPlacement` function sets the placement of the camera specified by the `camera` parameter to the position specified by the `placement` parameter.

Q3Camera_GetRange

You can use the `Q3Camera_GetRange` function to get the current range of a camera.

```
TQ3Status Q3Camera_GetRange (
                    TQ3CameraObject camera,
                    TQ3CameraRange *range);
```

`camera` A camera object.

`range` On exit, a pointer to a camera range structure.

DESCRIPTION

The `Q3Camera_GetRange` function returns, in the `range` parameter, a pointer to a camera range structure that describes the current range of the camera specified by the `camera` parameter.

Q3Camera_SetRange

You can use the `Q3Camera_SetRange` function to set the range of a camera.

```
TQ3Status Q3Camera_SetRange (
                    TQ3CameraObject camera,
                    const TQ3CameraRange *range);
```

`camera` A camera object.

`range` A pointer to a camera range structure.

DESCRIPTION

The `Q3Camera_SetRange` function sets the range of the camera specified by the `camera` parameter to the range specified by the `range` parameter.

Q3Camera_GetViewPort

You can use the `Q3Camera_GetViewPort` function to get the current view port of a camera.

```
TQ3Status Q3Camera_GetViewPort (
                    TQ3CameraObject camera,
                    TQ3CameraViewPort *viewPort);
```

`camera` A camera object.

`viewPort` On exit, a pointer to a camera view port structure.

DESCRIPTION

The `Q3Camera_GetViewPort` function returns, in the `viewPort` parameter, a pointer to a camera view port structure that describes the current view port of the camera specified by the `camera` parameter.

Q3Camera_SetViewPort

You can use the `Q3Camera_SetViewPort` function to set the view port of a camera.

```
TQ3Status Q3Camera_SetViewPort (
                    TQ3CameraObject camera,
                    const TQ3CameraViewPort *viewPort);
```

`camera` A camera object.

`viewPort` A pointer to a camera view port structure.

DESCRIPTION

The `Q3Camera_SetViewPort` function sets the view port of the camera specified by the `camera` parameter to the view port specified by the `viewPort` parameter.

Q3Camera_GetWorldToView

You can use the `Q3Camera_GetWorldToView` function to get the current world-to-view space transform.

```
TQ3Status Q3Camera_GetWorldToView (
                    TQ3CameraObject camera,
                    TQ3Matrix4x4 *worldToView);
```

`camera` A camera object.

`worldToView` On output, a pointer to a 4-by-4 matrix.

DESCRIPTION

The `Q3Camera_GetWorldToView` function returns, in the `worldToView` parameter, a pointer to a 4-by-4 matrix that describes the current world-to-view space transform defined by the camera specified by the `camera` parameter. The world-to-view space transform is defined only by the placement of the camera; it establishes the camera location as the origin of the view space, with the view vector (that is, the vector from the camera's eye toward the point of interest) placed along the –*z* axis and the up vector placed along the *y* axis.

Q3Camera_GetViewToFrustum

You can use the `Q3Camera_GetViewToFrustum` function to get the current view-to-frustum transform.

```
TQ3Status Q3Camera_GetViewToFrustum (
                    TQ3CameraObject camera,
                    TQ3Matrix4x4 *viewToFrustum);
```

`camera` A camera object.

`viewToFrustum`
On output, a pointer to a 4-by-4 matrix.

DESCRIPTION

The `Q3Camera_GetViewToFrustum` function returns, in the `viewToFrustum` parameter, a pointer to a 4-by-4 matrix that describes the current view-to-frustum transform defined by the camera specified by the `camera` parameter.

Q3Camera_GetWorldToFrustum

You can use the `Q3Camera_GetWorldToFrustum` function to get the current world-to-frustum transform.

```
TQ3Status Q3Camera_GetWorldToFrustum (
                    TQ3CameraObject camera,
                    TQ3Matrix4x4 *worldToFrustum);
```

`camera` A camera object.

`worldToFrustum`
On output, a pointer to a 4-by-4 matrix.

DESCRIPTION

The `Q3Camera_GetWorldToFrustum` function returns, in the `worldToFrustum` parameter, a pointer to a 4-by-4 matrix that describes the current world-to-frustum transform defined by the camera specified by the `camera` parameter.

Managing Orthographic Cameras

QuickDraw 3D provides routines that you can use to create and edit orthographic cameras.

Q3OrthographicCamera_New

You can use the `Q3OrthographicCamera_New` function to create a new orthographic camera.

```
TQ3CameraObject Q3OrthographicCamera_New (
                    const TQ3OrthographicCameraData *orthographicData);
```

`orthographicData`
A pointer to an orthographic camera data structure.

DESCRIPTION

The `Q3OrthographicCamera_New` function returns, as its function result, a new orthographic camera having the camera characteristics specified by the `orthographicData` parameter.

Q3OrthographicCamera_GetData

You can use the `Q3OrthographicCamera_GetData` function to get the data that defines an orthographic camera.

```
TQ3Status Q3OrthographicCamera_GetData (
                    TQ3CameraObject camera,
                    TQ3OrthographicCameraData *cameraData);
```

`camera` An orthographic camera object.

`cameraData` On exit, a pointer to an orthographic camera data structure.

DESCRIPTION

The `Q3OrthographicCamera_GetData` function returns, through the `cameraData` parameter, information about the orthographic camera specified by the `camera` parameter. See "Orthographic Camera Data Structure" (page 686) for the structure of an orthographic camera data structure.

Q3OrthographicCamera_SetData

You can use the `Q3OrthographicCamera_SetData` function to set the data that defines an orthographic camera.

```
TQ3Status Q3OrthographicCamera_SetData (
                    TQ3CameraObject camera,
                    const TQ3OrthographicCameraData *cameraData);
```

`camera` An orthographic camera object.

`cameraData` A pointer to an orthographic camera data structure.

DESCRIPTION

The `Q3OrthographicCamera_SetData` function sets the data associated with the orthographic camera specified by the `camera` parameter to the data specified by the `cameraData` parameter.

Q3OrthographicCamera_GetLeft

You can use the `Q3OrthographicCamera_GetLeft` function to get the left side of an orthographic camera.

```
TQ3Status Q3OrthographicCamera_GetLeft (
                    TQ3CameraObject camera,
                    float *left);
```

`camera` An orthographic camera object.

`left` On exit, the left side of the specified orthographic camera.

DESCRIPTION

The `Q3OrthographicCamera_GetLeft` function returns, in the `left` parameter, a value that specifies the left side of the orthographic camera specified by the `camera` parameter.

Q3OrthographicCamera_SetLeft

You can use the `Q3OrthographicCamera_SetLeft` function to set the left side of an orthographic camera.

```
TQ3Status Q3OrthographicCamera_SetLeft (
                    TQ3CameraObject camera,
                    float left);
```

`camera` An orthographic camera object.

`left` The desired left side of the specified orthographic camera.

DESCRIPTION

The `Q3OrthographicCamera_SetLeft` function sets the left side of the orthographic camera specified by the `camera` parameter to the value specified by the `left` parameter.

Q3OrthographicCamera_GetTop

You can use the `Q3OrthographicCamera_GetTop` function to get the top side of an orthographic camera.

```
TQ3Status Q3OrthographicCamera_GetTop (
                    TQ3CameraObject camera,
                    float *top);
```

`camera` An orthographic camera object.

`top` On exit, the top side of the specified orthographic camera.

DESCRIPTION

The `Q3OrthographicCamera_GetTop` function returns, in the `top` parameter, a value that specifies the top side of the orthographic camera specified by the `camera` parameter.

Q3OrthographicCamera_SetTop

You can use the `Q3OrthographicCamera_SetTop` function to set the top side of an orthographic camera.

```
TQ3Status Q3OrthographicCamera_SetTop (
                    TQ3CameraObject camera,
                    float top);
```

`camera` An orthographic camera object.

`top` The desired top side of the specified orthographic camera.

DESCRIPTION

The `Q3OrthographicCamera_SetTop` function sets the top side of the orthographic camera specified by the `camera` parameter to the value specified by the `top` parameter.

Q3OrthographicCamera_GetRight

You can use the `Q3OrthographicCamera_GetRight` function to get the right side of an orthographic camera.

```
TQ3Status Q3OrthographicCamera_GetRight (
                    TQ3CameraObject camera,
                    float *right);
```

`camera` An orthographic camera object.

`right` On exit, the right side of the specified orthographic camera.

DESCRIPTION

The `Q3OrthographicCamera_GetRight` function returns, in the `right` parameter, a value that specifies the right side of the orthographic camera specified by the `camera` parameter.

Q3OrthographicCamera_SetRight

You can use the `Q3OrthographicCamera_SetRight` function to set the right side of an orthographic camera.

```
TQ3Status Q3OrthographicCamera_SetRight (
                    TQ3CameraObject camera,
                    float right);
```

`camera` An orthographic camera object.

`right` The desired right side of the specified orthographic camera.

DESCRIPTION

The `Q3OrthographicCamera_SetRight` function sets the right side of the orthographic camera specified by the `camera` parameter to the value specified by the `right` parameter.

Q3OrthographicCamera_GetBottom

You can use the `Q3OrthographicCamera_GetBottom` function to get the bottom side of an orthographic camera.

```
TQ3Status Q3OrthographicCamera_GetBottom (
                    TQ3CameraObject camera,
                    float *bottom);
```

`camera` An orthographic camera object.

`bottom` On exit, the bottom side of the specified orthographic camera.

DESCRIPTION

The `Q3OrthographicCamera_GetBottom` function returns, in the `bottom` parameter, a value that specifies the bottom side of the orthographic camera specified by the `camera` parameter.

Q3OrthographicCamera_SetBottom

You can use the `Q3OrthographicCamera_SetBottom` function to set the bottom side of an orthographic camera.

```
TQ3Status Q3OrthographicCamera_SetBottom (
                    TQ3CameraObject camera,
                    float bottom);
```

`camera` An orthographic camera object.

`bottom` The desired bottom side of the specified orthographic camera.

DESCRIPTION

The `Q3OrthographicCamera_SetBottom` function sets the bottom side of the orthographic camera specified by the `camera` parameter to the value specified by the `bottom` parameter.

Managing View Plane Cameras

QuickDraw 3D provides routines that you can use to create and edit view plane cameras.

Q3ViewPlaneCamera_New

You can use the `Q3ViewPlaneCamera_New` function to create a new view plane camera.

```
TQ3CameraObject Q3ViewPlaneCamera_New (
                    const TQ3ViewPlaneCameraData *cameraData);
```

`cameraData` A pointer to a view plane camera data structure.

DESCRIPTION

The `Q3ViewPlaneCamera_New` function returns, as its function result, a new view plane camera having the camera characteristics specified by the `cameraData` parameter.

Q3ViewPlaneCamera_GetData

You can use the `Q3ViewPlaneCamera_GetData` function to get the data that defines a view plane camera.

```
TQ3Status Q3ViewPlaneCamera_GetData (
                    TQ3CameraObject camera,
                    TQ3ViewPlaneCameraData *cameraData);
```

`camera` A view plane camera object.

`cameraData` On exit, a pointer to a view plane camera data structure.

DESCRIPTION

The `Q3ViewPlaneCamera_GetData` function returns, through the `cameraData` parameter, information about the view plane camera specified by the `camera` parameter. See "View Plane Camera Data Structure" (page 686) for the structure of a view plane camera data structure.

Q3ViewPlaneCamera_SetData

You can use the `Q3ViewPlaneCamera_SetData` function to set the data that defines a view plane camera.

```
TQ3Status Q3ViewPlaneCamera_SetData (
                    TQ3CameraObject camera,
                    const TQ3ViewPlaneCameraData *cameraData);
```

`camera` A view plane camera object.

`cameraData` A pointer to a view plane camera data structure.

DESCRIPTION

The `Q3ViewPlaneCamera_SetData` function sets the data associated with the view plane camera specified by the `camera` parameter to the data specified by the `cameraData` parameter.

Q3ViewPlaneCamera_GetViewPlane

You can use the `Q3ViewPlaneCamera_GetViewPlane` function to get the current distance of the view plane from a view plane camera.

```
TQ3Status Q3ViewPlaneCamera_GetViewPlane (
                    TQ3CameraObject camera,
                    float *viewPlane);
```

`camera`	A view plane camera object.
`viewPlane`	On exit, the distance of the view plane from the specified camera.

DESCRIPTION

The `Q3ViewPlaneCamera_GetViewPlane` function returns, in the `viewPlane` parameter, the distance of the view plane from the camera specified by the `camera` parameter.

Q3ViewPlaneCamera_SetViewPlane

You can use the `Q3ViewPlaneCamera_SetViewPlane` function to set the distance of the view plane from a view plane camera.

```
TQ3Status Q3ViewPlaneCamera_SetViewPlane (
                    TQ3CameraObject camera,
                    float viewPlane);
```

`camera`	A view plane camera object.
`viewPlane`	The desired distance of the view plane from the specified camera.

DESCRIPTION

The `Q3ViewPlaneCamera_SetViewPlane` function sets the distance from the camera specified by the `camera` parameter to its view plane to the value specified in the `viewPlane` parameter.

Q3ViewPlaneCamera_GetHalfWidth

You can use the `Q3ViewPlaneCamera_GetHalfWidth` function to get the half-width of the object specifying a view plane camera.

```
TQ3Status Q3ViewPlaneCamera_GetHalfWidth (
                    TQ3CameraObject camera,
                    float *halfWidthAtViewPlane);
```

`camera` A view plane camera object.

`halfWidthAtViewPlane`
On exit, the half-width of the cross section of the viewed object.

DESCRIPTION

The `Q3ViewPlaneCamera_GetHalfWidth` function returns, in the `halfWidthAtViewPlane` parameter, the half-width of the cross section of the viewed object of the camera specified by the `camera` parameter.

Q3ViewPlaneCamera_SetHalfWidth

You can use the `Q3ViewPlaneCamera_SetHalfWidth` function to set the half-width of the object specifying a view plane camera.

```
TQ3Status Q3ViewPlaneCamera_SetHalfWidth (
                    TQ3CameraObject camera,
                    float halfWidthAtViewPlane);
```

`camera` A view plane camera object.

`halfWidthAtViewPlane`
The desired half-width of the cross section of the viewed object of the specified camera.

DESCRIPTION

The `Q3ViewPlaneCamera_SetHalfWidth` function sets the half-width of the cross section of the viewed object of the camera specified by the `camera` parameter to the value specified in the `halfWidthAtViewPlane` parameter.

Q3ViewPlaneCamera_GetHalfHeight

You can use the `Q3ViewPlaneCamera_GetHalfHeight` function to get the half-height of the object specifying a view plane camera.

```
TQ3Status Q3ViewPlaneCamera_GetHalfHeight (
                    TQ3CameraObject camera,
                    float *halfHeightAtViewPlane);
```

`camera` A view plane camera object.

`halfHeightAtViewPlane`
On exit, the half-height of the cross section of the viewed object.

DESCRIPTION

The `Q3ViewPlaneCamera_GetHalfHeight` function returns, in the `halfHeightAtViewPlane` parameter, the half-height of the cross section of the viewed object of the camera specified by the `camera` parameter.

Q3ViewPlaneCamera_SetHalfHeight

You can use the `Q3ViewPlaneCamera_SetHalfHeight` function to set the half-height of the object specifying a view plane camera.

```
TQ3Status Q3ViewPlaneCamera_SetHalfHeight (
                    TQ3CameraObject camera,
                    float halfHeightAtViewPlane);
```

`camera` A view plane camera object.

`halfHeightAtViewPlane`
The desired half-height of the cross section of the viewed object.

DESCRIPTION

The `Q3ViewPlaneCamera_SetHalfHeight` function sets the half-height of the cross section of the viewed object of the camera specified by the `camera` parameter to the value specified in the `halfHeightAtViewPlane` parameter.

Q3ViewPlaneCamera_GetCenterX

You can use the `Q3ViewPlaneCamera_GetCenterX` function to get the horizontal center of the viewed object.

```
TQ3Status Q3ViewPlaneCamera_GetCenterX (
                    TQ3CameraObject camera,
                    float *centerXOnViewPlane);
```

`camera` A view plane camera object.

`centerXOnViewPlane`
On exit, the *x* coordinate of the center of the viewed object.

DESCRIPTION

The `Q3ViewPlaneCamera_GetCenterX` function returns, in the `centerXOnViewPlane` parameter, the *x* coordinate of the center of the viewed object of the camera specified by the `camera` parameter.

Q3ViewPlaneCamera_SetCenterX

You can use the `Q3ViewPlaneCamera_SetCenterX` function to set the horizontal center of the viewed object.

```
TQ3Status Q3ViewPlaneCamera_SetCenterX (
                    TQ3CameraObject camera,
                    float centerXOnViewPlane);
```

`camera` A view plane camera object.

`centerXOnViewPlane`
The desired *x* coordinate of the center of the viewed object.

DESCRIPTION

The `Q3ViewPlaneCamera_SetCenterX` function sets the *x* coordinate of the center of the viewed object of the camera specified by the `camera` parameter to the value specified in the `centerXOnViewPlane` parameter.

Q3ViewPlaneCamera_GetCenterY

You can use the `Q3ViewPlaneCamera_GetCenterY` function to get the vertical center of the viewed object.

```
TQ3Status Q3ViewPlaneCamera_GetCenterY (
                    TQ3CameraObject camera,
                    float *centerYOnViewPlane);
```

`camera` A view plane camera object.

`centerYOnViewPlane`
On exit, the *y* coordinate of the center of the viewed object.

DESCRIPTION

The `Q3ViewPlaneCamera_GetCenterY` function returns, in the `centerYOnViewPlane` parameter, the *y* coordinate of the center of the viewed object of the camera specified by the `camera` parameter.

Q3ViewPlaneCamera_SetCenterY

You can use the `Q3ViewPlaneCamera_SetCenterY` function to set the vertical center of the viewed object.

```
TQ3Status Q3ViewPlaneCamera_SetCenterY (
                    TQ3CameraObject camera,
                    float centerYOnViewPlane);
```

`camera` A view plane camera object.

`centerYOnViewPlane`
The desired *y* coordinate of the center of the viewed object.

DESCRIPTION

The `Q3ViewPlaneCamera_SetCenterY` function sets the *y* coordinate of the center of the viewed object of the camera specified by the `camera` parameter to the value specified in the `centerYOnViewPlane` parameter.

Managing Aspect Ratio Cameras

QuickDraw 3D provides routines that you can use to create and edit aspect ratio cameras.

Q3ViewAngleAspectCamera_New

You can use the `Q3ViewAngleAspectCamera_New` function to create a new aspect ratio camera.

```
TQ3CameraObject Q3ViewAngleAspectCamera_New (
                    const TQ3ViewAngleAspectCameraData *cameraData);
```

`cameraData` A pointer to an aspect ratio camera data structure.

DESCRIPTION

The `Q3ViewAngleAspectCamera_New` function returns, as its function result, a new aspect ratio camera having the camera characteristics specified by the `cameraData` parameter.

Q3ViewAngleAspectCamera_GetData

You can use the `Q3ViewAngleAspectCamera_GetData` function to get the data that defines an aspect ratio camera.

```
TQ3Status Q3ViewAngleAspectCamera_GetData (
                    TQ3CameraObject camera,
                    TQ3ViewAngleAspectCameraData *cameraData);
```

`camera` An aspect ratio camera object.

`cameraData` On exit, a pointer to an aspect ratio camera data structure.

DESCRIPTION

The `Q3ViewAngleAspectCamera_GetData` function returns, through the `cameraData` parameter, information about the aspect ratio camera specified by the `camera` parameter. See "Aspect Ratio Camera Data Structure" (page 687) for a description of an aspect ratio camera data structure.

Q3ViewAngleAspectCamera_SetData

You can use the `Q3ViewAngleAspectCamera_SetData` function to set the data that defines an aspect ratio camera.

```
TQ3Status Q3ViewAngleAspectCamera_SetData (
                    TQ3CameraObject camera,
                    const TQ3ViewAngleAspectCameraData *cameraData);
```

`camera` An aspect ratio camera object.

`cameraData` A pointer to an aspect ratio camera data structure.

DESCRIPTION

The `Q3ViewAngleAspectCamera_SetData` function sets the data associated with the aspect ratio camera specified by the `camera` parameter to the data specified by the `cameraData` parameter.

Q3ViewAngleAspectCamera_GetFOV

You can use the `Q3ViewAngleAspectCamera_GetFOV` function to get the maximum field of view of an aspect ratio camera.

```
TQ3Status Q3ViewAngleAspectCamera_GetFOV (
                    TQ3CameraObject camera,
                    float *fov);
```

`camera` An aspect ratio camera object.

`fov` On exit, the maximum field of view, in radians, of the specified camera.

DESCRIPTION

The `Q3ViewAngleAspectCamera_GetFOV` function returns, in the `fov` parameter, the maximum field of view of the aspect ratio camera specified by the `camera` parameter.

Q3ViewAngleAspectCamera_SetFOV

You can use the `Q3ViewAngleAspectCamera_SetFOV` function to set the maximum field of view of an aspect ratio camera.

```
TQ3Status Q3ViewAngleAspectCamera_SetFOV (
                    TQ3CameraObject camera,
                    float fov);
```

`camera` An aspect ratio camera object.

`fov` The desired maximum field of view, in radians, of the camera.

DESCRIPTION

The `Q3ViewAngleAspectCamera_SetFOV` function sets the maximum field of view of the camera specified by the `camera` parameter to the value specified in the `fov` parameter.

Q3ViewAngleAspectCamera_GetAspectRatio

You can use the `Q3ViewAngleAspectCamera_GetAspectRatio` function to get the aspect ratio of an aspect ratio camera.

```
TQ3Status Q3ViewAngleAspectCamera_GetAspectRatio (
                    TQ3CameraObject camera,
                    float *aspectRatioXToY);
```

`camera` An aspect ratio camera object.

`aspectRatioXToY`
On exit, the horizontal-to-vertical aspect ratio of the specified camera.

DESCRIPTION

The `Q3ViewAngleAspectCamera_GetAspectRatio` function returns, in the `aspectRatioXToY` parameter, the horizontal-to-vertical aspect ratio of the aspect ratio camera specified by the `camera` parameter.

Q3ViewAngleAspectCamera_SetAspectRatio

You can use the `Q3ViewAngleAspectCamera_SetAspectRatio` function to set the aspect ratio of an aspect ratio camera.

```
TQ3Status Q3ViewAngleAspectCamera_SetAspectRatio (
                    TQ3CameraObject camera,
                    float aspectRatioXToY);
```

`camera` An aspect ratio camera object.

`aspectRatioXToY`
The desired horizontal-to-vertical aspect ratio of the specified camera.

DESCRIPTION

The `Q3ViewAngleAspectCamera_SetAspectRatio` function sets the horizontal-to-vertical aspect ratio of the camera specified by the `camera` parameter to the value specified in the `aspectRatioXToY` parameter.

Camera Errors

The following error may be returned by camera routines. A list of general QuickDraw 3D errors is given in "QuickDraw 3D Errors, Warnings, and Notices" (page 87).

`kQ3ErrorInvalidCameraValues`

CHAPTER 9

Camera Objects

CHAPTER 10

Group Objects

This chapter describes group objects and the functions you can use to manipulate them. You can use groups to collect objects into lists or hierarchical models, which you can draw or otherwise manipulate with group object routines.

To use this chapter, you should already be familiar with the QuickDraw 3D class hierarchy, described in the chapter "QuickDraw 3D Objects" earlier in this book.

This chapter begins by describing group objects and their features. Then it shows how to create and manipulate groups. The section "Group Objects Reference," beginning on page 721 provides a complete description of the group objects and the routines you can use to create and manipulate them.

About Group Objects

A **group object** (or, more briefly, a **group**) is a type of QuickDraw 3D object that you can use to collect objects together into lists or hierarchical models. A group object is an instance of the `TQ3GroupObject` class. As you've seen, the `TQ3GroupObject` class is a subclass of the `TQ3ShapeObject`, which is itself a subclass of the `TQ3SharedObject` class. As a result, a group object is associated with a reference count, which is incremented or decremented whenever you create or dispose of an instance of that group.

The objects you put into in a group are not copied into the group. Instead, references to the objects are maintained in the group. Accordingly, you can include in a group only shared objects (that is, the types of objects that have reference counts). A group can contain other groups, because groups are shared objects. QuickDraw 3D provides functions that you can use to add objects to a

group or remove objects from a group. It also provides functions that you can use to access objects by their position in the group.

Group Types

The base class of group object is of type `kQ3ShapeTypeGroup`, a type of shape object. You can create a group of that type (by calling the `Q3Group_New` function) and you can put any kinds of shared objects into it (for example, by calling the `Q3Group_AddObject` function). In addition, QuickDraw 3D provides three subclasses of groups: light groups, display groups, and information groups. These subclasses are distinguished from one another by the kinds of objects you can put into them.

- A **light group** is a group that contains one or more lights (and no other types of QuickDraw 3D objects). You'll typically create light groups to provide illumination on the objects in a model. The light group is attached to a view object by calling the `Q3View_SetLightGroup` function. See the chapter "View Objects" for complete details on attaching light groups to views.
- A **display group** is a group of objects that are drawable. Drawable objects include geometric objects, styles, transforms, attributes and attribute sets, and other display groups. When you draw a display group into a view, each object in the group is executed (that is, drawn) in the order in which it appears in the group (which is determined by the order in which the objects were inserted into the group). You can create a display group, or you can create one of two subclasses of display groups: ordered display groups and I/O proxy display groups.
- An **ordered display group** is a display group in which the objects in the group are sorted by their type. Ordered groups are sometimes more useful than unordered groups because the order of object execution is always the same. During rendering, QuickDraw 3D executes objects in this order:

  1. transforms
  2. styles
  3. attribute sets
  4. shaders
  5. geometric objects
  6. groups
  7. unknown objects

This order of execution ensures that all transforms, styles, attribute sets, and shaders in a group are applied to the geometric objects, groups, and unknown objects that form the hierarchy below the ordered display group.

- An **I/O proxy display group** (or sometimes **proxy display group**) is a display group that contains several representations of a single geometric object. You can use I/O proxy display groups to encapsulate, in a metafile, two or more descriptions of an object. This is useful when an application reading the file is unable to understand some of those descriptions. For example, you might know that some other applications cannot handle NURB patches but do handle meshes. As a result, you can create an I/O proxy display group that contains two descriptions of a surface (one as a NURB patch and one as a mesh) and write that group into a metafile. Any application reading the metafile can select from the display group the representation of the surface that it can work with. You should put objects into the I/O proxy display group in the order you deem to be preferable. (In other words, the first object in the group should be the representation you deem most useful, and the last object should be the one that you deem least useful.) In this way, an application reading the metafile can simply use the first object in the proxy display group whose type is not `kQ3SharedTypeUnknown`.
- An **information group** is a group that contains one or more strings (and no other types of QuickDraw 3D objects). You'll typically create information groups to provide human-readable information in a metafile. For example, if you want to include a copyright notice in a metafile, you can simply create an information group that contains a string of the appropriate data and then write that group to the metafile.

Group Positions

You access an object within a group (for example, to remove the object from the group or to replace it with some other object) by referring to the object's group position. A **group position** is a pointer to a private (that is, opaque) data structure maintained internally by QuickDraw 3D. A group position is defined by the `TQ3GroupPosition` data type.

```
typedef struct TQ3GroupPositionPrivate      *TQ3GroupPosition;
```

You receive a group position for an object when you first insert the object into the group (for example, by calling `Q3Group_AddObject`). In general, however, you don't need to maintain that information, because you can use QuickDraw 3D

routines to walk through a group. For instance, you can get the group position of the first object in a group by calling `Q3Group_GetFirstPosition`. Then you can retrieve the positions of all subsequent objects in the group by calling `Q3Group_GetNextPosition`.

IMPORTANT

An object's group position is valid only as long as that object is in the group. When you remove an object from a group, the corresponding group position becomes invalid. Similarly, when you remove all objects from a group (for example, by calling `Q3Group_EmptyObjects`), the group positions of those objects become invalid. ▲

See "Accessing Objects by Position," beginning on page 718 for sample code that illustrates how to traverse a group using group positions.

Group State Flags

Every display group has **group state value** (built out of a set of **group state flags**) that determine how the group is traversed during rendering or picking, or during the computation of a bounding box or sphere. Here are the currently defined group state flags:

```
typedef enum TQ3DisplayGroupStateMasks {
    kQ3DisplayGroupStateNone                        = 0,
    kQ3DisplayGroupStateMaskIsDrawn                 = 1 << 0,
    kQ3DisplayGroupStateMaskIsInline                = 1 << 1,
    kQ3DisplayGroupStateMaskUseBoundingBox          = 1 << 2,
    kQ3DisplayGroupStateMaskUseBoundingSphere       = 1 << 3,
    kQ3DisplayGroupStateMaskIsPicked                = 1 << 4,
    kQ3DisplayGroupStateMaskIsWritten               = 1 << 5
} TQ3DisplayGroupStateMasks;

typedef unsigned long       TQ3DisplayGroupState;
```

A group state value contains a flag, called the **drawable flag,** that determines whether the group is to be drawn when it is passed to a view for rendering or picking. By default, the drawable flag of a group state value is set, indicating that the group is to be drawn to a view. If the drawable flag is clear, the group is not traversed when it is encountered in a hierarchical model. This allows you to

place “invisible” objects in a model that assist you in bounding complex geometric objects, for example.

An ordered display group can be constructed in such a way that the group has a hierarchical structure. This allows properties (such as attributes, styles, and transforms) to be inherited by child nodes from their parent nodes in the hierarchy. Occasionally, however, you might want to override this inheritance and allow a group contained in a hierarchical model to define its own graphics state independently of any other objects or groups in the model. To allow this feature, a group state value contains an **inline flag** that specifies whether or not the group should be executed inline. A group is executed **inline** if it does not push and pop the graphics state stack before and after it is executed (that is, if it is simply executed as a bundle of objects). By default, the inline flag of a group is not set, indicating that the group pushes and pops its graphics state.

For more information on pushing and popping the graphics state, see the descriptions of the functions `Q3Push_Submit` and `Q3Pop_Submit` in the chapter “View Objects.”

A group state value contains a **picking flag** that determines whether the group can be picked. In general, you’ll want all groups in a model to be eligible for picking. In some cases, however, you can clear the picking flag of a group’s group state value in order to establish the group as a decoration in the model that cannot be picked.

Using Group Objects

QuickDraw 3D provides functions that you can use to create a group, add objects to a group, remove objects from a group, and dispose of a group. It also provides functions that you can use to count the number of objects in a group, access objects by their position in the group, draw a group, pick objects in a group, and perform other operations on group objects. This section illustrates how to use some of these functions. In particular, it shows:

- how to create groups and add objects to them
- how to operate on all objects in a group, or on all objects of a particular type in a group

Creating Groups

You create a new light group, for example, by calling the `Q3LightGroup_New` function. If there is sufficient memory to create the group, `Q3LightGroup_New` returns to your application a reference to a group object, which you pass to other group routines. The new group is initially empty, and you add objects to the group by calling QuickDraw 3D routines (such as `Q3Group_AddObject`). When an object is added to a group, its reference count is incremented. (QuickDraw 3D uses the reference count to ensure that an object is not prematurely disposed.) If you don't want to maintain references to all the objects inside a group, you can use the technique illustrated in Listing 10-1.

Listing 10-1 Creating a group

```
myGroup = Q3LightGroup_New();
myLight = Q3SpotLight_New(mySpotLightData);
Q3Group_AddObject(myGroup, myLight);
Q3Object_Dispose(myLight);
```

By calling `Q3Object_Dispose`, you decrement the light's reference count once it's been added to the light group. When the group itself is later disposed of, QuickDraw 3D decrements the light's reference count, which may cause it also to be disposed of.

Accessing Objects by Position

You can iterate through a group by getting the position of its first object and then getting the positions of any subsequent objects. All groups, regardless of type, are stored in a single list which you can step through only by calling QuickDraw 3D routines.

Listing 10-2 shows how to access all the lights in a light group. The `MyTurnOnOrOffAllLights` function takes a view parameter and an on/off state value. It turns all the lights in the view's light group on or off, as specified by the state value.

Listing 10-2 Accessing all the lights in a light group

```
TQ3Status MyTurnOnOrOffViewLights (TQ3ViewObject myView, TQ3Boolean myState)
{
    TQ3GroupObject          myGroup;            /*the view's light group*/
    TQ3GroupPosition        myPos;              /*a group position*/
    TQ3Object               myLight;            /*a light*/
    TQ3Status               myResult;           /*a result code*/

    myResult = Q3View_GetLightGroup(myView, &myGroup);
    if (myResult == kQ3Failure)
        goto bail;

    for (Q3Group_GetFirstPosition(myGroup, &myPos);
          myPos != NULL;
          Q3Group_GetNextPosition(myGroup, &myPos))
    {
        myResult = Q3Group_GetPositionObject(myGroup, myPos, myLight);
        if (myResult == kQ3Failure)
            goto bail;
        myResult = Q3Light_SetState(myLight, myState);
        Q3Object_Dispose(myLight);          /*balance reference count of light*/
    }

    return(kQ3Success);

bail:
    return(kQ3Failure);
}
```

You can use the looping technique illustrated in Listing 10-2 to traverse ordered display groups as well, as shown in Listing 10-3. The function `MyToggleOrderedGroupLights` traverses an ordered display group and toggles any lights it finds. Notice that `MyToggleOrderedGroupLights` calls the `Q3Group_GetFirstPositionOfType` function to find the position of the first light in the group.

Listing 10-3 Accessing all the lights in an ordered display group

```
TQ3Status MyToggleOrderedGroupLights (TQ3GroupObject myGroup)
{
    TQ3GroupPosition        myPos;              /*a group position*/
    TQ3Object               myLight;            /*a light*/
    TQ3Boolean              myState;            /*a light state*/
    TQ3Status               myResult;           /*a result code*/

    for (Q3Group_GetFirstPositionOfType(myGroup, kQ3ShapeTypeLight, &myPos);
          myPos != NULL;
          Q3Group_GetNextPositionOfType(myGroup, kQ3ShapeTypeLight, &myPos))
    {
        myResult = Q3Group_GetPositionObject(myGroup, myPos, myLight);
        if (myResult == kQ3Failure)
            goto bail;
        myResult = Q3Light_GetState(myLight, &myState);
        myState = !myState;                 /*toggle the light state*/
        myResult = Q3Light_SetState(myLight, myState);
        Q3Object_Dispose(myLight);          /*balance reference count of light*/
    }

    return(kQ3Success);

bail:
    return(kQ3Failure);
}
```

It's also possible to find the position of the next object in an ordered display group by calling the `Q3Group_GetNextPosition` function. `Q3Group_GetNextPosition` is not, however, guaranteed to return a position of an object that is of the same type as the object immediately before it. If you use `Q3Group_GetNextPosition` to iterate through an ordered display group, you must therefore make sure not to step past the part of the list that contains objects of the type you're interested in. Listing 10-4 shows, in outline, how to call `Q3Group_GetNextPosition` to iterate safely through an object type in an ordered display group.

Listing 10-4 Accessing all the lights in an ordered display group using `Q3Group_GetNextPosition`

```
TQ3GroupPosition        myFirst;            /*group position of first light*/
TQ3GroupPosition        myLast;             /*group position of last light*/
TQ3Object               myLight;            /*a light*/
TQ3Status               myResult;           /*a result code*/

Q3Group_GetFirstPositionOfType(myGroup, kQ3ShapeTypeLight, &myFirst);
if (myFirst) {
    Q3Group_GetLastPositionOfType(myGroup, kQ3ShapeTypeLight, &myLast);
    do
    {
        myResult = Q3Group_GetPositionObject(myGroup, myFirst, myLight);
        if (myResult == kQ3Failure)
            goto bail;
        myResult = Q3Light_GetState(myLight, &myState);
        myState = !myState;                 /*toggle the light state*/
        myResult = Q3Light_SetState(myLight, myState);
        Q3Object_Dispose(myLight);          /*balance reference count of light*/
        Q3Group_GetNextPosition(myGroup, &myFirst);
    } while (myFirst != myLast);
}
```

Group Objects Reference

This section describes the QuickDraw 3D constants and routines that you can use to manage groups.

Constants

QuickDraw 3D provides constants that define group state values.

Group State Flags

QuickDraw 3D defines a set of **group state flags** for constructing a group state value. You pass a group state value to the `Q3DisplayGroup_SetState` function to set the state of a display group. The state value is a set of flags that determine how a group is traversed during rendering or picking, or when you want to compute a bounding box or sphere. Here are the group state flags:

```
typedef enum TQ3DisplayGroupStateMasks {
    kQ3DisplayGroupStateNone                            = 0,
    kQ3DisplayGroupStateMaskIsDrawn                     = 1 << 0,
    kQ3DisplayGroupStateMaskIsInline                    = 1 << 1,
    kQ3DisplayGroupStateMaskUseBoundingBox              = 1 << 2,
    kQ3DisplayGroupStateMaskUseBoundingSphere           = 1 << 3,
    kQ3DisplayGroupStateMaskIsPicked                    = 1 << 4,
    kQ3DisplayGroupStateMaskIsWritten                   = 1 << 5
} TQ3DisplayGroupStateMasks;
```

Constant descriptions

`kQ3DisplayGroupStateNone`
: No mask.

`kQ3DisplayGroupStateMaskIsDrawn`
: If this flag is set, the group and the objects it contains are drawn to a view during rendering or picking.

`kQ3DisplayGroupStateMaskIsInline`
: If this flag is set, the group is executed inline (that is, without pushing the graphics state onto a stack before group execution and popping it off after execution).

`kQ3DisplayGroupStateMaskUseBoundingBox`
: If this flag is set, the bounding box of a display group is used for rendering.

`kQ3DisplayGroupStateMaskUseBoundingSphere`
: If this flag is set, the bounding sphere of a display group is used for rendering.

`kQ3DisplayGroupStateMaskIsPicked`
: If this flag is set, the display group is eligible for inclusion in the hit list of a pick object.

`kQ3DisplayGroupStateMaskIsWritten`
: If this flag is set, the group and the objects it contains are written to a file object during writing.

IMPORTANT
By default, all group state flags are set except for the `kQ3DisplayGroupStateMaskIsInline` flag, which is clear. ▲

Group Objects Routines

This section describes routines you can use to create and manage groups and group positions.

Creating Groups

QuickDraw 3D provides a number of routines for creating group objects.

Q3Group_New

You can use the `Q3Group_New` function to create a new group.

```
TQ3GroupObject Q3Group_New (void);
```

DESCRIPTION

The `Q3Group_New` function returns, as its function result, a new group. The new group is initially empty. If an error occurs, `Q3Group_New` returns `NULL`.

ERRORS

`kQ3ErrorOutOfMemory`

Q3LightGroup_New

You can use the `Q3LightGroup_New` function to create a new light group.

```
TQ3GroupObject Q3LightGroup_New (void);
```

DESCRIPTION

The `Q3LightGroup_New` function returns, as its function result, a new light group. The new group is initially empty. If an error occurs, `Q3LightGroup_New` returns `NULL`.

Note
See the chapter "Light Objects" in this book for information on creating and manipulating individual lights. ◆

ERRORS

`kQ3ErrorOutOfMemory`

Q3DisplayGroup_New

You can use the `Q3DisplayGroup_New` function to create a new display group.

```
TQ3GroupObject Q3DisplayGroup_New (void);
```

DESCRIPTION

The `Q3DisplayGroup_New` function returns, as its function result, a new display group. The new group is initially empty. If an error occurs, `Q3DisplayGroup_New` returns `NULL`.

ERRORS

`kQ3ErrorOutOfMemory`

Q3InfoGroup_New

You can use the `Q3InfoGroup_New` function to create a new information group.

```
TQ3GroupObject Q3InfoGroup_New (void);
```

DESCRIPTION

The `Q3InfoGroup_New` function returns, as its function result, a new information group. The new group is initially empty. If an error occurs, `Q3InfoGroup_New` returns `NULL`.

ERRORS

`kQ3ErrorOutOfMemory`

Q3OrderedDisplayGroup_New

You can use the `Q3OrderedDisplayGroup_New` function to create a new ordered display group.

```
TQ3GroupObject Q3OrderedDisplayGroup_New (void);
```

DESCRIPTION

The `Q3OrderedDisplayGroup_New` function returns, as its function result, a new ordered display group. The new group is initially empty. If an error occurs, `Q3OrderedDisplayGroup_New` returns `NULL`.

ERRORS

`kQ3ErrorOutOfMemory`

Q3IOProxyDisplayGroup_New

You can use the `Q3IOProxyDisplayGroup_New` function to create a new I/O proxy display group.

```
TQ3GroupObject Q3IOProxyDisplayGroup_New (void);
```

DESCRIPTION

The `Q3IOProxyDisplayGroup_New` function returns, as its function result, a new I/O proxy display group. The new group is initially empty. If an error occurs, `Q3IOProxyDisplayGroup_New` returns `NULL`.

ERRORS

`kQ3ErrorOutOfMemory`

Managing Groups

QuickDraw 3D provides a number of general routines for managing group objects. Unless otherwise indicated, you can use these functions with groups of any type.

Q3Group_GetType

You can use the `Q3Group_GetType` function to determine the type of a group.

```
TQ3ObjectType Q3Group_GetType (TQ3GroupObject group);
```

`group` A group.

DESCRIPTION

The `Q3Group_GetType` function returns, as its function result, the type of the group specified by the `group` parameter. `Q3Group_GetType` returns one of these values:

```
kQ3GroupTypeDisplay
kQ3GroupTypeInfo
kQ3GroupTypeLight
```

If `Q3Group_GetType` cannot determine the type of a group or an error occurs, it returns `kQ3ObjectTypeInvalid`.

ERRORS

```
kQ3ErrorInvalidObject
kQ3ErrorNULLParameter
```

Q3Group_CountObjects

You can use the `Q3Group_CountObjects` function to determine how many objects a group contains.

```
TQ3Status Q3Group_CountObjects (
                    TQ3GroupObject group,
                    unsigned long *nObjects);
```

`group`	A group.
`nObjects`	On exit, a pointer to the number of objects in the specified group.

DESCRIPTION

The `Q3Group_CountObjects` function returns, in the `nObjects` parameter, the number of objects contained in the group specified by the `group` parameter. If that group contains other groups, each contained group is counted only once.

ERRORS

```
kQ3ErrorInvalidObject
kQ3ErrorNULLParameter
```

Q3Group_CountObjectsOfType

You can use the `Q3Group_CountObjectsOfType` function to determine how many objects of a particular type a group contains.

```
TQ3Status Q3Group_CountObjectsOfType (
                    TQ3GroupObject group,
                    TQ3ObjectType isType,
                    unsigned long *nObjects);
```

`group` A group.

`isType` An object type.

`nObjects` On exit, a pointer to the number of objects in the specified group that have the specified type.

DESCRIPTION

The `Q3Group_CountObjectsOfType` function returns, in the `nObjects` parameter, the number of objects contained in the group specified by the `group` parameter that have the object type specified by the `isType` parameter. The object type can be either a parent class (for example, `kQ3SharedType_Shape`) or a leaf class (for example, `EcGeometryType_Box`).

ERRORS

```
kQ3ErrorInvalidObject
kQ3ErrorNULLParameter
```

Q3Group_AddObject

You can use the `Q3Group_AddObject` function to add an object to a group.

```
TQ3GroupPosition Q3Group_AddObject (
                    TQ3GroupObject group,
                    TQ3Object object);
```

`group` A group.

`object` An object.

DESCRIPTION

The `Q3Group_AddObject` function inserts the object specified by the `object` parameter into the group specified by the `group` parameter. If `group` is a unordered group, the object is appended to the list of objects in the group. If `group` is an ordered group, the object is appended to the part of the list of objects in the group that are of the same type as `object`. `Q3Group_AddObject` returns the new position of the object in the group. If an error occurs as an object is inserted into the group, `Q3Group_AddObject` returns `NULL`.

ERRORS

```
kQ3ErrorInvalidObject
kQ3ErrorOutOfMemory
```

Q3Group_AddObjectBefore

You can use the `Q3Group_AddObjectBefore` function to add an object to a group, positioning it before a certain object already in the group.

```
TQ3GroupPosition Q3Group_AddObjectBefore (
                    TQ3GroupObject group,
                    TQ3GroupPosition position,
                    TQ3Object object);
```

`group`	A group.
`position`	A group position.
`object`	An object.

DESCRIPTION

The `Q3Group_AddObjectBefore` function inserts the object specified by the `object` parameter into the group specified by the `group` parameter, before the group position specified by the `position` parameter. `Q3Group_AddObjectBefore` returns, as its function result, the new position of the object in the group. If an error occurs during the insertion of the object into the group, `Q3Group_AddObjectBefore` returns `NULL`.

ERRORS

```
kQ3ErrorInvalidObject
kQ3ErrorInvalidPositionForGroup
kQ3ErrorOutOfMemory
```

Q3Group_AddObjectAfter

You can use the `Q3Group_AddObjectAfter` function to add an object to a group, positioning it after a certain object already in the group.

```
TQ3GroupPosition Q3Group_AddObjectAfter (
                    TQ3GroupObject group,
                    TQ3GroupPosition position,
                    TQ3Object object);
```

`group`	A group.
`position`	A group position.
`object`	An object.

DESCRIPTION

The `Q3Group_AddObjectAfter` function inserts the object specified by the `object` parameter into the group specified by the `group` parameter, after the group position specified by the `position` parameter. `Q3Group_AddObjectAfter` returns, as its function result, the new position of the object in the group. If an error occurs during the insertion of the object into the group, `Q3Group_AddObjectAfter` returns `NULL`.

ERRORS

```
kQ3ErrorInvalidObject
kQ3ErrorInvalidPositionForGroup
kQ3ErrorOutOfMemory
```

Q3Group_GetPositionObject

You can use the `Q3Group_GetPositionObject` function to get the object located at a certain position in a group.

```
TQ3Status Q3Group_GetPositionObject (
                    TQ3GroupObject group,
                    TQ3GroupPosition position,
                    TQ3Object *object);
```

`group` A group.

`position` A group position.

`object` On exit, a reference to a QuickDraw 3D object.

DESCRIPTION

The `Q3Group_GetPositionObject` function returns, in the `object` parameter, a reference to the object having the position specified by the `position` parameter in the group specified by the `group` parameter. The reference count of the returned object is incremented. If an error occurs when getting the object, `Q3Group_GetPositionObject` returns `NULL`.

ERRORS

```
kQ3ErrorInvalidObject
kQ3Error_InvalidPositionForGroup
kQ3Error_NULLParameter
```

Q3Group_SetPositionObject

You can use the `Q3Group_SetPositionObject` function to set the object located at a certain position in a group.

```
TQ3Status Q3Group_SetPositionObject (
                    TQ3GroupObject group,
                    TQ3GroupPosition position,
                    TQ3Object object);
```

group	A group.
position	A group position.
object	An object.

DESCRIPTION

The Q3Group_SetPositionObject function sets the object having the position specified by the position parameter in the group specified by the group parameter to the object specified by the object parameter. The object previously occupying that position is disposed of. The reference count of object is incremented.

Q3GroupPosition_SetObject returns, as its function result, either a pointer to the object installed in the specified position, or NULL if an error occurs.

ERRORS

```
kQ3ErrorInvalidObject
kQ3ErrorInvalidObjectForGroup
kQ3ErrorInvalidObjectForPosition
kQ3ErrorInvalidPositionForGroup
```

Q3Group_RemovePosition

You can use the Q3Group_RemovePosition function to remove an object from a group.

```
TQ3Object Q3Group_RemovePosition (
                    TQ3GroupObject group,
                    TQ3GroupPosition position);
```

group	A group.
position	A group position.

DESCRIPTION

The Q3Group_RemovePosition function removes the object having the group position specified by the position parameter from the group specified by the

group parameter. After you call Q3Group_RemovePosition, the position specified by the position parameter is invalid. Q3Group_RemovePosition returns, as its function result, the object removed from the group. If an error occurs when removing the object from the group, Q3Group_RemovePosition returns NULL.

ERRORS

```
kQ3ErrorInvalidObject
kQ3ErrorInvalidPositionForGroup
```

Q3Group_EmptyObjects

You can use the Q3Group_EmptyObjects function to remove all objects from a group.

```
TQ3Status Q3Group_EmptyObjects (TQ3GroupObject group);
```

`group` A group.

DESCRIPTION

The Q3Group_EmptyObjects function disposes of every object contained in the group specified by the group parameter, thereby effectively emptying the contents of the group. The group itself is not disposed of.

ERRORS

```
kQ3ErrorInvalidObject
```

Q3Group_EmptyObjectsOfType

You can use the Q3Group_EmptyObjectsOfType function to remove all objects of a particular type from a group.

```
TQ3Status Q3Group_EmptyObjectsOfType (
                    TQ3GroupObject group,
                    TQ3ObjectType isType);
```

`group` A group.

`isType` An object type.

DESCRIPTION

The `Q3Group_EmptyObjectsOfType` function disposes of every object contained in the group specified by the `group` parameter that has the type specified by the `isType` parameter.

ERRORS

`kQ3ErrorInvalidObject`

Managing Display Groups

QuickDraw 3D provides routines that you can use to manage display groups in general.

Q3DisplayGroup_GetType

You can use the `Q3DisplayGroup_GetType` function to determine the type of a display group.

```
TQ3ObjectType Q3DisplayGroup_GetType (TQ3GroupObject group);
```

`group` A group.

DESCRIPTION

The `Q3DisplayGroup_GetType` function returns, as its function result, the type of the display group specified by the `group` parameter. `Q3DisplayGroup_GetType` returns one of these values:

```
kQ3DisplayGroupTypeIOProxy
kQ3DisplayGroupTypeOrdered
```

If `Q3DisplayGroup_GetType` cannot determine the type of a group or an error occurs, it returns `kQ3ObjectTypeInvalid`.

ERRORS

```
kQ3ErrorInvalidObject
```

Q3DisplayGroup_GetState

You can use the `Q3DisplayGroup_GetState` function to get the current state of a display group.

```
TQ3Status Q3DisplayGroup_GetState (
                    TQ3GroupObject group,
                    TQ3DisplayGroupState *state);
```

`group`	A display group.
`state`	On exit, a pointer to the current state value for the specified display group.

DESCRIPTION

The `Q3DisplayGroup_GetState` function returns, in the `state` parameter, a pointer to a state value for the display group specified by the `group` parameter. The state value is a set of flags that determine how a display group is traversed during rendering or picking, or during computation of a bounding box or sphere. See "Group State Flags" (page 722) for a description of the flags currently defined for a group state value.

ERRORS

```
kQ3ErrorInvalidObject
kQ3ErrorNULLParameter
```

Q3DisplayGroup_SetState

You can use the `Q3DisplayGroup_SetState` function to set the state of a display group.

```
TQ3Status Q3DisplayGroup_SetState (
                    TQ3GroupObject group,
                    TQ3DisplayGroupState state);
```

`group` A display group.

`state` The desired state value for the specified display group.

DESCRIPTION

The `Q3DisplayGroup_SetState` function sets the state value of the display group specified by the `group` parameter to the value pointed to by the `state` parameter. See "Group State Flags" (page 722) for a description of the flags currently defined for a group state value.

ERRORS

`kQ3ErrorInvalidObject`

Q3DisplayGroup_Submit

You can use the `Q3DisplayGroup_Submit` function to submit a display group for drawing, picking, bounding, or writing.

```
TQ3Status Q3DisplayGroup_Submit (
                    TQ3GroupObject group,
                    TQ3ViewObject view);
```

`group` A group.

`view` A view.

DESCRIPTION

The `Q3DisplayGroup_Submit` function submits the display group specified by the `group` parameter for drawing, picking, bounding, or writing in the view specified by the `view` parameter.

SPECIAL CONSIDERATIONS

You should call this function only in a submitting loop.

ERRORS

```
kQ3ErrorInvalidObject
kQ3ErrorOutOfMemory
kQ3ErrorViewNotStarted
```

Getting Group Positions

QuickDraw 3D provides routines that you can use to move forward and backward through the objects in a group. You do so by finding the currently occupied group positions in the group and then determining which objects occupy those positions. This section describes the routines you can use to find the valid positions in a group.

Q3Group_GetFirstPosition

You can use the `Q3Group_GetFirstPosition` function to get the position of the first object in a group.

```
TQ3Status Q3Group_GetFirstPosition (
                    TQ3GroupObject group,
                    TQ3GroupPosition *position);
```

`group` A group.

`position` On exit, a group position.

DESCRIPTION

The `Q3Group_GetFirstPosition` function returns, in the `position` parameter, the position of the first object in the group specified by the `group` parameter.

ERRORS

```
kQ3ErrorInvalidObject
kQ3ErrorNULLParameter
```

Q3Group_GetFirstPositionOfType

You can use the `Q3Group_GetFirstPositionOfType` function to get the position of the first object of a particular type in a group.

```
TQ3Status Q3Group_GetFirstPositionOfType (
                    TQ3GroupObject group,
                    TQ3ObjectType isType,
                    TQ3GroupPosition *position);
```

`group`	A group.
`isType`	An object type.
`position`	On exit, a group position.

DESCRIPTION

The `Q3Group_GetFirstPositionOfType` function returns, in the `position` parameter, the position of the first object in the group specified by the `group` parameter that has the type specified by the `isType` parameter.

ERRORS

```
kQ3ErrorInvalidObject
kQ3ErrorNULLParameter
```

Q3Group_GetLastPosition

You can use the `Q3Group_GetLastPosition` function to get the position of the last object in a group.

```
TQ3Status Q3Group_GetLastPosition (
                    TQ3GroupObject group,
                    TQ3GroupPosition *position);
```

`group` A group.

`position` On exit, a group position.

DESCRIPTION

The `Q3Group_GetLastPosition` function returns, in the `position` parameter, the position of the last object in the group specified by the `group` parameter.

ERRORS

```
kQ3ErrorInvalidObject
kQ3ErrorNULLParameter
```

Q3Group_GetLastPositionOfType

You can use the `Q3Group_GetLastPositionOfType` function to get the position of the last object of a particular type in a group.

```
TQ3Status Q3Group_GetLastPositionOfType (
                    TQ3GroupObject group,
                    TQ3ObjectType isType,
                    TQ3GroupPosition *position);
```

`group` A group.

`isType` An object type.

`position` On exit, a group position.

DESCRIPTION

The `Q3Group_GetLastPositionOfType` function returns, in the `position` parameter, the position of the last object in the group specified by the `group` parameter that has the type specified by the `isType` parameter.

ERRORS

```
kQ3ErrorInvalidObject
kQ3ErrorNULLParameter
```

Q3Group_GetNextPosition

You can use the `Q3Group_GetNextPosition` function to get the position of the next object in a group.

```
TQ3Status Q3Group_GetNextPosition (
                    TQ3GroupObject group,
                    TQ3GroupPosition *position);
```

`group` A group.

`position` On entry, a pointer to a valid group position. On exit, a pointer to the position in the specified group of the object that immediately follows the object in that position.

DESCRIPTION

The `Q3Group_GetNextPosition` function returns, in the `position` parameter, the position in the group specified by the `group` parameter of the object that immediately follows the object having the position specified on entry in the `position` parameter. If the object specified on entry is the last object in the group, `Q3Group_GetNextPosition` returns the value `NULL` in the `position` parameter.

ERRORS

```
kQ3ErrorInvalidObject
kQ3ErrorInvalidPositionForGroup
kQ3ErrorNULLParameter
```

Q3Group_GetNextPositionOfType

You can use the `Q3Group_GetNextPositionOfType` function to get the position of the next object of a particular type in a group.

```
TQ3Status Q3Group_GetNextPositionOfType (
                    TQ3GroupObject group,
                    TQ3ObjectType isType,
                    TQ3GroupPosition *position);
```

`group` A group.

`isType` An object type.

`position` On entry, a pointer to a valid group position. On exit, a pointer to the position in the specified group of the next object that follows the object in that position and that has the specified type.

DESCRIPTION

The `Q3Group_GetNextPositionOfType` function returns, in the `position` parameter, the position in the group specified by the `group` parameter of the next object that follows the object having the position specified on entry in the `position` parameter and that has the type specified by the `isType` parameter. If the object specified on entry is the last object of that type in the group, `Q3Group_GetNextPositionOfType` returns the value `NULL` in the `position` parameter. Note that the type of the object in the position specified by the `position` parameter on entry to `Q3Group_GetNextPositionOfType` does not have to be the same as the type specified by the `isType` parameter.

ERRORS

```
kQ3ErrorInvalidObject
kQ3ErrorInvalidPositionForGroup
kQ3ErrorNULLParameter
```

Q3Group_GetPreviousPosition

You can use the `Q3Group_GetPreviousPosition` function to get the position of the previous object in a group.

```
TQ3Status Q3Group_GetPreviousPosition (
                    TQ3GroupObject group,
                    TQ3GroupPosition *position);
```

`group` A group.

`position` On entry, a pointer to a valid group position. On exit, a pointer to the position in the specified group of the object that immediately precedes the object in that position.

DESCRIPTION

The `Q3Group_GetPreviousPosition` function returns, in the `position` parameter, the position in the group specified by the `group` parameter of the object that immediately precedes the object having the position specified on entry in the `position` parameter. If the object specified on entry is the first object in the group, `Q3Group_GetPreviousPosition` returns the value `NULL` in the `position` parameter.

ERRORS

```
kQ3ErrorInvalidObject
kQ3ErrorInvalidPositionForGroup
kQ3ErrorNULLParameter
```

Q3Group_GetPreviousPositionOfType

You can use the `Q3Group_GetPreviousPositionOfType` function to get the position of the previous object of a particular type in a group.

```
TQ3Status Q3Group_GetPreviousPositionOfType (
                    TQ3GroupObject group,
                    TQ3ObjectType isType,
                    TQ3GroupPosition *position);
```

group	A group.
isType	An object type.
position	On entry, a pointer to a valid group position. On exit, a pointer to the position in the specified group of the next object that follows the object in that position and that has the specified type.

DESCRIPTION

The Q3Group_GetPreviousPositionOfType function returns, in the position parameter, the position in the group specified by the group parameter of the previous object that precedes the object having the position specified on entry in the position parameter and that has the type specified by the isType parameter. If the object specified on entry is the first object of that type in the group, Q3Group_GetNextPositionOfType returns the value NULL in the position parameter. Note that the type of the object in the position specified by the position parameter on entry to Q3Group_GetPreviousPositionOfType does not have to be the same as the type specified by the isType parameter.

ERRORS

```
kQ3ErrorInvalidObject
kQ3ErrorInvalidPositionForGroup
kQ3ErrorNULLParameter
```

Getting Object Positions

QuickDraw 3D provides routines that you can use to find instances of objects in groups.

Q3Group_GetFirstObjectPosition

You can use the Q3Group_GetFirstObjectPosition function to get the position of the first instance of an object in a group.

```
TQ3Status Q3Group_GetFirstObjectPosition (
                    TQ3GroupObject group,
                    TQ3Object object,
                    TQ3GroupPosition *position);
```

`group` A group.

`object` An object.

`position` On exit, a group position.

DESCRIPTION

The `Q3Group_GetFirstObjectPosition` function returns, in the `position` parameter, the position of the first instance in the group specified by the `group` parameter of the object specified by the `object` parameter.

ERRORS

```
kQ3ErrorInvalidObject
kQ3ErrorNULLParameter
```

Q3Group_GetLastObjectPosition

You can use the `Q3Group_GetLastObjectPosition` function to get the position of the last instance of an object in a group.

```
TQ3Status Q3Group_GetLastObjectPosition (
                    TQ3GroupObject group,
                    TQ3Object object,
                    TQ3GroupPosition *position);
```

`group` A group.

`object` An object.

`position` On exit, a group position.

DESCRIPTION

The `Q3Group_GetLastObjectPosition` function returns, in the `position` parameter, the position of the last instance in the group specified by the `group` parameter of the object specified by the `object` parameter.

ERRORS

```
kQ3ErrorInvalidObject
kQ3ErrorNULLParameter
```

Q3Group_GetNextObjectPosition

You can use the `Q3Group_GetNextObjectPosition` function to get the position of the next instance of an object in a group.

```
TQ3Status Q3Group_GetNextObjectPosition (
                    TQ3GroupObject group,
                    TQ3Object object,
                    TQ3GroupPosition *position);
```

`group`	A group.
`object`	An object.
`position`	On entry, a pointer to a valid group position. On exit, a pointer to the position in the specified group of the next instance of the specified object.

DESCRIPTION

The `Q3Group_GetNextObjectPosition` function returns, in the `position` parameter, the position of the next instance in the group specified by the `group` parameter of the object specified by the `object` parameter. If the position specified on entry is the last instance of that object in the group, `Q3Group_GetNextObjectPosition` returns the value `NULL` in the `position` parameter.

ERRORS

```
kQ3ErrorInvalidObject
kQ3ErrorInvalidPositionForGroup
kQ3ErrorNULLParameter
```

Q3Group_GetPreviousObjectPosition

You can use the `Q3Group_GetPreviousObjectPosition` function to get the position of the previous instance of an object in a group.

```
TQ3Status Q3Group_GetPreviousObjectPosition (
                    TQ3GroupObject group,
                    TQ3Object object,
                    TQ3GroupPosition *position);
```

`group`	A group.
`object`	An object.
`position`	On entry, a pointer to a valid group position. On exit, a pointer to the position in the specified group of the previous instance of the specified object.

DESCRIPTION

The `Q3Group_GetPreviousObjectPosition` function returns, in the `position` parameter, the position of the previous instance in the group specified by the `group` parameter of the object specified by the `object` parameter. If the position specified on entry is the first instance of that object in the group, `Q3Group_GetPreviousObjectPosition` returns the value `NULL` in the `position` parameter.

ERRORS

```
kQ3ErrorInvalidObject
kQ3ErrorInvalidPositionForGroup
kQ3ErrorNULLParameter
```

Extending Group Objects

QuickDraw 3D provides a programming interface by which you can add new group objects to the class `TQ3GroupObject`. This section first describes the `Q3XGroup_GetPositionPrivate` function, by which you can access the private data in your group object, and then lists the methods that may be called.

Q3XGroup_GetPositionPrivate

You can use the `Q3XGroup_GetPositionPrivate` function to return the private data stored in a group object.

```
void Q3XGroup_GetPositionPrivate (
                    TQ3GroupObject group,
                    TQ3GroupPosition position);
```

`group` A group object.

`position` A position in the group object.

DESCRIPTION

The `Q3XGroup_GetPositionPrivate` function returns the the private data stored at position `position` in group object `group`.

TQ3XGroupAcceptObjectMethod

The `TQ3XGroupAcceptObjectMethod` method reports whether a group will accept a particular object type.

```
TQ3Boolean (*TQ3XGroupAcceptObjectMethod) (
                    TQ3GroupObject group,
                    TQ3Object object);
```

`group`	A group object.
`object`	A new object to be placed in the group.

DESCRIPTION

The `TQ3XGroupAcceptObjectMethod` method returns `TQ3True` if the group object `group` will accept an object of type `object` and `TQ3False` otherwise.

TQ3XGroupAddObjectMethod

The `TQ3XGroupAddObjectMethod` method adds an object to a group.

```
TQ3GroupPosition (*TQ3XGroupAddObjectMethod) (
                    TQ3GroupObject group,
                    TQ3Object object);
```

`group`	A group object.
`object`	A new object to be placed in the group.

DESCRIPTION

The `TQ3XGroupAddObjectMethod` method adds the object `object` to the group `group`. It returns the position of the new object if successful and `NULL` otherwise.

TQ3XGroupAddObjectBeforeMethod

The `TQ3XGroupAddObjectBeforeMethod` method adds an object before a given position in a group.

```
TQ3GroupPosition (*TQ3XGroupAddObjectBeforeMethod) (
                    TQ3GroupObject group,
                    TQ3GroupPosition position,
                    TQ3Object object);
```

`group`	A group object.

`position`	A position in a group object.
`object`	A new object to be placed in the group.

DESCRIPTION

The `TQ3XGroupAddObjectBeforeMethod` method adds the object `object` to the group `group` before the position `position`. It returns the position of the new object if successful and `NULL` otherwise.

TQ3XGroupAddObjectAfterMethod

The `TQ3XGroupAddObjectAfterMethod` method adds an object after a given position in a group.

```
TQ3GroupPosition (*TQ3XGroupAddObjectAfterMethod) (
                    TQ3GroupObject group,
                    TQ3GroupPosition position,
                    TQ3Object object);
```

`group`	A group object.
`position`	A position in a group object.
`object`	A new object to be placed in the group.

DESCRIPTION

The `TQ3XGroupAddObjectAfterMethod` method adds the object `object` to the group `group` after the position `position`. It returns the position of the new object if successful and `NULL` otherwise.

TQ3XGroupSetPositionObjectMethod

The `TQ3XGroupSetPositionObjectMethod` method replaces an object in a group.

```
TQ3Status (*TQ3XGroupSetPositionObjectMethod) (
                    TQ3GroupObject group,
                    TQ3GroupPosition gPos,
                    TQ3Object obj);
```

`group` A group object.

`gPos` A position in a group object.

`obj` A new object to be placed in the group.

DESCRIPTION

The `TQ3XGroupSetPositionObjectMethod` method replaces the object currently at position `gPos` in the group `group` with the new object `obj`. It returns `kQ3Success` if successful and `kQ3Failure` if the given group position is not in the group.

TQ3XGroupRemovePositionMethod

The `TQ3XGroupRemovePositionMethod` method replaces an object in a group.

```
TQ3Object (*TQ3XGroupRemovePositionMethod) (
                    TQ3GroupObject group,
                    TQ3GroupPosition position);
```

`group` A group object.

`position` A position in a group object.

DESCRIPTION

The `TQ3XGroupRemovePositionMethod` method deletes the object currently at position `position` in the group `group`. It returns the deleted object if successful; otherwise it returns `NULL`.

TQ3XGroupGetFirstPositionOfTypeMethod

The `TQ3XGroupGetFirstPositionOfTypeMethod` method gets the position of the first object of a specified type in a group.

```
TQ3Status (*TQ3XGroupGetFirstPositionOfTypeMethod) (
                    TQ3GroupObject group,
                    TQ3ObjectType isType,
                    TQ3GroupPosition *gPos);
```

`group` A group object.

`isType` An object type.

`gPos` A position in a group object.

DESCRIPTION

The `TQ3XGroupGetFirstPositionOfTypeMethod` method returns in `gPos` the position of the first object of type `isType` in the group `group`. It returns `kQ3Success` if successful and `kQ3Failure` if there is no object of type `isType` in the group.

TQ3XGroupGetLastPositionOfTypeMethod

The `TQ3XGroupGetLastPositionOfTypeMethod` method gets the position of the last object of a specified type in a group.

```
TQ3Status (*TQ3XGroupGetLastPositionOfTypeMethod) (
                    TQ3GroupObject group,
                    TQ3ObjectType isType,
                    TQ3GroupPosition *gPos);
```

`group` A group object.

`isType` An object type.

`gPos` A position in a group object.

DESCRIPTION

The `TQ3XGroupGetLastPositionOfTypeMethod` method returns in `gPos` the position of the last object of type `isType` in the group `group`. It returns `kQ3Success` if successful and `kQ3Failure` if there is no object of type `isType` in the group.

TQ3XGroupGetNextPositionOfTypeMethod

The `TQ3XGroupGetNextPositionOfTypeMethod` method gets the position of the next object of a specified type in a group.

```
TQ3Status (*TQ3XGroupGetNextPositionOfTypeMethod) (
                    TQ3GroupObject group,
                    TQ3ObjectType isType,
                    TQ3GroupPosition *gPos);
```

`group`	A group object.
`isType`	An object type.
`gPos`	A position in a group object.

DESCRIPTION

The `TQ3XGroupGetNextPositionOfTypeMethod` method returns in `gPos` the position of the next object of type `isType` after `gPos` in the group `group`. On exit, `gPos` contains `NULL` if there is no succeeding object of type `isType`.

TQ3XGroupGetPrevPositionOfTypeMethod

The `TQ3XGroupGetPrevPositionOfTypeMethod` method gets the position of the previous object of a specified type in a group.

```
TQ3Status (*TQ3XGroupGetPrevPositionOfTypeMethod) (
                    TQ3GroupObject group,
                    TQ3ObjectType isType,
                    TQ3GroupPosition *gPos);
```

`group`	A group object.
`isType`	An object type.
`gPos`	A position in a group object.

DESCRIPTION

The `TQ3XGroupGetPrevPositionOfTypeMethod` method returns in `gPos` the position of the previous object of type `isType` before `gPos` in the group `group`. On exit, `gPos` contains `NULL` if there is no prior object of type `isType`.

TQ3XGroupCountObjectsOfTypeMethod

The `TQ3XGroupCountObjectsOfTypeMethod` method returns the number of objects of a specified type in a group.

```
TQ3Status (*TQ3XGroupCountObjectsOfTypeMethod) (
                    TQ3GroupObject group,
                    TQ3ObjectType isType,
                    unsigned long *nObjects);
```

`group`	A group object.
`isType`	An object type.
`nObjects`	The number of objects of the given type in the group.

DESCRIPTION

The `TQ3XGroupCountObjectsOfTypeMethod` method returns in `nObjects` the count of the number of objects of type `isType` in the group `group`.

TQ3XGroupEmptyObjectsOfTypeMethod

The `TQ3XGroupEmptyObjectsOfTypeMethod` method disposes of all the objects of a specified type in a group.

```
TQ3Status (*TQ3XGroupEmptyObjectsOfTypeMethod) (
                    TQ3GroupObject group,
                    TQ3ObjectType isType);
```

`group` A group object.

`isType` An object type.

DESCRIPTION

The `TQ3XGroupEmptyObjectsOfTypeMethod` method disposes of all the objects of type `isType` in the group `group`.

TQ3XGroupGetFirstObjectPositionMethod

The `TQ3XGroupGetFirstObjectPositionMethod` method gets the position of the first instance of an object in a group.

```
TQ3Status (*TQ3XGroupGetFirstObjectPositionMethod) (
                    TQ3GroupObject group,
                    TQ3Object object,
                    TQ3GroupPosition *gPos);
```

`group` A group object.

`object` An object.

`gPos` A position in a group object.

DESCRIPTION

The `TQ3XGroupGetFirstObjectPositionMethod` method returns in `gPos` the position of the first instance of object `object` in the group `group`. It returns `kQ3Success` if successful and `kQ3Failure` if there is no instance of object `object` in the group.

TQ3XGroupGetLastObjectPositionMethod

The `TQ3XGroupGetLastObjectPositionMethod` method gets the position of the last instance of an object in a group.

```
TQ3Status (*TQ3XGroupGetLastObjectPositionMethod) (
                    TQ3GroupObject group,
                    TQ3Object object,
                    TQ3GroupPosition *gPos);
```

`group` A group object.

`object` An object.

`gPos` A position in a group object.

DESCRIPTION

The `TQ3XGroupGetLastObjectPositionMethod` method returns in `gPos` the position of the last instance of object `object` in the group `group`. It returns `kQ3Success` if successful and `kQ3Failure` if there is no instance of object `object` in the group.

TQ3XGroupGetNextObjectPositionMethod

The `TQ3XGroupGetNextObjectPositionMethod` method gets the position of the next instance of an object in a group.

```
TQ3Status (*TQ3XGroupGetNextObjectPositionMethod) (
                    TQ3GroupObject group,
                    TQ3Object object,
                    TQ3GroupPosition *gPos);
```

`group` A group object.

`object` An object.

`gPos` A position in a group object.

DESCRIPTION

The `TQ3XGroupGetNextObjectPositionMethod` method returns in `gPos` the position of the next instance of object `object` after `gPos` in the group `group`. On exit, `gPos` contains `NULL` if there is no succeeding instance of object `object`.

TQ3XGroupGetPrevObjectPositionMethod

The `TQ3XGroupGetPrevObjectPositionMethod` method gets the position of the previous instance of an object in a group.

```
TQ3Status (*TQ3XGroupGetPrevObjectPositionMethod) (
                    TQ3GroupObject group,
                    TQ3Object object,
                    TQ3GroupPosition *gPos);
```

`group` A group object.

`object` An object.

`gPos` A position in a group object.

DESCRIPTION

The `TQ3XGroupGetPrevObjectPositionMethod` method returns in `gPos` the position of the previous instance of object `object` before `gPos` in the group `group`. On exit, `gPos` contains `NULL` if there is no prior instance of object `object`.

TQ3XMethodTypeGroupPositionSize

The `TQ3XMethodTypeGroupPositionSize` method gets the size of your group position private data.

```
unsigned long TQ3XMethodTypeGroupPositionSize;
```

DESCRIPTION

The `TQ3XMethodTypeGroupPositionSize` method returns the size in bytes of your group position private data.

TQ3XGroupPositionNewMethod

The `TQ3XGroupPositionNewMethod` method makes a new group position in a group object.

```
TQ3Status (*TQ3XGroupPositionNewMethod) (
                    void *gPos,
                    TQ3Object object,
                    const void *initData);
```

`gPos`	A position in a group object.
`object`	An object.
`initData`	Data with which to initialize the position.

DESCRIPTION

The `TQ3XGroupPositionNewMethod` method creates a new position `gPos` in the object `object`, initializing it with the data in `initData`.

TQ3XGroupPositionCopyMethod

The `TQ3XGroupPositionCopyMethod` method copies a group position in a group object.

```
TQ3Status (*TQ3XGroupPositionCopyMethod) (
                    void *srcGPos,
                    void *dstGPos);
```

`srcGPos`	The position to be copied from.
`srcGPos`	The position to be copied into.

DESCRIPTION

The `TQ3XGroupPositionCopyMethod` method copies position `srcGPos` in the group object into position `srcGPos`.

TQ3XGroupPositionDeleteMethod

The `TQ3XGroupPositionDeleteMethod` method deletes a group position in a group object.

```
TQ3Status (*TQ3XGroupPositionDeleteMethod) (void *gPos);
```

`gPos` The position to be deleted.

DESCRIPTION

The `TQ3XGroupPositionDeleteMethod` method deletes position `gPos` in the group object.

TQ3XGroupStartIterateMethod

The `TQ3XGroupStartIterateMethod` method helps draw a view by iteration. It is called once when drawing begins and returns the first object to be drawn.

```
TQ3Status (*TQ3XGroupStartIterateMethod) (
                    TQ3GroupObject group,
                    TQ3GroupPosition *iterator,
                    TQ3Object *object,
                    TQ3ViewObject view);
```

`group` A group object.

`iterator` An iteration position in the group.

`object` An object to be drawn.

`view` A view.

DESCRIPTION

The `TQ3XGroupStartIterateMethod` method finds the first object to be drawn in `view` and returns it in the `object` parameter. If the returned `object` value is `NULL`, then `GroupEndIterate` will not be called. The `iterator` parameter is uninitialized when `GroupStartIterate` is called.

EXAMPLE

```
TQ3Status GroupStartIterate(
    TQ3GroupObject      group,
    TQ3GroupPosition    *iterator,
    TQ3Object           *object,
    TQ3ViewObject       view)
{
    // initialize gPos and object
    *iterator   = NULL;
    *object     = NULL;

    // get position of first object in group
    if (Q3Group_GetFirstPosition(group, iterator) == kQ3Failure)
        return kQ3Failure;

    if (*iterator == NULL)
        return kQ3Success;

    // get first object in group
    if (Q3Group_GetPositionObject(group, *iterator, object)
        == kQ3Failure)
            return kQ3Failure;

    return kQ3Success;
}
```

TQ3XGroupEndIterateMethod

The `TQ3XGroupEndIterateMethod` method helps draw a view by iteration. It is called repeatedly while a group is traversed, returning the next object to be drawn each time.

```
TQ3Status (*TQ3XGroupEndIterateMethod) (
                    TQ3GroupObject group,
                    TQ3GroupPosition *iterator,
                    TQ3Object *object,
                    TQ3ViewObject view);
```

`group` A group object.

`iterator` An iteration position in the group.

`object` An object to be drawn.

`view` A view.

DESCRIPTION

The `TQ3XGroupEndIterateMethod` method is called repeatedly with the previous `object` and `iterator` values. It returns in `object` the next object to be drawn, or `NULL` when when there are no more objects to be drawn. It is your responsibility to dispose of `object`.

EXAMPLE

```
TQ3Status GroupEndIterate(
    TQ3GroupObject      group,
    TQ3GroupPosition    *iterator,
    TQ3Object           *object,
    TQ3ViewObject       view)
{
    if (*object != NULL) {
        // dispose previous object
        Q3Object_Dispose(*object);
        *object = NULL;
```

```
        // get position of next object in group
        if (Q3Group_GetNextPosition(group, iterator) == kQ3Failure)
            return kQ3Failure;

        if (*iterator == NULL)
            return kQ3Success;

        // get next object in group
        return Q3Group_GetPositionObject(group, *iterator, object);

    } else {
        *iterator = NULL;
        return kQ3Success;
        }
}
```

TQ3XGroupEndReadMethod

The `TQ3XGroupEndReadMethod` method is a cleanup method that is called when a group has been completely read.

```
TQ3Status (*TQ3XGroupEndReadMethod) (TQ3GroupObject group);
```

`group` A group object.

DESCRIPTION

The `TQ3XGroupEndReadMethod` method performs validation for the group `group` and cleans up any memory caches used for reading.

Group Errors

The following errors may be returned by group object routines. A list of general QuickDraw 3D errors is given in “QuickDraw 3D Errors, Warnings, and Notices” (page 87).

```
kQ3ErrorInvalidPositionForGroup
kQ3ErrorInvalidObjectForGroup
kQ3ErrorInvalidObjectForPosition
```

CHAPTER 11

Renderer Objects

This chapter describes renderer objects (or renderers) and the functions you can use to manipulate them. You use renderers to specify the various aspects of the kind of image you want to create. A single renderer is associated with a view, along with a list of lights, a camera, and other settings that affect the drawing of a model. QuickDraw 3D supplies several kinds of renderers, but you can easily add other kinds of renderers to support alternate drawing algorithms.

To use this chapter, you should already be familiar with the QuickDraw 3D class hierarchy, described in the chapter "QuickDraw 3D Objects" earlier in this book. For information about associating a renderer with a view, see the chapter "View Objects."

This chapter begins by describing renderer objects and their features. Then it shows how to create and manipulate renderers. The section "Renderer Objects Reference," beginning on page 771 provides a complete description of the routines you can use to create and manipulate renderer objects. The section "Renderer Methods," beginning on page 792 discusses methods that a custom renderer can or must support. The section "Draw Region Interface," beginning on page 817 describes the draw region interface, which lets custom renderers access the QuickDraw 3D frame buffer and communicate information about the configuration and state of the drawing context.

About Renderer Objects

A **renderer object** (or, more briefly, a **renderer**) is a type of QuickDraw 3D object that you can use to **render** a model—that is, to create an image from a view and a model. A renderer controls various aspects of the model and the resulting image, including:

- the kinds of geometric objects the renderer can draw without decomposing them into simpler objects
- the parts of objects to be drawn (for example, only the edges or filled faces)
- the types of lights that are available and the illumination model to be applied
- the types of shaders that are available and kinds of interpolation that can be performed

To render an image of a model, you first need to create an instance of a renderer object. To do that, you'll need to know which types of renderers are available. QuickDraw 3D provides routines that you can use to get information about the available renderers and their capabilities. Once you've decided which renderer you want to use, you then create an instance of that renderer and attach it to a view. You can do this in several, ways, by calling `Q3Renderer_NewFromType` and then `Q3View_SetRenderer`, or by calling the function `Q3View_SetRendererByType`.

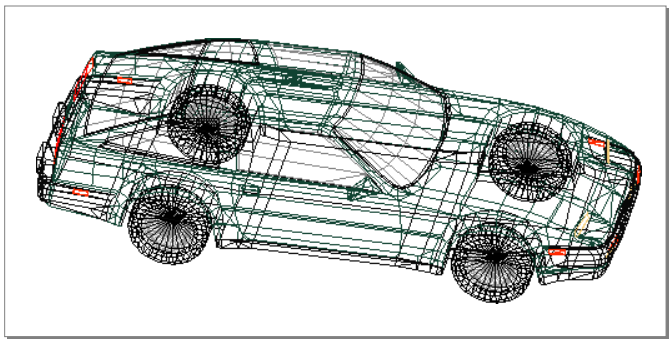
Types of Renderers

QuickDraw 3D currently supplies three types of renderers, a wireframe renderer, an interactive renderer, and a generic renderer. Only the wireframe and interactive renderers can actually draw images; the **generic renderer** is available for you to collect a view's state without actually rendering an image.

The **wireframe renderer** creates line drawings of models; it operates extremely quickly and with comparatively little memory. Figure 11-1 shows an example of a model drawn by QuickDraw 3D's wireframe renderer (see also Color Plate 1 at the beginning of this book).

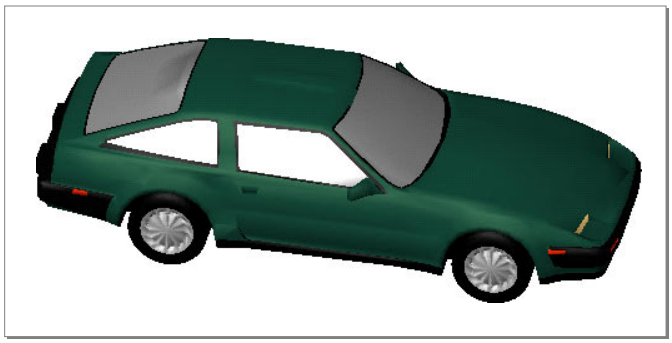
Because a wireframe image is simply a line drawing, there is no way to illuminate or shade surfaces. The wireframe renderer ignores the group of lights associated with a view and invokes none of the standard shaders supplied by QuickDraw 3D. Note, however, that the wireframe renderer does invoke any custom shaders you have associated with a view.

Figure 11-1 An image drawn by the wireframe renderer



The **interactive renderer** uses a fast and accurate depth-sorting algorithm for drawing solid, shaded surfaces as well as vectors. It is usually slower and requires more memory than the wireframe renderer. When the size of a model is reasonable and only very simple shadings are required, however, the interactive renderer is usually fast enough to provide acceptable interactive performance. The interactive renderer is also capable of rendering highly detailed, complex models with very realistic surface illumination and shading, but at the expense of time and memory. On machines with small amounts of memory, the interactive renderer may need to traverse a model in multiple passes to render the image completely. Figure 11-2 shows an image created by QuickDraw 3D's interactive renderer.

Figure 11-2 An image drawn by the interactive renderer



The interactive renderer is capable of driving either a software-only rasterizer or a hardware accelerator. In general, the interactive renderer uses a hardware accelerator if one is available, to provide maximum performance. You can, however, set the renderer preferences to indicate whether the interactive renderer should operate in software only or whether it should take advantage of a hardware accelerator. (See the "Using Renderer Objects" for details on setting a renderer's preferences.)

The interactive renderer supports all three available illumination shaders (Phong, Lambert, and null). Some rendering capabilities, however, are available only when the interactive renderer is using the hardware accelerator supplied by Apple Computer, Inc., including transparency, shadows, and constructive solid geometry (CSG).

Renderer Features

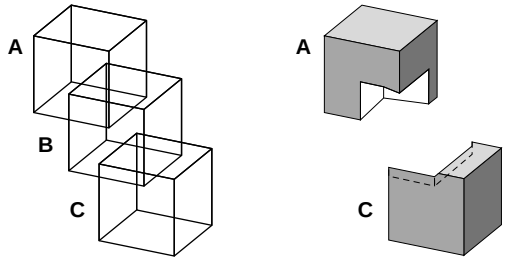
It's possible that the renderer allows the user to activate or deactivate certain renderer features in a modal dialog box displayed by the renderer. You can call the `Q3Renderer_HasModalConfigure` function to determine whether a particular renderer supports a modal settings dialog box. If it does, you can cause that dialog box to be displayed by calling the `Q3Renderer_ModalConfigure` function.

Constructive Solid Geometry

When the hardware accelerator provided by Apple Computer, Inc., is available, the interactive renderer can support **constructive solid geometry (CSG),** a method of modeling solid objects constructed from the union, intersection, or difference of other solid objects. For instance, you can define two cubes and then render the solid object that is the intersection of those two cubes. Similarly, you can define three cubes and render the solid object that is the union of two of them minus the third. For example, Figure 11-3 shows three cubes (A, B, and C) together with the result of using CSG to create the solid object defined by the function $(A \cup B) \cap \neg C$.

Note
In this chapter, CSG operations are described using standard set operators: the operation $A \cap B$ is the set of all points that are in both A and B (that is, the **intersection** of A and B); $A \cup B$ is the set of all points that are in either A or B (that is, the **union** of A and B); $\neg A$ is the set of all points that are not in A (that is, the **complement** of A). ◆

Figure 11-3 A constructed CSG object



The interactive renderer supports CSG operations on up to five objects in a model. You select the objects to operate on by assigning a **CSG object ID** to an object, an attribute of type `kQ3AttributeTypeConstructiveSolidGeometryID`. There are five CSG object IDs:

```
kQ3SolidGeometryObjA
kQ3SolidGeometryObjB
kQ3SolidGeometryObjC
kQ3SolidGeometryObjD
kQ3SolidGeometryObjE
```

You specify the CSG operations to perform by passing a **CSG equation** to the `Q3InteractiveRenderer_SetCSGEquation` function. A CSG equation is a 32-bit value that encodes which CSG operations are to be performed on which CSG objects. QuickDraw 3D provides constants for some common CSG operations:

```
typedef enum TQ3CSGEquation {
    kQ3CSGEquationAandB                         = (int) 0x88888888,
    kQ3CSGEquationAandnotB                      = 0x22222222,
    kQ3CSGEquationAanBonCad                     = 0x2F222F22,
    kQ3CSGEquationnotAandB                      = 0x44444444,
    kQ3CSGEquationnAaBorCanD                    = 0x74747474
} TQ3CSGEquation;
```

For instance, the constant `kQ3CSGEquationAandB` indicates that the interactive renderer should render only the intersecting portion of the objects with CSG object IDs `kQ3SolidGeometryObjA` and `kQ3SolidGeometryObjB`. There are 2^{32} CSG equations for the five possible CSG objects. You calculate a CSG equation for a particular configuration of objects *A*, *B*, *C*, *D*, and *E* by using Table 11-1.

You calculate a CSG equation by determining which of the rows in the table satisfy the desired CSG construction. Then you set the indicated bit positions in a 32-bit value and clear the remaining bit positions. For instance, the value 1 appears in both of the columns for objects *A* and *B* for bit positions 3, 7, 11, 15, 19, 23, 27, and 31. The CSG equation, then, for the operation $A \cap B$ is 10001000100010001000100010001000, or 0x88888888 (`kQ3CSGEquationAandB`). Similarly, the value 1 appears in the column for object *A* and the value 0 appears in the column for object *B* for bit positions 1, 5, 9, 13, 17, 21, 25, and 29. The CSG equation, then, for the operation $A \cap \neg B$ is 00100010001000100010001000100010, or 0x22222222 (`kQ3CSGEquationAandnotB`). Finally, the CSG equation used to construct the composite object shown in Figure 11-3 (page 767), drawn using the operation $(A \cup B) \cap \neg C$, is 00110010001100100011001000110010, or 0x32323232.

Table 11-1 Calculating CSG equations

E	D	C	B	A	Object	
4	3	2	1	0	Bit position	
0	0	0	0	0	0	LSB
0	0	0	0	1	1	
0	0	0	1	0	2	
0	0	0	1	1	3	
0	0	1	0	0	4	
0	0	1	0	1	5	
0	0	1	1	0	6	
0	0	1	1	1	7	

Table 11-1 Calculating CSG equations (continued)

E 4	D 3	C 2	B 1	A 0	Object Bit position	
0	1	0	0	0	8	
0	1	0	0	1	9	
0	1	0	1	0	10	
0	1	0	1	1	11	
0	1	1	0	0	12	
0	1	1	0	1	13	
0	1	1	1	0	14	
0	1	1	1	1	15	
1	0	0	0	0	16	
1	0	0	0	1	17	
1	0	0	1	0	18	
1	0	0	1	1	19	
1	0	1	0	0	20	
1	0	1	0	1	21	
1	0	1	1	0	22	
1	0	1	1	1	23	
1	1	0	0	0	24	
1	1	0	0	1	25	
1	1	0	1	0	26	
1	1	0	1	1	27	
1	1	1	0	0	28	
1	1	1	0	1	29	
1	1	1	1	0	30	
1	1	1	1	1	31	MSB

Transparency

Transparency is the ability of an object to transmit light, possibly permitting a viewer to see objects behind it. The interactive renderer allows you to draw objects with varying degrees of transparency. You specify how much light can pass through an object by setting its **transparency color.** A transparency color is an attribute of type `TQ3ColorRGB`, where the value (0, 0, 0) indicates complete transparency, and (1, 1, 1) indicates complete opacity. By default, objects are rendered opaque.

You specify an object's transparency color by adding an attribute of type `kQ3AttributeTypeTransparencyColor` to the object's attribute set. QuickDraw 3D

multiplies that transparency color by the object's diffuse color whenever a transparency color attribute is attached to the object.

Using Renderer Objects

A renderer is of type `TQ3RendererObject`, which is a type of shared object. You create an instance of a renderer by calling `Q3Renderer_New` or `Q3Renderer_NewFromType`. Once you've created a new renderer, you need to associate it with a particular view, for example by calling `Q3View_SetRenderer`.

You've already seen (in the section "Creating a View," beginning on page 67) how to create a renderer object and attach it to a view object. As indicated previously, you can ensure that you take advantage of any available hardware accelerator by using the interactive renderer, as follows:

```
myRenderer = Q3Renderer_NewFromType(kQ3RendererTypeInteractive);
```

To make the rendered images coherent, you should make the associated draw context double buffered (that is, you should set the `doubleBufferState` field of the draw context data structure to the value `kQ3True`). Some hardware rasterizer engines (such as the one supplied by Apple Computer, Inc.) can make coherent images without double buffering. This can provide a significant speed advantage, at the possible cost of some tearing. To take advantage of such hardware, you keep the draw context double buffered (to indicate that you want the images to be coherent) and call the function `Q3InteractiveRenderer_SetDoubleBufferBypass`, as follows:

```
Q3InteractiveRenderer_SetDoubleBufferBypass(myRenderer, kQ3True);
```

In the unlikely event that you want to use a particular rasterizer with the interactive renderer, you can set a preference with the code:

```
Q3InteractiveRenderer_SetPreferences(myRenderer, vendor, engine);
```

Values that define the available vendors and engines are described in "Vendor IDs" (page 771) and "Engine IDs" (page 771).

Renderer Objects Reference

This section describes the constants, data structures, and routines provided by QuickDraw 3D that you can use to create and manage renderers.

Constants

This section describes the constants that you can use to specify vendor and engine IDs, CSG object IDs, and CSG equations.

Vendor IDs

QuickDraw 3D provides constants that you can use to specify an ID for a renderer vendor.

```
#define kQAVendor_BestChoice                    (-1)
#define kQAVendor_Apple                         0
#define kQAVendor_ATI                           1
#define kQAVendor_Radius                        2
#define kQAVendor_Mentor                        3
#define kQAVendor_Matrox                        4
#define kQAVendor_Yarc                          5
#define kQAVendor_DiamondMM                     6
#define kQAVendor_3DLabs                        7
```

Constant description

`kQAVendor_BestChoice`
The best available choice. QuickDraw 3D selects the available drawing engine that produces the best output on the target device.

Engine IDs

QuickDraw 3D provides constants that you can use to specify an ID for the rendering engines supplied by Apple Computer, Inc.

```
#define kQAEngine_AppleHW                       (-1)
#define kQAEngine_AppleSW                       0
```

Constant descriptions

`kQAEngine_AppleHW` The rasterizer associated with the hardware accelerator supplied by Apple Computer, Inc.

`kQAEngine_AppleSW` The default software rasterizer supplied by Apple Computer, Inc.

CSG Object IDs

QuickDraw 3D provides constants that you can use to specify an ID for a CSG object. You assign a CSG object ID to an object by including an attribute of type `kQ3AttributeTypeConstructiveSolidGeometryID` in the object's attribute set. Currently, QuickDraw 3D supports up to five CSG objects per model.

```
#define kQ3SolidGeometryObjA                    0
#define kQ3SolidGeometryObjB                    1
#define kQ3SolidGeometryObjC                    2
#define kQ3SolidGeometryObjD                    3
#define kQ3SolidGeometryObjE                    4
```

Constant descriptions

`kQ3SolidGeometryObjA`
The CSG object *A*.

`kQ3SolidGeometryObjB`
The CSG object *B*.

`kQ3SolidGeometryObjC`
The CSG object *C*.

`kQ3SolidGeometryObjD`
The CSG object *D*.

`kQ3SolidGeometryObjE`
The CSG object *E*.

CSG Equations

QuickDraw 3D provides constants for some common CSG equations. See "Constructive Solid Geometry" (page 766) for more information on how CSG equations are determined.

```
typedef enum TQ3CSGEquation {
    kQ3CSGEquationAandB                         = (int) 0x88888888,
    kQ3CSGEquationAandnotB                      = 0x22222222,
    kQ3CSGEquationAanBonCad                     = 0x2F222F22,
    kQ3CSGEquationnotAandB                      = 0x44444444,
    kQ3CSGEquationnAaBorCanD                    = 0x74747474
} TQ3CSGEquation;
```

Constant descriptions

`kQ3CSGEquationAandB`
: $A \cap B$. The renderer draws the intersection of object *A* and object *B*.

`kQ3CSGEquationAandnotB`
: $A \cap \neg B$. The renderer draws the portion of object *A* that lies outside of object *B*.

`kQ3CSGEquationAanBonCad`
: $(A \cap \neg B) \cup (\neg C \cap D)$. The renderer draws the portion of object *A* that lies outside of object *B*, and the portion of object *D* that lies outside of object *C*.

`kQ3CSGEquationnotAandB`
: $\neg A \cap B$. The renderer draws the portion of object *B* that lies outside of object *A*.

`kQ3CSGEquationnAaBorCanD`
: $(\neg A \cap B) \cup (C \cap \neg D)$. The renderer draws the portion of object *B* that lies outside of object *A*, and the portion of object *C* that lies outside of object *D*.

Data Structures

This section describes the data structures that you use with renderer objects.

Dialog Anchor

You use the dialog anchor data structure with the `Q3Renderer_ModalConfigure` function, described on page 781. It has three forms, as described below.

With the Mac OS, the dialog anchor data structure has this form:

```
typedef struct TQ3DialogAnchor {
    TQ3MacOSDialogEventHandler clientEventHandler;
} TQ3DialogAnchor;
```

With Windows 32, the dialog anchor data structure has this form:

```
typedef struct TQ3DialogAnchor {
    HWND ownerWindow;
} TQ3DialogAnchor;
```

With other operating systems, the dialog anchor data structure has this form:

```
typedef struct TQ3DialogAnchor {
    char notUsed;                       /* place holder */
} TQ3DialogAnchor;
```

Renderer Object Routines

This section describes QuickDraw 3D routines that you can use to manage renderer objects.

Creating and Managing Renderers

QuickDraw 3D provides routines that you can use to create and manage instances of a renderer.

Q3Renderer_NewFromType

You can use the `Q3Renderer_NewFromType` function to create an instance of a certain type of renderer.

```
TQ3RendererObject Q3Renderer_NewFromType (
                    TQ3ObjectType rendererObjectType);
```

`rendererObjectType`
A value that specifies a renderer type.

DESCRIPTION

The `Q3Renderer_NewFromType` function returns, as its function result, a new renderer of the type specified by the `rendererObjectType` parameter. You can use these values to specify QuickDraw 3D's wireframe and interactive renderers:

```
kQ3RendererTypeWireFrame
kQ3RendererTypeInteractive
```

You can also pass the value `kQ3RendererTypeGeneric` to create a generic renderer. A generic renderer does not render any image, but you can use it to collect state information.

If `Q3Renderer_NewFromType` is not able to create an instance of the specified renderer type, it returns `NULL`.

SPECIAL CONSIDERATIONS

You should create a renderer object once and associate it with a view (by calling `Q3View_SetRenderer`); you should not create a new renderer object for each frame.

SEE ALSO

You can call the `Q3View_SetRendererByType` function to create a new renderer of a specified type and attach it to a view. See the chapter "View Objects" for complete information.

Q3Renderer_GetType

You can use the `Q3Renderer_GetType` function to get the type of a renderer.

```
TQ3ObjectType Q3Renderer_GetType (TQ3RendererObject renderer);
```

`renderer` A renderer.

DESCRIPTION

The `Q3Renderer_GetType` function returns, as its function result, the type of the renderer object specified by the `renderer` parameter. The types of renderer objects currently supported by QuickDraw 3D are defined by these constants:

```
kQ3RendererTypeWireFrame
kQ3RendererTypeGeneric
kQ3RendererTypeInteractive
```

If the specified renderer object is invalid or is not one of these types, `Q3Renderer_GetType` returns the value `kQ3ObjectTypeInvalid`.

Synchronizing and Flushing Renderers

You can use the `Q3View_Sync` function (page 885) to ensure that a drawing operation has finished. You should call `Q3View_Sync` only after you've called `Q3View_EndRendering`.

You can use the `Q3View_Flush` function (page 885) to flush any image buffers maintained internally by a renderer. You should call `Q3View_Flush` only between calls to the `Q3View_StartRendering` and `Q3View_EndRendering` functions.

Managing Interactive Renderers

QuickDraw 3D provides routines that you can use to manage interactive renderers.

Q3Renderer_IsInteractive

You can use the `Q3Renderer_IsInteractive` function to determine whether a renderer is interactive.

```
TQ3Boolean Q3Renderer_IsInteractive (TQ3RendererObject renderer);
```

`renderer` A renderer.

DESCRIPTION

The `Q3Renderer_IsInteractive` function returns `kQ3True` if `renderer` is an interactive renderer and `kQ3False` otherwise.

Q3InteractiveRenderer_GetPreferences

You can use the `Q3InteractiveRenderer_GetPreferences` function to get the current preference settings of an interactive renderer.

```
TQ3Status Q3InteractiveRenderer_GetPreferences (
                    TQ3RendererObject renderer,
                    long *vendorID,
                    long *engineID);
```

`renderer` An interactive renderer.

`vendorID` On exit, the vendor ID currently associated with the interactive renderer. See "Vendor IDs" (page 771) for the values that can be returned in this parameter.

`engineID` On exit, the engine ID currently associated with the interactive renderer. See "Engine IDs" (page 771) for the values that can be returned in this parameter.

DESCRIPTION

The `Q3InteractiveRenderer_GetPreferences` function returns, in the `vendorID` and `engineID` parameters, the vendor and engine IDs currently associated with the interactive renderer specified by the `renderer` parameter.

Q3InteractiveRenderer_SetPreferences

You can use the `Q3InteractiveRenderer_SetPreferences` function to set the preference settings of the interactive renderer.

```
TQ3Status Q3InteractiveRenderer_SetPreferences (
                    TQ3RendererObject renderer,
                    long vendorID,
                    long engineID);
```

`renderer` An interactive renderer.

`vendorID` A vendor ID. See "Vendor IDs" (page 771) for the values you can pass in this parameter.

`engineID` An engine ID. See "Engine IDs" (page 771) for the values you can pass in this parameter.

DESCRIPTION

The `Q3InteractiveRenderer_SetPreferences` function sets the default vendor and engine to be used by the interactive renderer specified by the `renderer` parameter to the values passed in the `vendorID` and `engineID` parameters.

Q3InteractiveRenderer_GetCSGEquation

You can use the `Q3InteractiveRenderer_GetCSGEquation` function to get the CSG equation used by the interactive renderer.

```
TQ3Status Q3InteractiveRenderer_GetCSGEquation (
                    TQ3RendererObject renderer,
                    TQ3CSGEquation *equation);
```

`renderer` An interactive renderer.

`equation` On exit, the CSG equation currently associated with the interactive renderer. See "CSG Equations" (page 773) for the values that can be returned in this parameter.

DESCRIPTION

The `Q3InteractiveRenderer_GetCSGEquation` function returns, in the `equation` parameter, the CSG equation currently associated with the interactive renderer specified by the `renderer` parameter.

Q3InteractiveRenderer_SetCSGEquation

You can use the `Q3InteractiveRenderer_SetCSGEquation` function to set the CSG equation used by the interactive renderer.

```
TQ3Status Q3InteractiveRenderer_SetCSGEquation (
                    TQ3RendererObject renderer,
                    TQ3CSGEquation equation);
```

`renderer` An interactive renderer.

`equation` A CSG equation. See "CSG Equations" (page 773) for the values you can pass in this parameter.

DESCRIPTION

The `Q3InteractiveRenderer_SetCSGEquation` function sets the CSG equation to be used by the interactive renderer specified by the `renderer` parameter to the equation specified by the `equation` parameter.

Q3InteractiveRenderer_GetDoubleBufferBypass

You can use the `Q3InteractiveRenderer_GetDoubleBufferBypass` function to get the current double buffer bypass state of the interactive renderer.

```
TQ3Status Q3InteractiveRenderer_GetDoubleBufferBypass (
                    TQ3RendererObject renderer,
                    TQ3Boolean *bypass);
```

`renderer` An interactive renderer.

`bypass` On exit, a Boolean value that indicates the current double buffer bypass state of the specified interactive renderer.

DESCRIPTION

The `Q3InteractiveRenderer_GetDoubleBufferBypass` function returns, in the `bypass` parameter, a Boolean value that indicates the current double buffer

bypass state of the interactive renderer specified by the `renderer` parameter. If `bypass` is `kQ3True`, double buffering is currently being bypassed.

Q3InteractiveRenderer_SetDoubleBufferBypass

You can use the `Q3InteractiveRenderer_SetDoubleBufferBypass` function to set the double buffer bypass state of the interactive renderer.

```
TQ3Status Q3InteractiveRenderer_SetDoubleBufferBypass (
                    TQ3RendererObject renderer,
                    TQ3Boolean bypass);
```

`renderer` An interactive renderer.

`bypass` A Boolean value that indicates the desired double buffer bypass state of the specified interactive renderer.

DESCRIPTION

The `Q3InteractiveRenderer_SetDoubleBufferBypass` function sets the state of double buffer bypassing for the interactive renderer specified by the `renderer` parameter to the Boolean value specified by the `bypass` parameter.

Managing Renderer Features

QuickDraw 3D provides routines that you can use to manage a renderer's features.

Q3Renderer_HasModalConfigure

You can use the `Q3Renderer_HasModalConfigure` function to determine whether a renderer can display a modal settings dialog box.

```
TQ3Boolean Q3Renderer_HasModalConfigure (TQ3RendererObject renderer);
```

`renderer` A renderer.

DESCRIPTION

The `Q3Renderer_HasModalConfigure` function returns, as its function result, a Boolean value that indicates whether the renderer specified by the `renderer` parameter can display a modal settings dialog box (`kQ3True`) or not (`kQ3False`).

Q3Renderer_ModalConfigure

You can use the `Q3Renderer_ModalConfigure` function to pop up a modal dialog box used to configure the renderer's settings.

```
TQ3Status Q3Renderer_ModalConfigure (
                    TQ3RendererObject renderer,
                    TQ3DialogAnchor dialogAnchor,
                    TQ3Boolean *canceled);
```

`renderer`	A renderer.
`dialogAnchor`	A dialog anchor data structure (see page 774).
`canceled`	Returned value indicating whether the request has been canceled (`kQ3True`) or not (`kQ3False`).

DESCRIPTION

The `Q3Renderer_ModalConfigure` function displays a modal settings dialog box to configure the settings for the renderer specified by the `renderer` parameter. The `canceled` parameter is a Boolean that returns `kQ3True` if the request has been canceled, `kQ3False` otherwise.

The `Q3Renderer_ModalConfigure` function returns `kQ3Failure` if an error occurred.

Q3Renderer_GetConfigurationData

An application can use the `Q3Renderer_GetConfigurationData` function to access private renderer configuration data, which it can then save in a preference file or style template. The application should tag this data with the renderer's object name.

```
TQ3Status Q3Renderer_GetConfigurationData (
                    TQ3RendererObject    renderer,
                    unsigned char        *dataBuffer,
                    unsigned long        bufferSize,
                    unsigned long         *actualDataSize);
```

`renderer` A renderer object.

`dataBuffer` A pointer to a data buffer.

`bufferSize` The actual size in bytes of the memory block pointed to by `dataBuffer`.

`actualDataSize`
On return, the actual number of bytes written to the buffer. If you set `dataBuffer` to `null`, `actualDataSize` returns the number of bytes that will be required to store the configuration data .

DESCRIPTION

The `Q3Renderer_GetConfigurationData` function stores private configuration data for the renderer object designated by `renderer` in a buffer pointed to by the `dataBuffer` parameter, and returns in `actualDataSize` the number of bytes written. If you set `dataBuffer` to `null`, `Q3Renderer_GetConfigurationData` will return in `actualDataSize` the buffer size required, without writing data out.

Q3Renderer_SetConfigurationData

You can use the `Q3Renderer_SetConfigurationData` function to set a renderer to a configuration state previously accessed by `Q3Renderer_GetConfigurationData`.

```
TQ3Status Q3Renderer_SetConfigurationData (
                    TQ3RendererObject    renderer,
                    unsigned char        *dataBuffer,
                    unsigned long         bufferSize);
```

`renderer` A renderer object.

`dataBuffer` A pointer to a data buffer.

`bufferSize` The actual size in bytes of the memory block pointed to by `dataBuffer`.

DESCRIPTION

The `Q3Renderer_SetConfigurationData` function sets the renderer designated by `renderer` to the configuration state determined by the private data structure pointed to by `dataBuffer`. The `bufferSize` parameter indicates the size of the buffer.

Q3RendererClass_GetNickNameString

You can use the `Q3RendererClass_GetNickNameString` function to to get a renderer's name string. The name string can then be used to provide user selections, for example in a menu.

```
TQ3Status Q3RendererClass_GetNickNameString(
                    TQ3ObjectType               rendererClassType,
                    TQ3ObjectClassNameString    rendererClassString);
```

`rendererClassType`
A renderer object type.

`rendererClassString`
A string containing the renderer's name.

DESCRIPTION

The `Q3RendererClass_GetNickNameString` function returns, in the `rendererClassString` parameter, the user-identifiable name of a renderer.

The renderer is responsible for storing the name in a localizable format—for example as a resource. If `Q3RendererClass_GetNickNameString` returns `NULL` in `rendererClassString`, then the caller may choose to use the renderer's class name instead. Applications should always try to get the name string before using the class name, because the class name is not localizable.

Managing RAVE Features

The QuickDraw 3D Renderer Acceleration Virtual Engine (RAVE) is the part of the Mac OS that controls 3D drawing engines, also called *3D drivers.* It is used internally by QuickDraw 3D. A drawing engine is software that supports the low-level rasterization operations required for interactive 3D rendering. To achieve interactive performance, a drawing engine is often associated with some hardware device designed specifically to accelerate 3D rasterization.

For most 3D drawing and interaction, you should use the high-level application programming interfaces described in this book. In some cases, however, you may need to use the low-level services provided by RAVE; for example,

- if you are writing a specialized application (such as a game-development framework) that needs to take advantage of any available 3D acceleration hardware.
- if you are writing interactive software (such as a game or other entertainment software) that requires the extremely fast 3D rendering that can be achieved with a very low-level, lightweight graphics library.
- if you are developing 3D acceleration hardware or software that is to be accessed by any applications rendering 3D images.

This section describes four QuickDraw 3D routines that provide you with limited access to RAVE. Two of them get and set the RAVE texture filter; the other two get and set RAVE context hints.

The texture mapping filter mode of the drawing engine determines how the engine performs texture mapping. The default value for a drawing engine that supports texture mapping is `kQATextureFilter_Fast`. The texture mapping filter state variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_Texture` feature.

You specify an engine's texture filter by assigning a value to its `kQATag_TextureFilter` state variable. The default value of this variable for a drawing engine that supports texture mapping is `kQATextureFilter_Fast`.

```
#define kQATextureFilter_Fast                       0
#define kQATextureFilter_Mid                        1
#define kQATextureFilter_Best                       2
```

Constant descriptions

`kQATextureFilter_Fast`
The drawing engine performs whatever level of texture

filtering it can do with no speed penalty. This often means that no texture filtering is performed.

`kQATextureFilter_Mid`
The drawing engine performs a medium level of texture filtering. You should use this texture mapping filter mode when you want to perform texture mapping interactively.

`kQATextureFilter_Best`
The drawing engine performs the highest level of texture filtering it can. This mode may be unsuitable for interactive rendering.

Q3InteractiveRenderer_GetRAVETextureFilter

You can use the `Q3InteractiveRenderer_GetRAVETextureFilter` function to access the current RAVE `kQATag_TextureFilter` state variable described above.

```
TQ3Status Q3InteractiveRenderer_GetRAVETextureFilter (
                    TQ3RendererObject    renderer,
                    unsigned long         *RAVEtextureFilterValue);
```

`renderer` A renderer object.

`RAVEtextureFilterValue`
The value of the renderer's `kQATag_TextureFilter` state variable.

DESCRIPTION

The `Q3InteractiveRenderer_GetRAVETextureFilter` function returns, in the `RAVEtextureFilterValue` parameter, the current value of the renderer's `kQATag_TextureFilter` state variable.

Q3InteractiveRenderer_SetRAVETextureFilter

You can use the `Q3InteractiveRenderer_SetRAVETextureFilter` function to set the current RAVE `kQATag_TextureFilter` state variable.

```
TQ3Status Q3InteractiveRenderer_SetRAVETextureFilter (
                    TQ3RendererObject    renderer,
                    unsigned long         RAVEtextureFilterValue);
```

`renderer` A renderer object.

`RAVEtextureFilterValue`
The value of the renderer's `kQATag_TextureFilter` state variable.

DESCRIPTION

The `Q3InteractiveRenderer_SetRAVETextureFilter` function sets the value in the `RAVEtextureFilterValue` parameter to be the current value of the renderer's `kQATag_TextureFilter` state variable. For a list of possible values, see page 784.

Although a drawing engine may be capable of supporting more than one device, it cannot divide a raster across multiple devices. Instead, every drawing command sent to a drawing engine must be destined for a single device. QuickDraw 3D RAVE guarantees this by requiring a calling application to specify a draw context as a parameter for every drawing command. A **draw context** is a structure (of type `TQADrawContext`) that maintains state information and other data associated with a particular drawing engine and device.

As mentioned at the end of the previous section, you need to create several draw contexts if you want to draw into a window that spans several devices. Similarly, you need to create several draw contexts if you want to draw into several different windows on the same device. Each draw context maintains its own state information image buffers and is unaffected by any functions that operate on another draw context.

The state information associated with a draw context is maintained using a large number of **state variables.** For example, the background color of a draw context is specified by four state variables, designated by the four identifiers (or **tags**) `kQATag_ColorBG_a`, `kQATag_ColorBG_r`, `kQATag_ColorBG_g`, and `kQATag_ColorBG_b`.

Q3InteractiveRenderer_GetRAVEContextHints

You can use the `Q3InteractiveRenderer_GetRAVEContextHints` function to get the RAVE draw context hints for a specific renderer.

```
TQ3Status Q3InteractiveRenderer_GetRAVEContextHints (
                    TQ3RendererObject    renderer,
                    unsigned long        *RAVEContextHints);
```

`renderer` A renderer.

`RAVEContextHints`
A set of RAVE draw context hints.

DESCRIPTION

The `Q3InteractiveRenderer_GetRAVEContextHints` function returns, in the `RAVEContextHints` parameter, the stored context hints for `renderer`.

Q3InteractiveRenderer_SetRAVEContextHints

You can use the `Q3InteractiveRenderer_SetRAVEContextHints` function to set the RAVE draw context hints for a specific renderer.

```
TQ3Status Q3InteractiveRenderer_SetRAVEContextHints (
                    TQ3RendererObject    renderer,
                    unsigned long        RAVEContextHints);
```

`renderer` A renderer.

`RAVEContextHints`
A set of RAVE draw context hints.

DESCRIPTION

The `Q3InteractiveRenderer_SetRAVEContextHints` function sets the draw context hints for `renderer` to the value in the `RAVEContextHints` parameter.

Using Renderer Attribute Set Tools

QuickDraw 3D supports two tools, which you can call only from a renderer plug-in module, that provide fast access to geometry attribute sets.

Q3XAttributeSet_GetPointer

You can use the `Q3XAttributeSet_GetPointer` function to obtain a pointer to QuickDraw 3D's internal data structure for elements and attributes in an attribute set.

```
void *Q3XAttributeSet_GetPointer (
                    TQ3AttributeSet     attributeSet,
                    TQ3AttributeType    attributeType);
```

`attributeSet` An attribute set.

`attributeType` An attribute type.

return value Pointer to an attribute set data structure.

DESCRIPTION

The `Q3XAttributeSet_GetPointer` function returns a pointer to internal data structure for elements and attributes in the attribute set designated by `attributeSet`, of the type designated by `attributeType`. It returns null if no such attribute set data structure exists. `Q3XAttributeSet_GetPointer` uses the same internal data structure as `Q3AttributeSet_Add`, described on page 530.

Q3XAttributeSet_GetMask

A renderer can use the `Q3XAttributeSet_GetMask` function to obtain a set of masks for the internal data structure returned by `Q3XAttributeSet_GetPointer`.

```
TQ3XAttributeMask Q3XAttributeSet_GetMask (
                    TQ3AttributeSet    attributeSet);

typedef unsigned long TQ3XAttributeMask;
```

`attributeSet` An attribute set.

return value A set of masks of type `TQ3XAttributeMask`.

DESCRIPTION

To determine which attributes are present in an attribute set, a renderer can obtain a set of the masks by calling `Q3XAttributeSet_GetMask`. It returns the masks for the attribute set designated by `attributeSet` as an `unsigned long` value.

The returned mask values are as follows:

```
#define kQ3XAttributeMaskNone 0L

#define kQ3XAttributeMaskSurfaceUV (1 << (kQ3AttributeTypeSurfaceUV - 1))

#define kQ3XAttributeMaskShadingUV (1 << (kQ3AttributeTypeShadingUV - 1))

#define kQ3XAttributeMaskNormal (1 << (kQ3AttributeTypeNormal - 1))

#define kQ3XAttributeMaskAmbientCoefficient
    (1 << (kQ3AttributeTypeAmbientCoefficient - 1))

#define kQ3XAttributeMaskDiffuseColor
    (1 << (kQ3AttributeTypeDiffuseColor - 1))

#define kQ3XAttributeMaskSpecularColor
    (1 << (kQ3AttributeTypeSpecularColor - 1))

#define kQ3XAttributeMaskSpecularControl
    (1 << (kQ3AttributeTypeSpecularControl - 1))

#define kQ3XAttributeMaskTransparencyColor
    (1 << (kQ3AttributeTypeTransparencyColor - 1))

#define kQ3XAttributeMaskSurfaceTangent
    (1 << (kQ3AttributeTypeSurfaceTangent - 1))

#define kQ3XAttributeMaskHighlightState
    (1 << (kQ3AttributeTypeHighlightState - 1))

#define kQ3XAttributeMaskSurfaceShader
    (1 << (kQ3AttributeTypeSurfaceShader - 1))

#define kQ3XAttributeMaskCustomAttribute 0x80000000
```

```
#define kQ3XAttributeMaskAll
    (  kQ3XAttributeMaskSurfaceUV
    |  kQ3XAttributeMaskShadingUV
    |  kQ3XAttributeMaskNormal
    |  kQ3XAttributeMaskAmbientCoefficient
    |  kQ3XAttributeMaskDiffuseColor
    |  kQ3XAttributeMaskSpecularColor
    |  kQ3XAttributeMaskSpecularControl
    |  kQ3XAttributeMaskTransparencyColor
    |  kQ3XAttributeMaskSurfaceTangent
    |  kQ3XAttributeMaskHighlightState
    |  kQ3XAttributeMaskSurfaceShader
    |  kQ3XAttributeMaskCustomAttribute )

#define kQ3XAttributeMaskInherited
    (  kQ3XAttributeMaskSurfaceUV
    |  kQ3XAttributeMaskShadingUV
    |  kQ3XAttributeMaskNormal
    |  kQ3XAttributeMaskAmbientCoefficient
    |  kQ3XAttributeMaskDiffuseColor
    |  kQ3XAttributeMaskSpecularColor
    |  kQ3XAttributeMaskSpecularControl
    |  kQ3XAttributeMaskTransparencyColor
    |  kQ3XAttributeMaskSurfaceTangent
    |  kQ3XAttributeMaskHighlightState
    |  kQ3XAttributeMaskSurfaceShader
    |  kQ3XAttributeMaskCustomAttribute )

#define kQ3XAttributeMaskInterpolated
    (  kQ3XAttributeMaskSurfaceUV
    |  kQ3XAttributeMaskShadingUV
    |  kQ3XAttributeMaskNormal
    |  kQ3XAttributeMaskAmbientCoefficient
    |  kQ3XAttributeMaskDiffuseColor
    |  kQ3XAttributeMaskSpecularColor
    |  kQ3XAttributeMaskSpecularControl
    |  kQ3XAttributeMaskTransparencyColor
    |  kQ3XAttributeMaskSurfaceTangent )
```

Using Renderer View Tools

QuickDraw 3D supports two tools, which you can call only from a renderer plug-in module, that report rendering progress.

Q3XView_IdleProgress

The `Q3XView_IdleProgress` function can be called by a renderer to call the user idle method and provide progress information. The user must have supplied an `idleProgress` method using `Q3XView_SetIdleProgressMethod`; otherwise, the generic idle method `Q3View_SetIdleMethod` will be called with no progress data.

```
TQ3Status Q3XView_IdleProgress (
                    TQ3ViewObject    view,
                    unsigned long    current,
                    unsigned long    completed );
```

`view` A view object.

`current` A progress value in the range 0..*n*–1, where *n* is the value of `completed`.

`completed` The maximum progress number.

DESCRIPTION

The `Q3XView_IdleProgress` function passes, in the `current` parameter, a value that represents rendering progress from 0 to the value of `completed`. It returns `kQ3Failure` if rendering is cancelled.

SEE ALSO

You can use view idle methods to interrupt long renderings. While running the idler callback, the application can check for a Command-period key event or a mouse click on a Cancel button to see if the user wants to interrupt rendering. View idle methods are discussed in Chapter 13, "Application-Defined Routines," beginning on page 909.

Q3XView_EndFrame

An asynchronous renderer calls the `Q3XView_EndFrame` function when it completes a frame.

```
TQ3Status Q3XView_EndFrame (TQ3ViewObject view);
```

`view` A view object.

DESCRIPTION

The `Q3XView_EndFrame` function tells the view object `view` that an asynchronous renderer has finished rendering a frame.

The `Q3XView_EndFrame` function differs from `Q3View_Sync` in that notification of frame completion takes place in the opposite direction. With `Q3View_Sync`, the application asks a renderer to finish rendering a frame and blocks until the frame is complete. With `Q3XView_EndFrame`, the renderer tells the application that it has completed a frame.

If `Q3View_Sync` is called before `Q3XView_EndFrame`, `Q3XView_EndFrame` will never be called. If `Q3View_Sync` is called after `Q3XView_EndFrame`, `Q3XView_EndFrame` will return immediately because the frame has already been completed.

`Q3View_Sync` is described in "Q3View_Sync" (page 885).

Application-Defined Routines

Among the functions that you might need to define when working with renderer objects is an event filter function to handle events that occur while a movable modal dialog box is displayed.

Renderer Methods

This section describes methods that a renderer can or must support.

IMPORTANT
Some of the methods described here are required, as noted below. ▲

The renderer support methods include **update methods** and **submit methods.** An update method is called whenever the state has changed, before invoking a submit method in a renderer. A renderer will have many update methods that do little more than copy data or pointers into the renderer's private state.

Updates are not called in any particular order and therefore cross-state dependencies should not be resolved until a submit method is invoked.For example, every view has an attribute state and a shader state, each of which may contain a `SurfaceShader` object. The attribute `SurfaceShader` object (if not `null`) overrides the shader object, because it is inherited deeper in a geometry. However, a renderer should not depend on these being called in any particular order. It should not keep one `surfaceShader` state variable and perform this inheritance in the update methods, because the behavior may vary.

Note
Some exceptions apply. In particular, matrix updates are called in a specific order based on matrix dependencies. For example, `LocalToWorld` is always called before `LocalToWorldInverse`, as explained on page 805. ◆

Update methods update only those items which have changed, and only those objects supported by a renderer are updated. In addition, updates are accumulated until a geometry, light, or camera is encountered; therefore data submitted to the view may never reach a renderer if it never applies to a geometry, light, or camera.

If the renderer supports the `RendererPush` and `RendererPop` methods, it must maintain its own state stack; updates are not called for changed data when the view stack is popped. See "Push and Pop Methods," beginning on page 813 for more information.

The following renderer submit functionality is discussed in "Submit Method," beginning on page 794:

```
TQ3XRendererSubmitGeometryMethod
```

The following renderer configuration functionalities are discussed in "Configuration Methods," beginning on page 796:

```
kQ3XMethodTypeRendererIsInteractive
TQ3XRendererModalConfigureMethod
TQ3XRendererGetNickNameStringMethod
TQ3XRendererGetConfigurationDataMethod
TQ3XRendererSetConfigurationDataMethod
```

The following renderer update functionalities are discussed in "Update Methods," beginning on page 801:

```
TQ3XRendererUpdateStyleMethod
TQ3XRendererUpdateAttributeMethod
TQ3XRendererUpdateShaderMethod
TQ3XRendererUpdateMatrixMethod
```

The following renderer drawing state methods are discussed in "Drawing State Methods," beginning on page 807:

```
TQ3XRendererStartFrameMethod
TQ3XRendererStartPassMethod
TQ3XRendererFlushFrameMethod
TQ3XRendererEndPassMethod
TQ3XRendererEndFrameMethod
TQ3XRendererCancelMethod
```

The following state stack functions are discussed in "Push and Pop Methods," beginning on page 813:

```
TQ3XRendererPushMethod
TQ3XRendererPopMethod
```

The renderer support method `TQ3XRendererIsBoundingBoxVisibleMethod`, used to cull group and geometric objects, is described in "Renderer Cull Method," beginning on page 816.

Submit Method

This section describes the renderer geometry submit method.

TQ3XRendererSubmitGeometryMethod

The `TQ3XRendererSubmitGeometryMethod` renderer support functionality is required.

```
#define kQ3XMethodTypeRendererSubmitGeometryMetaHandler
                    Q3_METHOD_TYPE ('r','d','g','m')

typedef TQ3XFunctionPointer
                    (*TQ3XRendererSubmitGeometryMetaHandlerMethod)(
                    TQ3ObjectType    geometryType)

typedef TQ3Status (*TQ3XRendererSubmitGeometryMethod)(
                    TQ3ViewObject        view,
                    void                 *rendererPrivate,
                    TQ3GeometryObject    geometry,
                    const void           *publicData);
```

`geometryType` A geometric object type (see "About Geometric Objects," beginning on page 237).

`view` The current view being rendered to.

`rendererPrivate`
A pointer to structure of size `instanceSize`, passed into `Q3ObjectHierarchy_RegisterClass`, and initialized in your `kQ3MethodTypeObjectNew` method.

`Objectgeometry`
The geometry for which this call was registered. It must be a geometry object containing `publicData`, or `NULL` if this call was made in immediate mode.

`publicData` A pointer to the public data structure associated with `Objectgeometry`. This pointer is passed into `Submit` calls, so the data may become invalid after `TQ3XRendererSubmitGeometryMethod` exits. If `Objectgeometry` is not `null`, you may call `Q3Shared_GetReference` and save the `publicData` pointer to preserve the information. In this case, call `Q3Object_Dispose` on the geometry when you are through.

DESCRIPTION

The method type `kQ3XMethodTypeRendererSubmitGeometryMetaHandler` returns a function pointer of type `TQ3XRendererSubmitGeometryMetaHandlerMethod`. This function enables a geometry of type `geometryType` and returns methods of type `TQ3XRendererSubmitGeometryMethod`.

This renderer functionality is required, and it must support the following geometric object types:

```
kQ3GeometryTypeTriangle
kQ3GeometryTypeLine
kQ3GeometryTypePoint
kQ3GeometryTypePixmapMarker
```

Configuration Methods

This section describes renderer configuration methods, all of which are optional.

Q3XMethodTypeRendererIsInteractive

The `Q3XMethodTypeRendererIsInteractive` renderer support functionality is optional. If it is supplied, it should report whether or not the renderer is interactive.

```
#define kQ3XMethodTypeRendererIsInteractive
                    Q3_METHOD_TYPE('i','s','i','n')
```

DESCRIPTION

There is no actual method required for `kQ3XMethodTypeRendererIsInteractive`. The metahandler just returns `(TQ3XFunctionPointer)kQ3True` if the renderer is intended to be used in interactive settings and `(TQ3XFunctionPointer)kQ3False` otherwise.

If neither value is returned, the renderer is assumed to be noninteractive.

TQ3XRendererModalConfigureMethod

The `TQ3XRendererModalConfigureMethod` renderer support functionality is optional. If it is supplied, it should display a modal dialog to let the user edit the renderer settings found in the renderer's private data.

```
#define kQ3XMethodTypeRendererModalConfigure
                    Q3_METHOD_TYPE('r','d','m','c')

typedef TQ3Status (*TQ3XRendererModalConfigureMethod)(
                    TQ3RendererObject    renderer,
                    TQ3DialogAnchor      dialogAnchor,
                    TQ3Boolean           *canceled,
                    void                 *rendererPrivate);
```

`renderer` A renderer object.

`dialogAnchor` Platform-specific data passed by the client to support movable modal dialogs (described below).

`canceled` Returns a boolean value to indicate that the user canceled the dialog.

`rendererPrivate`
A pointer to structure of size `instanceSize`, passed into `Q3ObjectHierarchy_RegisterClass`, and initialized in your `kQ3MethodTypeObjectNew` method.

DESCRIPTION

The renderer calls `TQ3XRendererModalConfigureMethod` for events not handled by the normal settings dialog; this is needed to support movable modal dialogs.

The `dialogAnchor` parameter is platform-specific. With the MacOS it is a callback to the calling application's event handler, which must return `kQ3True` if it handles the event passed to the callback and `kQ3False` if not. An application that doesn't want to support a movable modal configure dialog should pass `NULL` for the `clientEventHandler` field of `TQ3DialogAnchor` and should implement a nonmovable dialog.

Modal dialogs in windows applications are always movable. With Windows, therefore, `dialogAnchor` is the handle of the owning window, typically the application's main window.

TQ3XRendererGetNickNameStringMethod

The `TQ3XRendererGetNickNameStringMethod` renderer support functionality is optional. If it is supplied, it lets an application collect the name of the renderer for display in user interface items such as menus. Such a name may be localized or may be more user-friendly than the name string provided at registration.

```
#define kQ3XMethodTypeRendererGetNickNameString
                    Q3_METHOD_TYPE ('r','d','y','u')

typedef TQ3Status (TQ3XRendererGetNickNameStringMethod)(
                    unsigned char    *dataBuffer,
                    unsigned long    bufferSize,
                    unsigned long    *actualDataSize);
```

`dataBuffer` Data buffer to hold the renderer's name.

`bufferSize` The actual size of the memory block pointed to by `dataBuffer`.

`actualDataSize`
On return, the actual number of bytes written to the buffer; or if `dataBuffer` is `NULL`, the required size of `dataBuffer`.

DESCRIPTION

An application uses `TQ3XRendererGetNickNameStringMethod` to get a user-identifiable name of a renderer. If `dataBuffer` is `NULL`, `actualDataSize` returns the required size in bytes of a data buffer large enough to store the renderer's name.

EXAMPLE

The following is example code for getting the name strings for all the currently-registered plug-in renderers.

```
void Menu_Init(
    void)
{
    Handle              menuBar;
    MenuHandle          menu;
    TQ3SubClassData     subClassData;
    TQ3ObjectType       *classPointer;
    short               i;

    if (!(menuBar = GetNewMBar(kMenuBar))) {
        SysBeep(1);
        ExitToShell();
    }

    SetMenuBar(menuBar);
    DisposeHandle(menuBar);
    DrawMenuBar();

    AddResMenu(GetMHandle(kMenu_Apple), 'DRVR');
    InsertMenu(GetMenu(…));

    menu = GetMHandle(kMenu_Renderer);

    Q3ObjectHierarchy_GetSubClassData(
        kQ3SharedTypeRenderer, &subClassData);

    classPointer = subClassData.classTypes;
    i = subClassData.numClasses;

    while (i--) {
        TQ3ObjectClassNameString            objectClassName;
        Q3RendererClass_GetNickNameString(
            *classPointer, objectClassName);
        AppendMenu(menu, c2pstr(objectClassName));
        gRendererCount++;
        classPointer++;
    }

    Q3ObjectHierarchy_EmptySubClassData(
    &subClassData);
}
```

TQ3XRendererGetConfigurationDataMethod

The `TQ3XRendererGetConfigurationDataMethod` renderer support functionality is optional. If it is supplied, it lets an application collect private configuration data from the renderer, which it will then save. The application may save the data in a preference file, in a registry key (in the Windows environment), or in a style template. The application will normally tag the data with the renderer's object name.

```
#define kQ3XMethodTypeRendererGetConfigurationData
                    Q3_METHOD_TYPE('r','d','g','p')

typedef TQ3Status (*TQ3XRendererGetConfigurationDataMethod)(
                    TQ3RendererObject    renderer,
                    unsigned char        *dataBuffer,
                    unsigned long        bufferSize,
                    unsigned long        *actualDataSize,
                    void                 *rendererPrivate);
```

`renderer` A renderer object.

`dataBuffer` Buffer to hold the renderer's configuration data.

`bufferSize` The actual size of the memory block pointed to by `dataBuffer`.

`actualDataSize`
On return, the actual number of bytes written to the buffer; if `dataBuffer` is `NULL`, it is the required size of the buffer.

`rendererPrivate`
A pointer to structure of size `instanceSize`, passed into `Q3ObjectHierarchy_RegisterClass`, and initialized in your `kQ3MethodTypeObjectNew` method.

DESCRIPTION

The renderer calls `TQ3XRendererGetConfigurationDataMethod` to access configuration data from a renderer's private data space. If `dataBuffer` is `NULL`, `actualDataSize` returns the required size in bytes of a data buffer large enough to store the private data.

TQ3XRendererSetConfigurationDataMethod

The `TQ3XRendererSetConfigurationDataMethod` renderer support functionality is optional. If it is supplied, it lets an application pass private configuration data that it previously obtained via `Q3Renderer_GetConfigurationData`.

```
#define kQ3XMethodTypeRendererSetConfigurationData
                    Q3_METHOD_TYPE('r','d','s','p')

typedef TQ3Status (*TQ3XRendererSetConfigurationDataMethod)(
                    TQ3RendererObject    renderer,
                    unsigned char        *dataBuffer,
                    unsigned long        bufferSize,
                    void                 *rendererPrivate);
```

`renderer` A renderer object.

`dataBuffer` Buffer to hold the renderer's configuration data.

`bufferSize` The actual size of the memory block pointed to by `dataBuffer`.

`rendererPrivate`
A pointer to structure of size `instanceSize`, passed into `Q3ObjectHierarchy_RegisterClass`, and initialized in your `kQ3MethodTypeObjectNew` method.

DESCRIPTION

The renderer calls `TQ3XRendererSetConfigurationDataMethod` to place previously obtained renderer configuration data into a renderer's private data space.

Update Methods

This section describes renderer update methods.

TQ3XRendererUpdateStyleMethod

```
#define kQ3XMethodTypeRendererUpdateStyleMetaHandler
                    Q3_METHOD_TYPE ('r','d','y','u')

typedef TQ3XFunctionPointer
                    (*TQ3XRendererUpdateStyleMetaHandlerMethod)(
                    TQ3ObjectType    styleType);

typedef TQ3Status (*TQ3XRendererUpdateStyleMethod)(
                    TQ3ViewObject    view,
                    void             *rendererPrivate,
                    const void       *publicData);
```

`styleType` A style object type (see "Q3Style_GetType," beginning on page 557).

`view` The current view being rendered to.

`rendererPrivate`
A pointer to structure of size `instanceSize`, passed into `Q3ObjectHierarchy_RegisterClass`, and initialized in your `kQ3MethodTypeObjectNew` method.

`publicData` A pointer the public data structure associated with a style. You may retain a copy of the pointer passed to you and use it to access the state. The state remains valid until either another update on this style type is made at this depth or a `pop` action occurs. A `push` action does not invalidate the pointer.

DESCRIPTION

The method type `kQ3XMethodTypeRendererUpdateStyleMetaHandler` returns a function pointer of type `TQ3XRendererUpdateStyleMetaHandlerMethod`. This function enables a style of type `styleType` and returns methods of type `TQ3XRendererUpdateStyleMethod`.

The types of the data structures pointed to by `publicData` corresponds to the style types shown in the following table:

Style type	Data structure type
`kQ3StyleTypeBackfacing`	`TQ3BackfacingStyle *`
`kQ3StyleTypeInterpolation`	`TQ3InterpolationStyle *`
`kQ3StyleTypeFill`	`TQ3FillStyle *`
`kQ3StyleTypePickID`	`unsigned long *`
`kQ3StyleTypeReceiveShadows`	`TQ3Boolean *`
`kQ3StyleTypeHighlight`	`TQ3AttributeSet *`
`kQ3StyleTypeSubdivision`	`TQ3SubdivisionStyleData *`
`kQ3StyleTypeOrientation`	`TQ3OrientationStyle *`
`kQ3StyleTypePickParts`	`TQ3PickParts *`
`kQ3StyleTypeAntiAlias`	`TQ3AntiAliasStyleData`

TQ3XRendererUpdateAttributeMethod

```
#define kQ3XMethodTypeRendererUpdateAttributeMetaHandler
                    Q3_METHOD_TYPE ('r','d','a','u')

typedef TQ3XFunctionPointer
                    (*TQ3XRendererUpdateAttributeMetaHandlerMethod)(
                    TQ3AttributeType    attributeType);

typedef TQ3Status (*TQ3XRendererUpdateAttributeMethod)(
                    TQ3ViewObject    view,
                    void             *rendererPrivate,
                    const void       *publicData);
```

`attributeType` An attribute object type (see "Types of Attributes and Attribute Sets," beginning on page 516).

`view` The current view being rendered to.

`rendererPrivate`
: A pointer to structure of size `instanceSize`, passed into `Q3ObjectHierarchy_RegisterClass`, and initialized in your `kQ3MethodTypeObjectNew` method.

`publicData`
: A pointer the public data structure associated with an attribute object. You may retain a copy of the pointer passed to you and use it to access the state. The state remains valid until either another update on this attribute type is made at this depth or a `pop` action occurs. A `push` action does not invalidate the pointer.

DESCRIPTION

The method type `kQ3XMethodTypeRendererUpdateAttributeMetaHandler` returns a function pointer of type `TQ3XRendererUpdateAttributeMetaHandlerMethod`. This function enables an attribute of type `attributeType` and returns methods of type `TQ3XRendererUpdateAttributeMethod`.

The types of the data structures pointed to by `publicData` corresponds to the attribute types shown in the following table:

Attribute type	Data structure type
`kQ3AttributeTypeAmbientCoefficient`	`float *`
`kQ3AttributeTypeDiffuseColor`	`TQ3ColorRGB *`
`kQ3AttributeTypeNormal`	`TQ3Vector3D *`
`kQ3AttributeTypeSpecularColor`	`TQ3ColorRGB *`
`kQ3AttributeTypeSpecularControl`	`float *`
`kQ3AttributeTypeTransparencyColor`	`TQ3ColorRGB *`
`kQ3AttributeTypeSurfaceShader`	`TQ3ShaderObject *`

TQ3XRendererUpdateShaderMethod

```
#define kQ3XMethodTypeRendererUpdateShaderMetaHandler
                    Q3_METHOD_TYPE ('r','d','s','u')

typedef TQ3XFunctionPointer
                    (*TQ3XRendererUpdateShaderMetaHandlerMethod)(
                    TQ3ObjectType    shaderType);
```

```
typedef TQ3Status (*TQ3XRendererUpdateShaderMethod)(
                    TQ3ViewObject     view,
                    void              *rendererPrivate,
                    TQ3ShaderObject    *shaderObject);
```

`shaderType` A shader object type (see "Q3Shader_GetType," beginning on page 929).

`view` The current view being rendered to.

`rendererPrivate`
A pointer to structure of size `instanceSize`, passed into `Q3ObjectHierarchy_RegisterClass`, and initialized in your `kQ3MethodTypeObjectNew` method.

`shaderObject` A shader object. A `shaderObject` pointer is never `NULL`. The pointer to it, `*shaderObject`, may be `NULL`; this value disables that particular type of shader. Generally, a renderer should retain a reference obtained from `Q3Shared_GetReference` to a non-`NULL` *`shaderObject` pointer and use it to shade any subsequently rendered objects.

DESCRIPTION

The method type `kQ3XMethodTypeRendererUpdateShaderMetaHandler` returns a function pointer of type `TQ3XRendererUpdateShaderMetaHandlerMethod`. This function enables a shader of type `shaderType` and returns methods of type `TQ3XRendererUpdateShaderMethod`.

▲ **WARNING**
The surface shader state may be overridden by a non-`NULL` surface shader attribute state. Do not depend on these states being updated in any particular order. ▲

TQ3XRendererUpdateMatrixMethod

```
#define kQ3XMethodTypeRendererUpdateMatrixMetaHandler
                    Q3_METHOD_TYPE ('r','d','x','u')

typedef TQ3XMetaHandler TQ3XRendererUpdateMatrixMetaHandlerMethod;
```

```
typedef TQ3Status (*TQ3XRendererUpdateMatrixMethod) (
                    TQ3ViewObject         view,
                    void                  *rendererPrivate,
                    const TQ3Matrix4x4    *matrix);
```

`view` The current view being rendered to.

`rendererPrivate`
A pointer to structure of size `instanceSize`, passed into `Q3ObjectHierarchy_RegisterClass`, and initialized in your `kQ3MethodTypeObjectNew` method.

`matrix` A pointer to a matrix.

DESCRIPTION

`TQ3XRendererUpdateMatrixMetaHandlerMethod` switches on the following types of methods and returns methods of type `TQ3XRendererUpdateMatrixMethod`:

```
#define kQ3XMethodTypeRendererUpdateMatrixLocalToWorld
                    Q3_METHOD_TYPE ('u','l','w','x')

#define kQ3XMethodTypeRendererUpdateMatrixLocalToWorldInverse
                    Q3_METHOD_TYPE ('u','l','w','i')

#define kQ3XMethodTypeRendererUpdateMatrixLocalToWorldInverseTranspose
                    Q3_METHOD_TYPE ('u','l','w','t')

#define kQ3XMethodTypeRendererUpdateMatrixLocalToCamera
                    Q3_METHOD_TYPE ('u','l','c','x')

#define kQ3XMethodTypeRendererUpdateMatrixLocalToFrustum
                    Q3_METHOD_TYPE ('u','l','f','x')

#define kQ3XMethodTypeRendererUpdateMatrixWorldToFrustum
                    Q3_METHOD_TYPE ('u','w','f','x')
```

IMPORTANT

Matrix update methods are called in the order shown in the foregoing list. ▲

Drawing State Methods

This section describes renderer support methods for the drawing state in a view.

TQ3XRendererStartFrameMethod

The `TQ3XRendererStartFrameMethod` functionality is required in a renderer.

```
#define kQ3XMethodTypeRendererStartFrame
                    Q3_METHOD_TYPE('r','d','c','l')

typedef TQ3Status (*TQ3XRendererStartFrameMethod)(
                    TQ3ViewObject          view,
                    void                   *rendererPrivate,
                    TQ3DrawContextObject    drawContext);
```

`view` The current view being rendered to.

`rendererPrivate`
A pointer to structure of size `instanceSize`, passed into `Q3ObjectHierarchy_RegisterClass`, and initialized in your `kQ3MethodTypeObjectNew` method.

`drawContext` A draw context object.

DESCRIPTION

The `kQ3XMethodTypeRendererStartFrame` method type returns a function pointer of type `TQ3XRendererStartFrameMethod`.

`TQ3XRendererStartFrameMethod` is first called from `Q3View_StartRendering`. It should perform these tasks:

- initialize any renderer states to their default values
- extract all useful data from the `drawContext` object
- if the renderer passed in `kQ3RendererFlagClearBuffer` at registration, it should also clear the draw context.
- clear the `drawContext` object

When clearing the `drawContext` object, the renderer may opt to use any one of these procedures:

- not clear anything (for example, if it already touches every pixel)
- clear with its own routine, or
- use the draw context default clear method by calling `Q3DrawContext_Clear`. `Q3DrawContext_Clear` takes advantage of any hardware in the system that is available for clearing the drawing context.

`TQ3XRendererStartFrameMethod` also signals the beginning of receiving default submit commands from the view. The renderer will receive updates for the default view state via its update methods before `StartScene` is called. Renderer submit and update methods are discussed on page 792.

TQ3XRendererStartPassMethod

The `TQ3XRendererStartPassMethod` functionality is required in a renderer.

```
#define kQ3XMethodTypeRendererStartPass
                    Q3_METHOD_TYPE('r','d','s','t')

typedef TQ3Status (*TQ3XRendererStartPassMethod)(
                    TQ3ViewObject      view,
                    void               *rendererPrivate,
                    TQ3CameraObject    camera,
                    TQ3GroupObject     lightGroup);
```

`view` The current view being rendered to.

`rendererPrivate`
A pointer to structure of size `instanceSize`, passed into `Q3ObjectHierarchy_RegisterClass`, and initialized in your `kQ3MethodTypeObjectNew` method.

`camera` A camera object.

`lightGroup` A light group object.

DESCRIPTION

The `kQ3XMethodTypeRendererStartPass` method type returns a function pointer of type `TQ3XRendererStartPassMethod`.

`TQ3XRendererStartPassMethod` is called during `Q3View_StartRendering` after the `StartFrame` command. It should perform these tasks:

- collect camera and light information
- prepare any additional states before object submit calls are made

If the renderer supports deferred camera transformation, `camera` represents the main camera that will be submitted somewhere in the hierarchy. Its value is never `NULL`. If your renderer does not support deferred camera transformation, `camera` represents the transformed camera.

If the renderer supports deferred light transformation, the value of `lightGroup` will be `NULL` and it will be submitted to your light draw methods instead.

Calling `TQ3XRendererStartPassMethod` signals the end of the default update state and the start of submit commands from the user to the view.

TQ3XRendererFlushFrameMethod

The `TQ3XRendererFlushFrameMethod` functionality is optional and is implemented only by asynchronous renderers.

```
#define kQ3XMethodTypeRendererFlushFrame
                    Q3_METHOD_TYPE('r','d','f','l')

typedef TQ3Status (*TQ3XRendererFlushFrameMethod)(
                    TQ3ViewObject          view,
                    void                   *rendererPrivate,
                    TQ3DrawContextObject    drawContext);
```

`view` The current view being rendered to.

`rendererPrivate`
A pointer to structure of size `instanceSize`, passed into `Q3ObjectHierarchy_RegisterClass`, and initialized in your `kQ3MethodTypeObjectNew` method.

`drawContext` A draw context object.

DESCRIPTION

The `kQ3XMethodTypeRendererFlushFrame` method type returns a function pointer of type `TQ3XRendererFlushFrameMethod`.

`TQ3XRendererFlushFrameMethod` is called between the `StartScene` and `EndScene` methods. It is called when the user wishes to flush any asynchronous drawing tasks that draw to the `drawContext` object, but does not want to block asynchronous drawing altogether. As a result, an image should eventually appear asynchronously. In asynchronous rendering, this call is **non-blocking.**

An interactive renderer should ensure that all received geometries are drawn in the image. If it controls the hardware, it should force the hardware to generate an image.

A deferred renderer should exhibit similar behavior, though this is not a requirement. A deferred renderer should spawn a process that generates a partial image from the currently accumulated drawing state.

Implementing `TQ3XRendererFlushFrameMethod` is not recommended for computation-intensive renderers such as ray-tracers.

TQ3XRendererEndPassMethod

The `TQ3XRendererEndPassMethod` functionality is required in a renderer.

```
#define kQ3XMethodTypeRendererEndPass
                    Q3_METHOD_TYPE('r','d','e','d')

typedef TQ3ViewStatus (*TQ3XRendererEndPassMethod)(
                    TQ3ViewObject    view,
                    void             *rendererPrivate);
```

`view` The current view being rendered to.

`rendererPrivate`
A pointer to structure of size `instanceSize`, passed into `Q3ObjectHierarchy_RegisterClass`, and initialized in your `kQ3MethodTypeObjectNew` method.

DESCRIPTION

The `kQ3XMethodTypeRendererEndPass` method type returns a function pointer of type `TQ3XRendererEndPassMethod`.

`TQ3XRendererEndPassMethod` is called at `Q3View_EndRendering` and signals the end of sending submit commands to the view. If the renderer requires another pass on the data being rendered, it should return `kQ3ViewStatusRetraverse`.

If rendering was cancelled, `TQ3XRendererEndPassMethod` will not be called and the view will return `kQ3ViewStatusCancelled`. Otherwise, your renderer should initiate completion of the process of generating the image in the drawing context. If it has buffered any drawing data, the data must be flushed. `TQ3XRendererEndPassMethod` should have an effect similar to that of `FlushFrame`.

A synchronous renderer must update the front buffer; otherwise `DrawContext` will update the front buffer after returning. If a synchronous renderer supports `kQ3RendererClassSupportDoubleBuffer`, it must finish rendering the entire frame.

An asynchronous renderer must spawn a rendering thread for the entire frame. If it supports `kQ3RendererClassSupportDoubleBuffer`, it must eventually either update the front buffer asynchronously, then call `Q3View_EndFrame`, or update the back buffer asynchronously, then call `Q3View_EndFrame`.

If an error occurs with `TQ3XRendererEndPassMethod`, the renderer should call `Q3Error_Post` and return `kQ3ViewStatusError`.

TQ3XRendererEndFrameMethod

The `TQ3XRendererEndFrameMethod` functionality is optional and is implemented only by asynchronous renderers.

```
#define kQ3XMethodTypeRendererEndFrame
                    Q3_METHOD_TYPE('r','d','s','y')

typedef TQ3Status (*TQ3XRendererEndFrameMethod)(
                    TQ3ViewObject          view,
                    void                   *rendererPrivate,
                    TQ3DrawContextObject    drawContext);
```

`view` The current view being rendered to.

`rendererPrivate`
: A pointer to structure of size `instanceSize`, passed into `Q3ObjectHierarchy_RegisterClass`, and initialized in your `kQ3MethodTypeObjectNew` method.

`drawContext`
: A draw context object.

DESCRIPTION

The `kQ3XMethodTypeRendererEndFrame` method type returns a function pointer of type `TQ3XRendererEndFrameMethod`.

`TQ3XRendererEndFrameMethod` is called from `Q3View_Sync`, which is called after `Q3View_EndRendering`. It signals that the user wishes to see the completed image and is willing to block drawing. No call to `Q3View_EndFrame` is needed.

If your renderer supports `kQ3RendererFlagDoubleBuffer`, it must update the front buffer completely; otherwise it must update the back buffer completely.

`TQ3XRendererEndFrameMethod` is equivalent in functionality to `FlushFrame`, but it blocks drawing until the image is completed.

If `TQ3XRendererEndFrameMethod` is not supplied, the default action is no operation.

Note
Registering a method of type `TQ3XRendererEndFrameMethod` indicates that your renderer will continue rendering after `Q3View_EndRendering` has been called. ◆

TQ3XRendererCancelMethod

The `TQ3XRendererEndPassMethod` functionality is required in a renderer.

```
#define kQ3XMethodTypeRendererCancel
                    Q3_METHOD_TYPE('r','d','a','b')

typedef void (*TQ3XRendererCancelMethod)(
                    TQ3ViewObject    view,
                    void             *rendererPrivate);
```

`view` The current view being rendered to.

`rendererPrivate`
A pointer to structure of size `instanceSize`, passed into `Q3ObjectHierarchy_RegisterClass`, and initialized in your `kQ3MethodTypeObjectNew` method.

DESCRIPTION

The `kQ3XMethodTypeRendererCancel` method type returns a function pointer of type `TQ3XRendererCancelMethod`.

`TQ3XRendererCancelMethod` is called after `Q3View_StartRendering` and signals the termination of all rendering operations. Your renderer should clean up any cached data and cancel all rendering operations.

IMPORTANT

If `TQ3XRendererCancelMethod` is called before `Q3View_EndRendering`, `TQ3XRendererEndPassMethod` is not called. ▲

If `TQ3XRendererCancelMethod` is called after `Q3View_EndRendering`, your renderer should kill any rendering threads and terminate any further rendering. If your renderer is asynchronous, `TQ3XRendererCancelMethod` will never be called after `Q3View_EndRendering`.

Push and Pop Methods

You can call renderer push and pop methods whenever the graphics state in the view needs to be pushed or popped. Code may isolate the state by submitting a display group that pushes and pops or by making calls such as the following:

```
Q3Attribute_Submit(kQ3AttributeTypeDiffuseColor, &red, view);
Q3Attribute_Submit(kQ3AttributeTypeTransparencyColor, &blue, view);
Q3Attribute_Submit(kQ3AttributeTypeSpecularColor, &white, view);
Q3Box_Submit(&unitBox, view);
Q3TranslateTransform_Submit(&unitVector, view);
Q3Push_Submit(view);
Q3Attribute_Submit(kQ3AttributeTypeDiffuseColor, &blue, view);
Q3Attribute_Submit(kQ3AttributeTypeTransparencyColor, &green, view);
Q3Box_Submit(&unitBox, view);
```

```
Q3Pop_Submit(view);
Q3TranslateTransform_Submit(&unitVector, view);
Q3Box_Submit(&unitBox, view);
```

Even though you support `RendererPush` and `RendererPop` in your renderer, you must also maintain your drawing state as a stack. Your code will not automatically be updated with the popped state after `RendererPop` is called. If you do not support push and pop functionality in your renderer, you may maintain a single copy of the drawing state. Your code will be updated with changed fields after the view stack is popped.

A renderer that supports `RendererPush` and `RendererPop` will be called in the following sequence, based on the previous example:

```
RendererUpdateAttributeDiffuseColor(...,&red)
RendererUpdateAttributeTransparencyColor(...,&blue)
RendererUpdateAttributeSpecularColor(...,&white)
RendererUpdateMatrixLocalToWorld(...)
RendererSubmitGeometryBox(...)
RendererPush(...)
RendererUpdateAttributeDiffuseColor(...,&blue)
RendererUpdateAttributeTransparencyColor(...,&green)
RendererSubmitGeometryBox(...)
RendererPop(...)
RendererUpdateMatrixLocalToWorld(...)
RendererSubmitGeometryBox(...)
```

A renderer that does not support `RendererPush` and `RendererPop` will be called in the following sequence:

```
RendererUpdateAttributeDiffuseColor(...,&red)
RendererUpdateAttributeTransparencyColor(...,&blue)
RendererUpdateAttributeSpecularColor(...,&white)
RendererUpdateMatrixLocalToWorld(...)
RendererSubmitGeometryBox(...)
RendererUpdateAttributeDiffuseColor(...,&blue)
RendererUpdateAttributeTransparencyColor(...,&green)
RendererSubmitGeometryBox(...)
RendererUpdateAttributeDiffuseColor(...,&red)
RendererUpdateAttributeTransparencyColor(...,&blue)
RendererUpdateMatrixLocalToWorld(...)
RendererSubmitGeometryBox(...)
```

Note
In both cases, update calls may be in a different order, as explained in "Renderer Methods," beginning on page 792. ◆

TQ3XRendererPushMethod

```
#define kQ3XMethodTypeRendererPush
                    Q3_METHOD_TYPE('r','d','p','s')

typedef TQ3Status (*TQ3XRendererPushMethod)(
                    TQ3ViewObject    view,
                    void             *rendererPrivate);
```

`view` The current view being rendered to.

`rendererPrivate`
A pointer to structure of size `instanceSize`, passed into `Q3ObjectHierarchy_RegisterClass`, and initialized in your `kQ3MethodTypeObjectNew` method.

DESCRIPTION

The `kQ3XMethodTypeRendererPush` method type returns a function pointer of type `TQ3XRendererPushMethod`.

TQ3XRendererPopMethod

```
#define kQ3XMethodTypeRendererPop
                    Q3_METHOD_TYPE('r','d','p','o')

typedef TQ3Status (*TQ3XRendererPopMethod)(
                    TQ3ViewObject    view,
                    void             *rendererPrivate);
```

`view` The current view being rendered to.

`rendererPrivate`
A pointer to structure of size `instanceSize`, passed into `Q3ObjectHierarchy_RegisterClass`, and initialized in your `kQ3MethodTypeObjectNew` method.

DESCRIPTION

The `kQ3XMethodTypeRendererPop` method type returns a function pointer of type `TQ3XRendererPopMethod`.

Renderer Cull Method

This section describes `TQ3XRendererIsBoundingBoxVisibleMethod`, a renderer support method used to cull group and geometric objects.

TQ3XRendererIsBoundingBoxVisibleMethod

```
#define kQ3XMethodTypeRendererIsBoundingBoxVisible
                    Q3_METHOD_TYPE('r','d','b','x')

typedef TQ3Boolean (*TQ3XRendererIsBoundingBoxVisibleMethod)(
                    TQ3ViewObject          view,
                    void                   *rendererPrivate,
                    const TQ3BoundingBox    *bBox);
```

`view` The current view being rendered to.

`rendererPrivate`
A pointer to structure of size `instanceSize`, passed into `Q3ObjectHierarchy_RegisterClass`, and initialized in your `kQ3MethodTypeObjectNew` method.

`bBox` A bounding box.

DESCRIPTION

The `kQ3XMethodTypeRendererIsBoundingBoxVisible` method type returns a function pointer of type `TQ3XRendererIsBoundingBoxVisibleMethod`.

This renderer support method is called to cull complex groups and geometries by passing their bounding box in local space. It should transform the local-space bounding box coordinates to frustum space and return a value of type `TQ3Boolean` indicating whether or not the box appears within the viewing frustum. If no method is supplied, the default behavior is to return `kQ3True`.

Draw Region Interface

The draw region interface lets renderers access the QuickDraw 3D frame buffer and communicate information about the configuration and state of the drawing context. Renderers are required to use some portions of the draw region interface. Other portions may or may not be required, depending on the nature of the renderer, as specified below.

Because of the complexity inherent in dealing with frame buffers and other rendering targets, you are encouraged to examine the Apple Sample Renderer (SR) provided on the QD3D 1.5 Software Developers Kit (SDK). It helps explain the functionality described here and illustrates how to use it.

Obtaining a DrawRegion

Draw regions are attached to the draw context by QuickDraw 3D. Typically, there will be one draw region per monitor attached to a machine. A draw region is an opaque structure:

```
typedef struct TQ3XDrawRegionPrivate *TQ3XDrawRegion;
```

You can use the `Q3XDrawContext_GetDrawRegion` routine to get the first draw region attached to a draw context. If there are additional draw regions, the rest of them can be obtained by repeated calls to `Q3XDrawRegion_GetNextRegion`. Your `Q3XDrawRegion_GetNextRegion` call will return `NULL` after the last draw region has been retrieved.

Q3XDrawContext_GetDrawRegion

The `Q3XDrawContext_GetDrawRegion` function lets you obtain the first draw region attached to a draw context.

```
TQ3Status Q3XDrawContext_GetDrawRegion(
                    TQ3DrawContextObject    drawContext,
                    TQ3XDrawRegion          *drawRegion);
```

`drawContext` A draw context.

`drawRegion` On return, the first draw region attached to `drawContext`.

DESCRIPTION

The `Q3XDrawContext_GetDrawRegion` function returns, in the `drawRegion` parameter, the first draw region attached to the draw context designated by the `drawContext` parameter.

Q3XDrawRegion_GetNextRegion

The `Q3XDrawRegion_GetNextRegion` function returns additional draw regions attached to a draw context.

```
TQ3Status Q3XDrawRegion_GetNextRegion(
                    TQ3XDrawRegion    drawRegion,
                    TQ3XDrawRegion    *nextDrawRegion);
```

`drawRegion` The draw region returned by `Q3XDrawContext_GetDrawRegion`.

`nextDrawRegion`
An additional draw region attached to the same draw context, or `NULL` if there are no more.

DESCRIPTION

The `Q3XDrawRegion_GetNextRegion` function returns, in the `nextDrawRegion` parameter, a draw region attached to the same draw context as `drawRegion`. It returns `NULL` after it has returned all the draw regions for that draw context.

Draw Region Validation

The state of all draw regions attached to a draw context are validated by the draw context and draw region internal code before a frame is rendered. However, renderers may wish to modify their own state—for example, to alter cached information. To support such modifications, QuickDraw 3D supplies renderers with information about what has changed in the draw region via the `Q3XDrawContext_GetValidationFlags` routine.

The following mask values let you determine which draw region states have changed since the last frame. Call the `Q3XDrawContext_ClearValidationFlags` routine afterward to clear the flag information.

```
typedef enum TQ3XDrawContextValidationMasks {
    kQ3XDrawContextValidationClearFlags             = 0x00000000L,
    kQ3XDrawContextValidationDoubleBuffer           = 1 << 0,
    kQ3XDrawContextValidationShader                 = 1 << 1,
    kQ3XDrawContextValidationClearFunction          = 1 << 2,
    kQ3XDrawContextValidationActiveBuffer           = 1 << 3,
    kQ3XDrawContextValidationInternalOffScreen      = 1 << 4,
    kQ3XDrawContextValidationPane                   = 1 << 5,
    kQ3XDrawContextValidationMask                   = 1 << 6,
    kQ3XDrawContextValidationDevice                 = 1 << 7,
    kQ3XDrawContextValidationWindow                 = 1 << 8,
    kQ3XDrawContextValidationWindowSize             = 1 << 9,
    kQ3XDrawContextValidationWindowClip             = 1 << 10,
    kQ3XDrawContextValidationWindowPosition         = 1 << 11,
    kQ3XDrawContextValidationPlatformAttributes     = 1 << 12,
    kQ3XDrawContextValidationForegroundShader       = 1 << 13,
    kQ3XDrawContextValidationBackgroundShader       = 1 << 14,
    kQ3XDrawContextValidationColorPalette           = 1 << 15,
    kQ3XDrawContextValidationAll                    = ~0
} TQ3XDrawContextValidationMasks;

typedef unsigned long       TQ3XDrawContextValidation;
```

Q3XDrawContext_GetValidationFlags

The `Q3XDrawContext_GetValidationFlags` function lets you determine which draw context states have changed since the last frame.

```
typedef unsigned long TQ3XDrawContextValidation;

TQ3Status Q3XDrawContext_GetValidationFlags(
                    TQ3DrawContextObject        drawContext,
                    TQ3XDrawContextValidation   *validationFlags);
```

`drawContext` A draw context.

`validationFlags`
Validation flags from the `TQ3XDrawContextValidationMasks` enumeration.

DESCRIPTION

The `Q3XDrawContext_GetValidationFlags` function returns, in the `validationFlags` parameter, flags indicating which states of the `drawContext` draw context have changed since the last frame.

Q3XDrawContext_ClearValidationFlags

Once your renderer has updated the current draw context state, it should call the `Q3XDrawContext_ClearValidationFlags` routine to clear the validation flags. Otherwise flags may remain set during the next frame when the state has not changed.

```
TQ3Status Q3XDrawContext_ClearValidationFlags(
                    TQ3DrawContextObject    drawContext);
```

`drawContext` A draw context.

DESCRIPTION

The `Q3XDrawContext_ClearValidationFlags` function clears all the validation flags returned by `Q3XDrawContext_GetValidationFlags`.

Draw Region Services

The draw region can perform, at the direction of the renderer, two optional services: it can clear the rendering target (e.g., a window) and it can lock the target's `DDSurface` draw context type in a Windows environment.

These two services are described by the following enumeration:

```
typedef enum TQ3XDrawRegionServicesMasks {
    kQ3XDrawRegionServicesNoneFlag                  = 0L,
    kQ3XDrawRegionServicesClearFlag                 = 1 << 0,
    kQ3XDrawRegionServicesDontLockDDSurfaceFlag     = 1 << 1
} TQ3XDrawRegionServicesMasks;

typedef unsigned long TQ3XDrawRegionServices;
```

The renderer controls which of these services are used by passing a value in `Q3XDrawRegion_Start` (page 823) or `Q3XDrawRegion_StartAccessToImageBuffer` (page 823).

Note
If a draw context must be cleared before each frame, either the draw region or the renderer must be directed to do the clearing. The same applies to locking the `DDSurface` when using that particular Windows draw context type. ◆

Starting and Ending Draw Regions

Generally, there is one draw region per monitor. At any particular time, however, a window associated with a draw context may not appear in the active monitor. Thus one or more draw regions may be inactive at a given time and should be explicitly ignored by the renderer. You can use the `Q3XDrawRegion_IsActive` function to determine if a given draw region is currently active.

For active draw regions, one of two calls must be made before rendering into the draw region, regardless of how much draw region functionality is required: `Q3XDrawRegion_Start` or `Q3XDrawRegion_StartAccessToImageBuffer`.

You can use the `Q3XDrawRegion_Start` function if double-buffering and image access services are not needed; it just returns a pointer to the draw region descriptor `TQ3XDrawRegionDescriptor` (page 825). However, the function described below is more useful for most plug-in renderers.

The `Q3XDrawRegion_StartAccessToImageBuffer` function provides image access and double-buffering services for your renderer. Besides returning a pointer to the draw region descriptor, it also returns a pointer to the raster image into which rendering occurs. For single-buffering, this is a pointer to video memory; for double buffering, it is a pointer to a system-allocated back buffer. The double-buffering process is handled by the draw region. Most plug-in renderers must use the `Q3XDrawRegion_StartAccessToImageBuffer` function instead of `Q3XDrawRegion_Start`.

Once a renderer is done with a draw region, it must call the `Q3XDrawRegion_End` function.

Q3XDrawRegion_IsActive

The `Q3XDrawRegion_IsActive` function lets you determine whether a draw region is active or inactive. If it is inactive, the renderer may ignore it.

```
TQ3Status Q3XDrawRegion_IsActive(
                    TQ3XDrawRegion    drawRegion,
                    TQ3Boolean        *isActive);
```

`drawRegion` A draw region.

`isActive` Returns `kQ3True` if the draw region is active; `kQ3False` otherwise.

DESCRIPTION

The `Q3XDrawRegion_IsActive` function returns, in the `isActive` parameter, `kQ3True` if the draw region designated by `drawRegion` is active and `kQ3False` otherwise.

Q3XDrawRegion_Start

The `Q3XDrawRegion_Start` function returns a draw region descriptor to a renderer that does not require double-buffering or image access.

```
TQ3Status Q3XDrawRegion_Start(
                    TQ3XDrawRegion              drawRegion,
                    TQ3XDrawRegionServices      services,
                    TQ3XDrawRegionDescriptor    **descriptor);
```

`drawRegion` A draw region.

`services` Draw region services requested (see page 821).

`descriptor` On return, a draw region descriptor (see page 825).

DESCRIPTION

The `Q3XDrawRegion_Start` function returns, in the `descriptor` parameter, a pointer to a draw region descriptor for the draw region designated by `drawRegion`.

The `Q3XDrawRegion_Start` function may be called if double-buffering and image access services are not needed. The renderer must provide a valid value in the services parameter, requesting if clearing or `DDSurface` locking is required (see page 821).

This function is rarely used, because the renderer is then required to do all the work of allocating, locating, clearing, and double buffering the image raster. Most plug-in renderers use the `Q3XDrawRegion_StartAccessToImageBuffer` function instead of `Q3XDrawRegion_Start`.

Q3XDrawRegion_StartAccessToImageBuffer

The `Q3XDrawRegion_StartAccessToImageBuffer` function provides image access and double-buffering services for a renderer. In addition to returning a pointer to the draw region descriptor, it also returns a pointer to the raster image into which rendering occurs is returned. In the case of single-buffering, this is a pointer to the video memory; in the case of double-buffering, it is a pointer to a

system-allocated back buffer. All double-buffering, etc., is now handled by the draw region.

```
TQ3Status Q3XDrawRegion_StartAccessToImageBuffer(
                    TQ3XDrawRegion              drawRegion,
                    TQ3XDrawRegionServices      services,
                    TQ3XDrawRegionDescriptor    **descriptor,
                    void                        **image);
```

`drawRegion` A draw region.

`services` Draw region services requested (see page 821).

`descriptor` On return, a draw region descriptor (see page 825).

`image` On return, a pointer to the target raster image.

DESCRIPTION

The `Q3XDrawRegion_StartAccessToImageBuffer` function returns a draw region descriptor in the `descriptor` parameter and a pointer to the renderer's target raster in the `image` parameter, both for the draw region designated by `drawRegion`.

SPECIAL CONSIDERATIONS

Most plug-in renderers use the `Q3XDrawRegion_StartAccessToImageBuffer` function instead of `Q3XDrawRegion_Start`.

Q3XDrawRegion_End

A renderer must call the `Q3XDrawRegion_End` function when it has finished with a draw region.

```
TQ3Status Q3XDrawRegion_End(
                    TQ3XDrawRegion    drawRegion);
```

`drawRegion` A draw region.

DESCRIPTION

After rendering, the `Q3XDrawRegion_End` function performs internal clean-up and memory release for the draw region designated by `drawRegion`.

Draw Region Descriptor

The `TQ3XDrawRegionDescriptor` data structure describes the raster into which a renderer draws. This data structure contains dimensions, pixel size, and format information.

You can get a pointer to the `TQ3XDrawRegionDescriptor` data structure by calling `Q3XDrawRegion_Start` (page 823) or `Q3XDrawRegion_StartAccessToImageBuffer` (page 823).

```
typedef struct TQ3XDrawRegionDescriptor {
    unsigned long               width;
    unsigned long               height;
    unsigned long               rowBytes;
    unsigned long               pixelSize;
    TQ3XDevicePixelType         pixelType;
    TQ3XColorDescriptor         colorDescriptor;
    TQ3Endian                   bitOrder;
    TQ3Endian                   byteOrder;
    TQ3Bitmap                   *clipMask;
} TQ3XDrawRegionDescriptor;
```

Field descriptions

`width`, `height`	Width and height, in pixels, of the area rendered into.
`rowBytes`	The raster may be embedded in a larger area of memory, so its scan lines may not be contiguous in memory. This field gives the number of bytes to the next row (scan line).
`pixelSize`	The number of bytes in each pixel.
`pixelType`	The formatting type of the pixel; see "Device Pixel Types" (page 826).
`colorDescriptor`	Currently not used.
`bitOrder`, `byteOrder`	Endianess; may be `kQ3EndianBig` or `kQ3EndianLittle`.
`clipMask`	The clip mask for the region; may be `NULL`.

Device Pixel Types

Pixels described in the `TQ3XDrawRegionDescriptor` data structure may be any of several types, as described in the following enumeration. Not all pixel types are supported on all devices.

```
typedef enum TQ3XDevicePixelType {
    kQ3XDevicePixelTypeInvalid      = 0,
    kQ3XDevicePixelTypeRGB32        = 1,
    kQ3XDevicePixelTypeARGB32       = 2,
    kQ3XDevicePixelTypeRGB24        = 3,
    kQ3XDevicePixelTypeRGB16        = 4,
    kQ3XDevicePixelTypeARGB16       = 5,
    kQ3XDevicePixelTypeRGB16_565    = 6,
    kQ3XDevicePixelTypeIndexed8     = 7,
    kQ3XDevicePixelTypeIndexed4     = 8,
    kQ3XDevicePixelTypeIndexed2     = 9,
    kQ3XDevicePixelTypeIndexed1     = 10
} TQ3XDevicePixelType;
```

Color Descriptor

Draw regions store color information in a color descriptor:

```
typedef struct TQ3XColorDescriptor {
    unsigned long       redShift;
    unsigned long       redMask;
    unsigned long       greenShift;
    unsigned long       greenMask;
    unsigned long       blueShift;
    unsigned long       blueMask;
    unsigned long       alphaShift;
    unsigned long       alphaMask;
} TQ3XColorDescriptor;
```

Clipping Information

Draw regions contain information about the clipping state. A region may be in one of three states, encoded by the following enumeration:

```
typedef enum TQ3XClipMaskState {
    kQ3XClipMaskFullyExposed,
    kQ3XClipMaskPartiallyExposed,
    kQ3XClipMaskNotExposed
} TQ3XClipMaskState;
```

A fully exposed draw region has no overlapping windows; a partially exposed draw region has some overlapping windows; a not exposed draw region is entirely offscreen or completely covered by another window. You can determine the clipping state with the `Q3XDrawRegion_GetClipFlags` function.

If the clipping state is `kQ3XClipMaskPartiallyExposed`, you can use the `Q3XDrawRegion_GetClipMask` function to query the draw region for a bitmap that describes the clipping of the window. In the Mac OS environment, you can also use the `Q3XDrawRegion_GetClipRegion` routine to get information about clipping in the form of a handle to a QuickDraw region (a `rgnHandle`), or you can use the `Q3XDrawRegion_GetGDHandle` routine to return a `GDHandle`.

Q3XDrawRegion_GetClipFlags

The `Q3XDrawRegion_GetClipFlags` function lets you get the clipping state of a draw region.

```
TQ3Status Q3XDrawRegion_GetClipFlags(
                    TQ3XDrawRegion      drawRegion,
                    TQ3XClipMaskState   *clipMaskState);
```

`drawRegion` A draw region.

`clipMaskState`
The draw region's clipping mask state.

DESCRIPTION

The `Q3XDrawRegion_GetClipFlags` function returns, in the `clipMaskState` parameter, one of three values that define the draw region's clipping mask state. The `TQ3XClipMaskState` values that can be returned are enumerated on page 827.

Q3XDrawRegion_GetClipMask

The `Q3XDrawRegion_GetClipMask` function lets you get a bitmap that describes the clipping state of a partially exposed draw region.

```
TQ3Status Q3XDrawRegion_GetClipMask(
                    TQ3XDrawRegion    drawRegion,
                    TQ3Bitmap         **clipMask);
```

`drawRegion` A draw region.

`clipMask` A bitmap that describes the clipping of the draw region's window.

DESCRIPTION

The `Q3XDrawRegion_GetClipMask` function returns, in the `clipMask` parameter, a bitmap that describes the clipping of the window for the partially exposed draw region `drawRegion`. In this bitmap, 1 bits indicate exposed pixels and 0 bits indicate occluded pixels.

On a Macintosh, additional window-system-specific information about clipping is available, in the form of a `RgnHandle` describing the clip region, or a `GDHandle`:

Q3XDrawRegion_GetClipRegion

The `Q3XDrawRegion_GetClipRegion` function lets you obtain a Mac OS `RgnHandle` value that describes the clip region of a partially exposed draw region.

```
TQ3Status Q3XDrawRegion_GetClipRegion(
                    TQ3XDrawRegion    drawRegion,
                    RgnHandle         *rgnHandle);
```

`drawRegion` A draw region.

`rgnHandle` A handle to a QuickDraw region.

DESCRIPTION

The `Q3XDrawRegion_GetClipRegion` function returns, in the `rgnHandle` parameter, a handle to a QuickDraw region that describes the clipping of the window for the partially exposed draw region `drawRegion`.

Q3XDrawRegion_GetGDHandle

The `Q3XDrawRegion_GetGDHandle` function lets you obtain a Mac OS `GDHandle` value that describes the clip region of a partially exposed draw region.

```
TQ3Status Q3XDrawRegion_GetGDHandle(
                    TQ3XDrawRegion    drawRegion,
                    GDHandle          *gdHandle);
```

`drawRegion` A draw region.

`gdHandle` A handle to a `gDevice` record.

DESCRIPTION

The `Q3XDrawRegion_GetGDHandle` function returns, in the `gdHandle` parameter, a handle to a `gDevice` record that describes the clipping of the window for the partially exposed draw region `drawRegion`. For further information about `gDevice` records, see *Inside Macintosh: QuickDraw Objects*.

Draw Region Location and Dimensions

QuickDraw 3D provides several routines to support renderers that are handling much of the draw region work themselves. They return the X-axis and Y-axis values of various sizes and offsets of the raster.

▲ **WARNING**
Putting functionality at this level of detail in a renderer is strongly discouraged. ▲

Q3XDrawRegion_GetDeviceScaleX

For a draw region, the `Q3XDrawRegion_GetDeviceScaleX` function lets you get the X-dimension size of the visible (onscreen) portion of the unclipped window.

```
TQ3Status Q3XDrawRegion_GetDeviceScaleX(
                    TQ3XDrawRegion    drawRegion,
                    float             *deviceScaleX);
```

`drawRegion` A draw region.

`deviceScaleX` The X-axis window size for the draw region.

DESCRIPTION

The `Q3XDrawRegion_GetDeviceScaleX` function returns, in the `deviceScaleX` parameter, the X-axis onscreen window dimension, ignoring clipping and occlusion, for the draw region designated by `drawRegion`.

Q3XDrawRegion_GetDeviceScaleY

For a draw region, the `Q3XDrawRegion_GetDeviceScaleY` function lets you get the Y-dimension size of the visible (onscreen) portion of the unclipped window.

```
TQ3Status Q3XDrawRegion_GetDeviceScaleY(
                    TQ3XDrawRegion    drawRegion,
                    float             *deviceScaleY);
```

`drawRegion` A draw region.

`deviceScaleY` The Y-axis window size for the draw region.

DESCRIPTION

The `Q3XDrawRegion_GetDeviceScaleY` function returns, in the `deviceScaleY` parameter, the Y-axis onscreen window dimension, ignoring clipping and occlusion, for the draw region designated by `drawRegion`.

Q3XDrawRegion_GetDeviceOffsetX

The `Q3XDrawRegion_GetDeviceOffsetX` function lets you get the X-axis window offset, relative to the monitor, for a draw region.

```
TQ3Status Q3XDrawRegion_GetDeviceOffsetX(
                    TQ3XDrawRegion    drawRegion,
                    float             *deviceOffsetX);
```

`drawRegion` A draw region.

`deviceOffsetX`
The X-axis window offset for the draw region.

DESCRIPTION

The `Q3XDrawRegion_GetDeviceOffsetX` function returns, in the `deviceOffsetX` parameter, the X-axis offset between the window and the monitor origin (upper left corner) for the draw region designated by `drawRegion`. With single buffering this value is the actual offset; with double buffering it is always 0.

Q3XDrawRegion_GetDeviceOffsetY

The `Q3XDrawRegion_GetDeviceOffsetY` function lets you get the Y-axis window offset, relative to the monitor, for a draw region.

```
TQ3Status Q3XDrawRegion_GetDeviceOffsetY(
                    TQ3XDrawRegion    drawRegion,
                    float             *deviceOffsetY);
```

`drawRegion` A draw region.

`deviceOffsetY`
The Y-axis window offset for the draw region.

DESCRIPTION

The `Q3XDrawRegion_GetDeviceOffsetY` function returns, in the `deviceOffsetY` parameter, the Y-axis offset between the window and the monitor origin (upper

left corner) for the draw region designated by `drawRegion`. With single buffering this value is the actual offset; with double buffering it is always 0.

Q3XDrawRegion_GetWindowScaleX

The `Q3XDrawRegion_GetWindowScaleX` function lets you get the X-axis window dimension for a draw region.

```
TQ3Status Q3XDrawRegion_GetWindowScaleX(
                    TQ3XDrawRegion    drawRegion,
                    float             *windowScaleX);
```

`drawRegion` A draw region.

`windowScaleX` The X-axis window dimension for the draw region.

DESCRIPTION

The `Q3XDrawRegion_GetWindowScaleX` function returns, in the `windowScaleX` parameter, the X-axis window dimension for the draw region designated by `drawRegion`. This dimension is the same with both single and double buffering.

Q3XDrawRegion_GetWindowScaleY

The `Q3XDrawRegion_GetWindowScaleY` function lets you get the Y-axis window dimension for a draw region.

```
TQ3Status Q3XDrawRegion_GetWindowScaleY(
                    TQ3XDrawRegion    drawRegion,
                    float             *windowScaleY);
```

`drawRegion` A draw region.

`windowScaleY` The Y-axis window dimension for the draw region.

DESCRIPTION

The `Q3XDrawRegion_GetWindowScaleY` function returns, in the `windowScaleY` parameter, the Y-axis window dimension for the draw region designated by `drawRegion`. This dimension is the same with both single and double buffering.

Q3XDrawRegion_GetWindowOffsetX

The `Q3XDrawRegion_GetWindowOffsetX` function lets you get the absolute X-axis window offset for a draw region.

```
TQ3Status Q3XDrawRegion_GetWindowOffsetX(
                    TQ3XDrawRegion    drawRegion,
                    float             *windowOffsetX);
```

`drawRegion` A draw region.

`windowOffsetX`
The X-axis window offset for the draw region.

DESCRIPTION

The `Q3XDrawRegion_GetWindowOffsetX` function returns, in the `windowOffsetX` parameter, the X-axis absolute offset of the window for the draw region designated by `drawRegion`. This value is the same with single and double buffering.

Q3XDrawRegion_GetWindowOffsetY

The `Q3XDrawRegion_GetWindowOffsetY` function lets you get the absolute Y-axis window offset for a draw region.

```
TQ3Status Q3XDrawRegion_GetWindowOffsetY(
                    TQ3XDrawRegion    drawRegion,
                    float             *windowOffsetY);
```

`drawRegion` A draw region.

`windowOffsetY`
The Y-axis window offset for the draw region.

DESCRIPTION

The `Q3XDrawRegion_GetWindowOffsetY` function returns, in the `windowOffsetY` parameter, the Y-axis absolute offset of the window for the draw region designated by `drawRegion`. This value is the same with single and double buffering.

Q3XDrawRegion_GetDeviceTransform

The `Q3XDrawRegion_GetDeviceTransform` function lets you get the device transform for a draw region.

```
TQ3Status Q3XDrawRegion_GetDeviceTransform(
                    TQ3XDrawRegion    drawRegion,
                    TQ3Matrix4x4      **deviceTransform);
```

`drawRegion` A draw region.

`deviceTransform`
The device transform for the draw region.

DESCRIPTION

The `Q3XDrawRegion_GetDeviceTransform` function returns, in the `deviceTransform` parameter, the device transform for the draw region designated by `drawRegion`.

Renderer-Private Data in Draw Regions

A renderer may attach private data to a draw region; for example, a Z-buffer renderer can attach its Z buffer to the draw region instead of maintaining it elsewhere.

To attach data, the renderer uses the `Q3XDrawRegion_SetRendererPrivate` function. It must provide an opaque pointer to the data (or data structure), as well as provide a method to be used by the draw region to dispose of the data;

this function is called when the draw region is itself disposed. Thus the renderer is responsible for providing the callback so that the attached data will properly be disposed of.

To retrieve data previously attached to a draw region, a renderer can use the `Q3XDrawRegion_GetRendererPrivate` function. Note that this merely returns the pointer provided in the `Q3XDrawRegion_SetRendererPrivate` call; it does not remove the need for a disposal callback.

Q3XDrawRegion_SetRendererPrivate

The `Q3XDrawRegion_SetRendererPrivate` function attaches renderer-private data to a draw region.

```
typedef void (*TQ3XDrawRegionRendererPrivateDeleteMethod)(
                    void    *rendererPrivate);

TQ3Status Q3XDrawRegion_SetRendererPrivate(
            TQ3XDrawRegion    drawRegion,
            const void        *rendererPrivate,
            TQ3XDrawRegionRendererPrivateDeleteMethod    deleteMethod);
```

`drawRegion` A draw region.

`rendererPrivate`
Pointer to private data.

`deleteMethod` Private data disposal method.

DESCRIPTION

The `Q3XDrawRegion_SetRendererPrivate` function attaches the private data pointed to by the `rendererPrivate` parameter to the draw region designated by the `drawRegion` parameter. It also registers the data disposal method designated by `deleteMethod`. The draw region will call this method to dispose of the private data when it is itself disposed of.

Q3XDrawRegion_GetRendererPrivate

The `Q3XDrawRegion_GetRendererPrivate` functionality lets a renderer retrieve private data that was previously attached to a draw region by using the `Q3XDrawRegion_SetRendererPrivate` routine.

```
TQ3Status Q3XDrawRegion_GetRendererPrivate(
                    TQ3XDrawRegion    drawRegion,
                    void              **rendererPrivate);
```

`drawRegion` A draw region.

`rendererPrivate`
On return, a pointer to the private data attached to the draw region designated by `drawRegion`.

DESCRIPTION

The `Q3XDrawRegion_GetRendererPrivate` function returns, in the `rendererPrivate` parameter, the pointer to private data that was passed in a previous call to `Q3XDrawRegion_SetRendererPrivate`.

Renderer Errors

Renderer routines may return the following errors. A list of general QuickDraw 3D errors is given in "QuickDraw 3D Errors, Warnings, and Notices" (page 87).

```
kQ3ErrorUnknownStudioType
kQ3ErrorAlreadyRendering
kQ3ErrorStartGroupRange
kQ3ErrorUnsupportedGeometryType
kQ3ErrorInvalidGeometryType
kQ3ErrorUnsupportedFunctionality
kQ3WarningFunctionalityNotSupported
```

CHAPTER 12

Draw Context Objects

This chapter describes draw context objects (or draw contexts) and the functions you can use to manipulate them. You use draw contexts to connect your application to a specific drawing destination, such as a window system. For example, to draw into a Mac OS window, you create an instance of a Macintosh draw context object and attach it to a view.

To use this chapter, you should already be familiar with the QuickDraw 3D class hierarchy, described in the chapter "QuickDraw 3D Objects" earlier in this book. For information about attaching a draw context to a view, see the chapter "View Objects" in this book. You do not, however, need to know how to create or manipulate views to read this chapter.

This chapter begins by describing draw contexts and their features. Then it shows how to configure the settings of a draw context object. The section "Draw Context Objects Reference," beginning on page 843 provides a complete description of draw context objects and the routines you can use to create and manipulate them.

About Draw Context Objects

The QuickDraw 3D graphics library is able to direct its output—a rendered image—into one or more destinations (hereafter called its **drawing destinations**). For instance, you can use QuickDraw 3D to draw three-dimensional images into a standard Mac OS window.To achieve this cross-platform drawing capability, and thereby to insulate most of the application programming interfaces from details of the underlying drawing destination, QuickDraw 3D uses objects called draw context objects. A **draw context object** (or, more briefly, a **draw context**) is a QuickDraw 3D object that maintains information specific to a particular window system or drawing destination.

In general, QuickDraw 3D does not duplicate existing methods of creating, handling user actions in, or manipulating drawing destinations. For example, QuickDraw 3D does not provide any means of creating a Mac OS window, handling events in the window, or modifying the size or location of the window. A QuickDraw 3D draw context, which provides a link between your application and the Mac OS window, simply contains the minimum amount of information it needs to draw into the window. You must use the Window Manager for all other operations on a Mac OS window.

A draw context is of type `TQ3DrawContextObject`, which is a subtype of shared object. You need to create an instance of a specific type of draw context object and then attach it to a view, usually by calling `Q3View_SetDrawContext`. QuickDraw 3D currently supports these types of draw contexts:

- Macintosh draw contexts
- pixmap draw contexts
- Microsoft Windows draw contexts

Not all drawing destinations are windows. QuickDraw 3D supports the pixmap draw context for drawing an image into an arbitrary region of memory (that is, a pixmap). You can, if necessary, even create instances of several kinds of draw contexts and draw the same scene into several different kinds of windows.

All draw contexts share a set of basic properties, which are maintained in a structure of type `TQ3DrawContextData`.

```
typedef struct TQ3DrawContextData {
    TQ3DrawContextClearImageMethod      clearImageMethod;
    TQ3ColorARGB                        clearImageColor;
    TQ3Area                             pane;
    TQ3Boolean                          paneState;
    TQ3Bitmap                           mask;
    TQ3Boolean                          maskState;
    TQ3Boolean                          doubleBufferState;
} TQ3DrawContextData;
```

IMPORTANT

Windows 32 draw contexts are always implicitly double buffered and Direct Draw surface draw contexts are single buffered, regardless of the value of `doubleBufferState`. ▲

The `TQ3DrawContextData` fields define the manner in which a window (or region of memory) is cleared, the size of the destination drawing pane, the drawing mask, and the state of the double buffering. These basic properties are designed to be independent of any particular window system. You can rely on the capabilities provided by these properties across window systems, whether or not the drawing destination supports them.

Note
Not all the basic properties maintained in the `TQ3DrawContextData` data structure are supported by all draw contexts. For example, it makes no sense to use double buffering when drawing into a pixmap. ◆

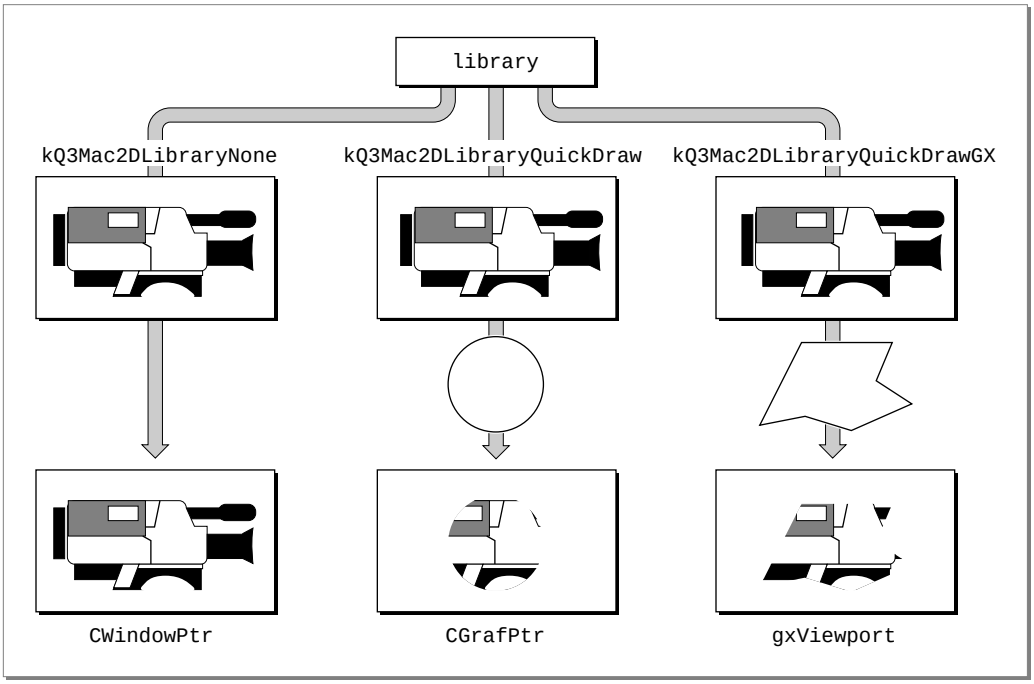
In addition to these basic properties that are common to all draw contexts, each specific type of draw context defines context-specific properties. For example, the Macintosh draw context maintains information about the window into which QuickDraw 3D is to draw, the optional use of a two-dimensional graphics library (QuickDraw or QuickDraw GX), and so forth. The following sections describe the specific draw context types.

Macintosh Draw Contexts

A **Macintosh draw context** is a draw context associated with a Mac OS window. You specify a Mac OS window by providing a pointer to a window (of type `CWindowPtr`) which defines the area into which QuickDraw 3D will draw images of rendered models. In addition, you can attach to a Macintosh draw context either a QuickDraw color graphics port (of type `CGrafPort`) or a QuickDraw GX view port (of type `gxViewPort`). Using this optional two-dimensional graphics library, you can achieve special effects such as clipping, dithering, and geometrical transforms of the image. At most one 2D graphics library can be associated with a Macintosh draw context at one time, and you are responsible for initializing the graphics library and performing any other required set-up.

QuickDraw 3D cannot use a two-dimensional graphics library unless the draw context is configured for double buffering and the active buffer is set to the back buffer. QuickDraw and QuickDraw GX effects are applied when the front buffer is updated from the back buffer. Figure 12-1 illustrates the three possibilities for drawing in a Macintosh draw context. You can use QuickDraw to set a clip region (middle possibility) and QuickDraw GX to set a clip shape (right possibility).

Figure 12-1 Using a two-dimensional graphics library in a Macintosh draw context



Pixmap Draw Contexts

A **pixmap draw context** is a draw context associated with a pixmap, that is, a region of memory not directly associated with a window. The two-dimensional image produced by the renderer is simply written into that memory region.

Note
See the chapter "Geometric Objects" for information on the structure of pixmaps. ◆

To draw an image into an offscreen graphics world (pointed to by a variable of type `GWorldPtr`), for instance, you need to (1) create the offscreen graphics world using standard QuickDraw routines, (2) call `LockPixels` to lock the pixels in memory, and (3) create a pixmap draw context in which the address of the pixmap is the pointer returned by the `GetPixBaseAddr` function. You need to lock

the pixmap in memory because QuickDraw 3D routines may move or purge memory.

Note
See the book *Inside Macintosh: Imaging With QuickDraw* for complete information about offscreen graphics worlds. ◆

You can update a window without rendering to it by rendering to an offscreen graphics world and then copying the data to the window.

Windows Draw Contexts

QuickDraw 3D provides routines that you can use to create and manipulate Windows 32 draw contexts. See "Managing Windows 32 Draw Contexts," beginning on page 864, for details.

Using Draw Context Objects

QuickDraw 3D supplies routines that you can use to create and configure draw context objects. This section describes how to accomplish these tasks.

Creating and Configuring a Draw Context

You create a draw context object by calling a constructor function such as `Q3MacDrawContext_New` or `Q3PixMapDrawContext_New`. These functions take as a parameter a pointer to a data structure that contains information about the draw context you want to create. For example, you pass the `Q3MacDrawContext_New` function a pointer to a structure of type `TQ3MacDrawContextData`, defined as follows:

```
typedef struct TQ3MacDrawContextData {
    TQ3DrawContextData                  drawContextData;
    CWindowPtr                          window;
    TQ3MacDrawContext2DLibrary          library;
    gxViewPort                          viewPort;
    CGrafPtr                            grafPort;
} TQ3MacDrawContextData;
```

The first field is just a draw context data structure that contains basic information about the draw context (see page 838). The remaining fields contain specific information about the Mac OS window and 2D graphics library associated with the draw context.

See Listing 1-7 (page 65) for a sample routine that creates a Macintosh draw context.

Using Double Buffering

In general, when drawing to a screen or other device visible by the user, you'll want to use QuickDraw 3D's double buffering capability to reduce the amount of flicker that occurs when the image on the screen is updated. You enable double buffering by calling `Q3DrawContext_SetDoubleBufferState` or by setting the `doubleBufferState` field of a draw context data structure to `kQ3True` before calling the draw context constructor method.

Note
In general, QuickDraw 3D will take advantage of any double buffering capabilities available on the target window system. ◆

When double buffering is active for a draw context, the draw context is associated with two buffers, the front buffer and the back buffer. The front buffer is the area of memory that is being displayed on the screen. The back buffer is some other area of memory that has the same size as the front buffer.

When double buffering is active, all drawing (as performed by routines such as `Q3Group_Submit` in a rendering loop) is done into the back buffer, and the front buffer is updated only after the call to `Q3View_EndRendering` on the final pass through your rendering loop. Some renderers (especially those that rely on hardware accelerators) may return control to your application before the image on the screen has been updated. You can call the `Q3View_Sync` function to block execution until the renderer is done drawing in the screen's draw context. You might want to do this if you intend to grab the image on the screen or if you intend to allow the user to pick objects displayed on the screen. See the chapter "Renderer Objects" for complete information about calling `Q3View_Sync`.

Draw Context Objects Reference

This section describes the QuickDraw 3D data structures and routines that you can use to manage drawing contexts.

Data Structures

QuickDraw 3D provides data structures that you can use to define draw contexts.

Draw Context Data Structure

QuickDraw 3D defines the **draw context data structure** to maintain information that is common to all the supported draw contexts. The draw context data structure is defined by the `TQ3DrawContextData` data type.

```
typedef struct TQ3DrawContextData {
    TQ3DrawContextClearImageMethod      clearImageMethod;
    TQ3ColorARGB                        clearImageColor;
    TQ3Area                             pane;
    TQ3Boolean                          paneState;
    TQ3Bitmap                           mask;
    TQ3Boolean                          maskState;
    TQ3Boolean                          doubleBufferState;
} TQ3DrawContextData;
```

Field descriptions

`clearImageMethod` A constant that indicates how the drawing destination should be cleared. You can use these constants to specify a method to clear the image.

```
typedef enum TQ3DrawContextClearImageMethod {
    kQ3ClearMethodNone,
    kQ3ClearMethodWithColor,
} TQ3DrawContextClearImageMethod;
```

	The constant `kQ3ClearMethodNone` indicates that the drawing destination should not be cleared. The exact behavior when `Q3View_StartRendering` is called is renderer-dependent. For example, some renderers expect to redraw every pixel in the drawing destination. By specifying `kQ3ClearMethodNone`, you allow those renderers to apply optimizations during rendering. The constant `kQ3ClearMethodWithColor` indicates that the drawing destination should be cleared with the color specified in the `clearImageColor` field.
`clearImageColor`	The color to be used when clearing the drawing destination with a color. This field is ignored unless the value in the `clearImageMethod` field is `kQ3ClearMethodWithColor`.
`pane`	The rectangular area (specified in window coordinates) in the drawing destination within which all drawing occurs. If the output pane is smaller than the window's port rectangle, the image is scaled (not clipped) to fit into the pane.
`paneState`	A Boolean value that determines whether the area specified in the `pane` field is to be used (`kQ3True`) or is to be ignored (`kQ3False`). Set this field to `kQ3False` to use the entire window as the output pane. If this field is set to `kQ3True`, the `pane` field must contain a valid area.
`mask`	A bitmap that is used to mask out certain portions of the drawing destination. Each bit in the bitmap corresponds to a pixel in the drawing area. If a bit is set, the corresponding pixel is drawn; if a bit is clear, the corresponding pixel is not drawn. If the value in this field is `NULL`, the entire window is used as the clipping region.
`maskState`	A Boolean value that determines whether the mask specified in the `mask` field is to be used (`kQ3True`) or is to be ignored (`kQ3False`). If this field is set to `kQ3True`, the `mask` field must contain a valid bitmap.
`doubleBufferState`	A Boolean value that determines whether double buffering is to used for the drawing destination (`kQ3True`) or not (`kQ3False`). When double buffering is enabled, the back buffer is the active buffer.

Macintosh Draw Context Structure

QuickDraw 3D defines the **Macintosh draw context data structure** to maintain information that is specific to Macintosh draw contexts. The Macintosh draw context data structure is defined by the `TQ3MacDrawContextData` data type.

```
typedef struct TQ3MacDrawContextData {
    TQ3DrawContextData                  drawContextData;
    CWindowPtr                          window;
    TQ3MacDrawContext2DLibrary          library;
    gxViewPort                          viewPort;
    CGrafPtr                            grafPort;
} TQ3MacDrawContextData;
```

Field descriptions

`drawContextData` A draw context data structure defining basic information about the draw context.

`window` A pointer to a window.

`library` The two-dimensional graphics library to use when rendering an image. You can use these constants to specify a 2D graphics library:

```
typedef enum TQ3MacDrawContext2DLibrary {
    kQ3Mac2DLibraryNone,
    kQ3Mac2DLibraryQuickDraw,
    kQ3Mac2DLibraryQuickDrawGX
} TQ3MacDrawContext2DLibrary;
```

The constants `kQ3Mac2DLibraryQuickDraw` and `kQ3Mac2DLibraryQuickDrawGX` indicate that the renderer should use QuickDraw or QuickDraw GX, respectively, in the final stage of rendering. Either the `viewPort` or the `grafPort` field must contain a non-null value if QuickDraw or QuickDraw GX is to be used. The two-dimensional library is used only when copying from the back to the front buffer, never when drawing directly to the front buffer.

`viewPort` A view port, as defined by QuickDraw GX. See the book *Inside Macintosh: QuickDraw GX Objects* for complete information about view ports.

`grafPort` A graphics port, as defined by QuickDraw. See the book *Inside Macintosh: Imaging With QuickDraw* for complete information about graphics ports.

Pixmap Draw Context Structure

QuickDraw 3D defines the **pixmap draw context data structure** to maintain information that is specific to pixmap draw contexts. The pixmap draw context data structure is defined by the `TQ3PixmapDrawContextData` data type.

```
typedef struct TQ3PixmapDrawContextData {
    TQ3DrawContextData                  drawContextData;
    TQ3Pixmap                           pixmap;
} TQ3PixmapDrawContextData;
```

Field descriptions

`drawContextData` A draw context data structure defining basic information about the draw context.

`pixmap` A pixmap (that is, a pixel map in memory). This pixmap is assumed to have a pixel size of 24 bits.

Windows 32 Draw Context Structure

QuickDraw 3D defines the **Windows 32 draw context data structure** to maintain information that is specific to Windows 32 draw contexts. The Windows 32 draw context data structure is defined by the `TQ3Win32DCDrawContextData` data type.

```
typedef struct TQ3Win32DCDrawContextData {
    HDC                                 hdc;
} TQ3Win32DCDrawContextData;
```

Field descriptions

`hdc` Microsoft Windows draw context (obtained from a window using the Windows `GetDC` function).

Direct Draw Surface Draw Context Structure

QuickDraw 3D defines the **direct draw surface draw context data structure** to maintain information that is specific to Windows direct draw surface draw contexts. The Windows direct draw surface draw context data structure is defined by the `TQ3DDSurfaceDrawContextData` data type.

```
typedef enum {
    kQ3DirectDrawObject     = 1,
    kQ3DirectDrawObject2    = 2
} TQ3DirectDrawObjectSelector;

typedef enum {
    kQ3DirectDrawSurface    = 1,
    kQ3DirectDrawSurface2   = 2
} TQ3DirectDrawSurfaceSelector;

typedef struct TQ3DDSurfaceDescriptor {
    TQ3DirectDrawObjectSelector     objectSelector;
    union
    {
        LPDIRECTDRAW    lpDirectDraw;
        LPDIRECTDRAW2   lpDirectDraw2;
    };
    TQ3DirectDrawSurfaceSelector    surfaceSelector;
    union
    {
        LPDIRECTDRAWSURFACE     lpDirectDrawSurface;
        LPDIRECTDRAWSURFACE2    lpDirectDrawSurface2;
    };
} TQ3DDSurfaceDescriptor;

typedef struct TQ3DDSurfaceDrawContextData {
    TQ3DrawContextData          drawContextData;
    TQ3DDSurfaceDescriptor      ddSurfaceDescriptor;
} TQ3DDSurfaceDrawContextData;
```

Field descriptions

`objectSelector` A Direct Draw object selector that specifies whether the caller is using version 1 or version 2 Direct Draw objects.

`lpDirectDraw` Direct Draw context data.

`surfaceSelector` A Direct Draw surface selector that specifies whether the caller is using version 1 or version 2 Direct Draw surfaces.

`lpDirectDrawSurface`
Direct Draw surface context data.

`drawContextData` A draw context data structure defining basic information about the draw context.

ddSurfaceDescriptor
The union of an object selector and a surface selector.

Data for the `lpDirectDraw` and `lpDirectDrawSurface` fields can be obtained within Microsoft Windows by calling `IDirectDraw::CreateSurface` or `IDirectDraw::GetGDISurface`.

Draw Context Objects Routines

This section describes routines you can use to manage draw contexts.

Managing Draw Contexts

QuickDraw 3D provides a number of general routines for operating with draw context objects.

Q3DrawContext_GetType

You can use the `Q3DrawContext_GetType` function to get the type of a draw context.

```
TQ3ObjectType Q3DrawContext_GetType (TQ3DrawContextObject drawContext);
```

`drawContext` A draw context object.

DESCRIPTION

The `Q3DrawContext_GetType` function returns, as its function result, the type of the draw context specified by the `drawContext` parameter. The types of draw contexts currently supported by QuickDraw 3D are defined by these constants:

```
kQ3DrawContextTypeMacintosh
kQ3DrawContextTypePixmap
kQ3DrawContextTypeWin32DC
kQ3DrawContextTypeDDSurface
```

Q3DrawContext_GetData

You can use the `Q3DrawContext_GetData` function to get the data associated with a draw context.

```
TQ3Status Q3DrawContext_GetData (
                    TQ3DrawContextObject context,
                    TQ3DrawContextData *contextData);
```

`context` A draw context object.

`contextData` On exit, a pointer to a draw context data structure.

DESCRIPTION

The `Q3DrawContext_GetData` function returns, in the `contextData` parameter, a pointer to a draw context data structure for the draw context specified by the `context` parameter.

Q3DrawContext_SetData

You can use the `Q3DrawContext_SetData` function to set the data associated with a draw context.

```
TQ3Status Q3DrawContext_SetData (
                    TQ3DrawContextObject context,
                    const TQ3DrawContextData *contextData);
```

`context` A draw context object.

`contextData` A pointer to a draw context data structure.

DESCRIPTION

The `Q3DrawContext_SetData` function sets the data associated with the draw context specified by the `context` parameter to that specified in the draw context data structure pointed to by the `contextData` parameter.

Q3DrawContext_GetClearImageColor

You can use the `Q3DrawContext_GetClearImageColor` function to get the image clearing color of a draw context.

```
TQ3Status Q3DrawContext_GetClearImageColor (
                    TQ3DrawContextObject context,
                    TQ3ColorARGB *color);
```

`context` A draw context object.

`color` On exit, the current image clearing color of the specified draw context.

DESCRIPTION

The `Q3DrawContext_GetClearImageColor` function returns, in the `color` parameter, a constant that indicates the current image clearing color for the draw context specified by the `context` parameter.

Q3DrawContext_SetClearImageColor

You can use the `Q3DrawContext_SetClearImageColor` function to set the image clearing color of a draw context.

```
TQ3Status Q3DrawContext_SetClearImageColor (
                    TQ3DrawContextObject context,
                    const TQ3ColorARGB *color);
```

`context` A draw context object.

`color` The desired image clearing color of the specified draw context.

DESCRIPTION

The `Q3DrawContext_SetClearImageColor` function sets the image clearing color of the draw context specified by the `context` parameter to the value specified in the `color` parameter.

Q3DrawContext_GetPane

You can use the `Q3DrawContext_GetPane` function to get the pane of a draw context.

```
TQ3Status Q3DrawContext_GetPane (
                    TQ3DrawContextObject context,
                    TQ3Area *pane);
```

`context` A draw context object.

`pane` On exit, the area in the specified draw context in which all drawing occurs.

DESCRIPTION

The `Q3DrawContext_GetPane` function returns, in the `pane` parameter, the area in the draw context specified by the `context` parameter in which all drawing occurs.

Q3DrawContext_SetPane

You can use the `Q3DrawContext_SetPane` function to set the pane of a draw context.

```
TQ3Status Q3DrawContext_SetPane (
                    TQ3DrawContextObject context,
                    const TQ3Area *pane);
```

context	A draw context object.
pane	The area in the specified draw context in which all drawing should occur.

DESCRIPTION

The Q3DrawContext_SetPane function sets the area of the draw context specified by the context parameter within which all drawing is to occur to the area specified in the pane parameter.

Q3DrawContext_GetPaneState

You can use the Q3DrawContext_GetPaneState function to get the pane state of a draw context.

```
TQ3Status Q3DrawContext_GetPaneState (
                    TQ3DrawContextObject context,
                    TQ3Boolean *state);
```

context	A draw context object.
state	On exit, the current pane state of the specified draw context.

DESCRIPTION

The Q3DrawContext_GetPaneState function returns, in the state parameter, a Boolean value that determines whether the pane associated with the draw context specified by the context parameter is to be used (kQ3True) or not (kQ3False).

Q3DrawContext_SetPaneState

You can use the Q3DrawContext_SetPaneState function to set the pane state of a draw context.

```
TQ3Status Q3DrawContext_SetPaneState (
                    TQ3DrawContextObject context,
                    TQ3Boolean state);
```

`context` A draw context object.

`state` The desired pane state of the specified draw context.

DESCRIPTION

The `Q3DrawContext_SetPaneState` function sets the pane state of the draw context specified by the `context` parameter to the value specified in the `state` parameter. If the value of `state` is `kQ3True`, the pane associated with that draw context is to be used; if `kQ3False`, the pane is not used.

Q3DrawContext_GetClearImageMethod

You can use the `Q3DrawContext_GetClearImageMethod` function to get the image clearing method of a draw context.

```
TQ3Status Q3DrawContext_GetClearImageMethod (
                    TQ3DrawContextObject context,
                    TQ3DrawContextClearImageMethod *method);
```

`context` A draw context object.

`method` On exit, the current image clearing method of the specified draw context. See page 843 for the values that can be returned in this parameter.

DESCRIPTION

The `Q3DrawContext_GetClearImageMethod` function returns, in the `method` parameter, a constant that indicates the current image clearing method for the draw context specified by the `context` parameter.

Q3DrawContext_SetClearImageMethod

You can use the `Q3DrawContext_SetClearImageMethod` function to set the image clearing method of a draw context.

```
TQ3Status Q3DrawContext_SetClearImageMethod (
                    TQ3DrawContextObject context,
                    TQ3DrawContextClearImageMethod method);
```

`context` A draw context object.

`method` The desired image clearing method of the specified draw context. See page 843 for the values that can be passed in this parameter.

DESCRIPTION

The `Q3DrawContext_SetClearImageMethod` function sets the image clearing method of the draw context specified by the `context` parameter to the value specified in the `method` parameter.

Q3DrawContext_GetMask

You can use the `Q3DrawContext_GetMask` function to get the mask of a draw context.

```
TQ3Status Q3DrawContext_GetMask (
                    TQ3DrawContextObject context,
                    TQ3Bitmap *mask);
```

`context` A draw context object.

`mask` On exit, the mask of the specified draw context.

DESCRIPTION

The `Q3DrawContext_GetMask` function returns, in the `mask` parameter, the current mask for the draw context specified by the `context` parameter. The mask is a bitmap whose bits determine whether or not corresponding pixels in the

drawing destination are drawn or are masked out. `Q3DrawContext_GetMask` allocates memory internally for the returned bitmap; when you're done using the bitmap, you should call the `Q3Bitmap_Empty` function to dispose of that memory.

Q3DrawContext_SetMask

You can use the `Q3DrawContext_SetMask` function to set the mask of a draw context.

```
TQ3Status Q3DrawContext_SetMask (
                    TQ3DrawContextObject context,
                    const TQ3Bitmap *mask);
```

`context` A draw context object.

`mask` The desired mask of the specified draw context.

DESCRIPTION

The `Q3DrawContext_SetMask` function sets the mask of the draw context specified by the `context` parameter to the bitmap specified in the `mask` parameter. `Q3DrawContext_SetMask` copies the bitmap to internal QuickDraw 3D memory, so you can dispose of the specified bitmap after calling `Q3DrawContext_SetMask`.

Q3DrawContext_GetMaskState

You can use the `Q3DrawContext_GetMaskState` function to get the mask state of a draw context.

```
TQ3Status Q3DrawContext_GetMaskState (
                    TQ3DrawContextObject context,
                    TQ3Boolean *state);
```

`context` A draw context object.

`state` On exit, the current mask state of the specified draw context.

DESCRIPTION

The `Q3DrawContext_GetMaskState` function returns, in the `state` parameter, a Boolean value that determines whether the mask associated with the draw context specified by the `context` parameter is to be used (`kQ3True`) or not (`kQ3False`).

Q3DrawContext_SetMaskState

You can use the `Q3DrawContext_SetMaskState` function to set the mask state of a draw context.

```
TQ3Status Q3DrawContext_SetMaskState (
                    TQ3DrawContextObject context,
                    TQ3Boolean state);
```

`context` A draw context object.

`state` The desired mask state of the specified draw context.

DESCRIPTION

The `Q3DrawContext_SetMaskState` function sets the mask state of the draw context specified by the `context` parameter to the value specified in the `state` parameter. Set `state` to `kQ3True` if you want the mask enabled and to `kQ3False` otherwise.

Q3DrawContext_GetDoubleBufferState

You can use the `Q3DrawContext_GetDoubleBufferState` function to get the double buffer state of a draw context.

```
TQ3Status Q3DrawContext_GetDoubleBufferState (
                    TQ3DrawContextObject context,
                    TQ3Boolean *state);
```

`context`	A draw context object.
`state`	On exit, the current mask state of the specified draw context.

DESCRIPTION

The `Q3DrawContext_GetDoubleBufferState` function returns, in the `state` parameter, a Boolean value that determines whether double buffering is enabled for the draw context specified by the `context` parameter (`kQ3True`) or not (`kQ3False`).

Q3DrawContext_SetDoubleBufferState

You can use the `Q3DrawContext_SetDoubleBufferState` function to set the double buffer state of a draw context.

```
TQ3Status Q3DrawContext_SetDoubleBufferState (
                    TQ3DrawContextObject context,
                    TQ3Boolean state);
```

`context`	A draw context object.
`state`	The desired mask state of the specified draw context.

DESCRIPTION

The `Q3DrawContext_SetDoubleBufferState` function sets the double buffer state of the draw context specified by the `context` parameter to the value specified in the `state` parameter. Set `state` to `kQ3True` if you want the double buffering enabled and to `kQ3False` otherwise. When you enable double buffering, the active buffer is the back buffer.

Managing Macintosh Draw Contexts

QuickDraw 3D provides routines that you can use to create and manipulate Macintosh draw contexts.

Q3MacDrawContext_New

You can use the `Q3MacDrawContext_New` function to create a new Macintosh draw context.

```
TQ3DrawContextObject Q3MacDrawContext_New (
                     const TQ3MacDrawContextData *drawContextData);
```

`drawContextData`
A pointer to a Macintosh draw context data structure.

DESCRIPTION

The `Q3MacDrawContext_New` function returns, as its function result, a new draw context object having the characteristics specified by the `drawContextData` parameter. See "Macintosh Draw Context Structure" (page 845) for information on the `drawContextData` parameter.

Q3MacDrawContext_GetWindow

You can use the `Q3MacDrawContext_GetWindow` function to get the window associated with a Macintosh draw context.

```
TQ3Status Q3MacDrawContext_GetWindow (
                     TQ3DrawContextObject drawContext,
                     CWindowPtr *window);
```

`drawContext` A Macintosh draw context object.

`window` On exit, a pointer to a window.

DESCRIPTION

The `Q3MacDrawContext_GetWindow` function returns, in the `window` parameter, a pointer to the window currently associated with the draw context specified by the `drawContext` parameter.

Q3MacDrawContext_SetWindow

You can use the `Q3MacDrawContext_SetWindow` function to set the window associated with a Macintosh draw context.

```
TQ3Status Q3MacDrawContext_SetWindow (
                    TQ3DrawContextObject drawContext,
                    const CWindowPtr window);
```

`drawContext` A Macintosh draw context object.

`window` A pointer to a window.

DESCRIPTION

The `Q3MacDrawContext_SetWindow` function sets the window associated with the draw context specified by the `drawContext` parameter to the window specified by the `window` parameter.

Q3MacDrawContext_Get2DLibrary

You can use the `Q3MacDrawContext_Get2DLibrary` function to get the two-dimensional drawing library associated with a Macintosh draw context.

```
TQ3Status Q3MacDrawContext_Get2DLibrary (
                    TQ3DrawContextObject drawContext,
                    TQ3MacDrawContext2DLibrary *library);
```

`drawContext` A Macintosh draw context object.

`library` On exit, a constant that specifies the two-dimensional graphics library used when rendering an image in the specified draw context. See page 845 for the values that can be returned in this field.

DESCRIPTION

The `Q3MacDrawContext_Get2DLibrary` function returns, in the `library` parameter, the two-dimensional drawing library currently associated with the draw context specified by the `drawContext` parameter.

Q3MacDrawContext_Set2DLibrary

You can use the `Q3MacDrawContext_Set2DLibrary` function to set the two-dimensional drawing library associated with a Macintosh draw context.

```
TQ3Status Q3MacDrawContext_Set2DLibrary (
                    TQ3DrawContextObject drawContext,
                    TQ3MacDrawContext2DLibrary library);
```

`drawContext` A Macintosh draw context object.

`library` A constant that specifies the desired two-dimensional graphics library to be used when rendering an image in the specified draw context. See page 845 for the values that can be passed in this field.

DESCRIPTION

The `Q3MacDrawContext_Set2DLibrary` function sets the two-dimensional drawing library associated with the draw context specified by the `drawContext` parameter to the library specified by the `library` parameter.

Q3MacDrawContext_GetGXViewPort

You can use the `Q3MacDrawContext_GetGXViewPort` function to get the QuickDraw GX view port associated with a Macintosh draw context.

```
TQ3Status Q3MacDrawContext_GetGXViewPort (
                    TQ3DrawContextObject drawContext,
                    gxViewPort *viewPort);
```

drawContext	A Macintosh draw context object.
viewPort	On exit, the QuickDraw GX view port currently associated with the specified draw context.

DESCRIPTION

The Q3MacDrawContext_GetGXViewPort function returns, in the viewPort parameter, the QuickDraw GX view port currently associated with the draw context specified by the drawContext parameter. If no view port is associated with the draw context or the two-dimensional graphics library is not set to kQ3Mac2DLibraryQuickDrawGX, Q3MacDrawContext_GetGXViewPort returns NULL in the viewPort parameter.

Q3MacDrawContext_SetGXViewPort

You can use the Q3MacDrawContext_SetGXViewPort function to set the QuickDraw GX view port associated with a Macintosh draw context.

```
TQ3Status Q3MacDrawContext_SetGXViewPort (
                    TQ3DrawContextObject drawContext,
                    const gxViewPort viewPort);
```

drawContext	A Macintosh draw context object.
viewPort	The QuickDraw GX view port to be associated with the specified draw context.

DESCRIPTION

The Q3MacDrawContext_SetGXViewPort function sets the QuickDraw GX view port associated with the draw context specified by the drawContext parameter to the view port specified by the viewPort parameter. The two-dimensional graphics library associated with the specified draw context must be kQ3Mac2DLibraryQuickDrawGX.

Q3MacDrawContext_GetGrafPort

You can use the `Q3MacDrawContext_GetGrafPort` function to get the QuickDraw graphics port associated with a Macintosh draw context.

```
TQ3Status Q3MacDrawContext_GetGrafPort (
                    TQ3DrawContextObject drawContext,
                    CGrafPtr *grafPort);
```

`drawContext` A Macintosh draw context object.

`grafPort` On exit, the QuickDraw graphics port currently associated with the specified draw context.

DESCRIPTION

The `Q3MacDrawContext_GetGrafPort` function returns, in the `grafPort` parameter, the QuickDraw graphics port currently associated with the draw context specified by the `drawContext` parameter. If no graphics port is associated with the draw context or the two-dimensional graphics library is not `kQ3Mac2DLibraryQuickDraw`, `Q3MacDrawContext_GetGrafPort` returns `NULL` in the `grafPort` parameter.

Q3MacDrawContext_SetGrafPort

You can use the `Q3MacDrawContext_SetGrafPort` function to set the QuickDraw graphics port associated with a Macintosh draw context.

```
TQ3Status Q3MacDrawContext_SetGrafPort (
                    TQ3DrawContextObject drawContext,
                    const CGrafPtr grafPort);
```

`drawContext` A Macintosh draw context object.

`grafPort` The QuickDraw graphics port to be associated with the specified draw context.

DESCRIPTION

The `Q3MacDrawContext_SetGrafPort` function sets the QuickDraw graphics port associated with the draw context specified by the `drawContext` parameter to the graphics port specified by the `grafPort` parameter. The two-dimensional graphics library associated with the specified draw context must be `kQ3Mac2DLibraryQuickDraw`.

Managing Pixmap Draw Contexts

QuickDraw 3D provides routines that you can use to create and manipulate pixmap draw contexts.

Q3PixmapDrawContext_New

You can use the `Q3PixmapDrawContext_New` function to create a new pixmap draw context.

```
TQ3DrawContextObject Q3PixmapDrawContext_New (
                     const TQ3PixmapDrawContextData *contextData);
```

`contextData` A pointer to a pixmap draw context data structure.

DESCRIPTION

The `Q3PixmapDrawContext_New` function returns, as its function result, a new draw context object having the characteristics specified by the `contextData` parameter.

Q3PixmapDrawContext_GetPixmap

You can use the `Q3PixmapDrawContext_GetPixmap` function to get the pixmap associated with a pixmap draw context.

```
TQ3Status Q3PixmapDrawContext_GetPixmap (
                    TQ3DrawContextObject drawContext,
                    TQ3Pixmap *pixmap);
```

`drawContext` A pixmap draw context object.

`pixmap` On exit, a pointer to a pixmap.

DESCRIPTION

The `Q3PixmapDrawContext_GetPixmap` function returns, in the `pixmap` parameter, a pointer to the pixmap currently associated with the draw context specified by the `drawContext` parameter.

Q3PixmapDrawContext_SetPixmap

You can use the `Q3PixmapDrawContext_SetPixmap` function to set the pixmap associated with a pixmap draw context.

```
TQ3Status Q3PixmapDrawContext_SetPixmap (
                    TQ3DrawContextObject drawContext,
                    const TQ3Pixmap *pixmap);
```

`drawContext` A pixmap draw context object.

`pixmap` A pointer to a pixmap.

DESCRIPTION

The `Q3PixmapDrawContext_SetPixmap` function sets the pixmap associated with the draw context specified by the `drawContext` parameter to the pixmap specified by the `pixmap` parameter.

Managing Windows 32 Draw Contexts

QuickDraw 3D provides routines that you can use to create and manipulate Windows 32 draw contexts.

Note
QuickDraw 3D both locks and clears the Direct Draw surface before each rendering loop, so your software need not perform these operations for a surface during rendering. ◆

Note
Windows 32 draw contexts are always implicitly double buffered. ◆

Q3Win32DCDrawContext_New

You can use the `Q3Win32DCDrawContext_New` function to create a new Windows 32 draw context object.

```
TQ3DrawContextObject Q3Win32DCDrawContext_New  (
                     const TQ3Win32DCDrawContextData *drawContextData);
```

`drawContextData`
Pointer to a Windows 32 draw context data structure.

return value A Windows 32 draw context object.

DESCRIPTION

`Q3Win32DCDrawContext_New` returns a Windows 32 draw context object if it is successful; otherwise it returns `NULL`. The application must set up the necessary data structure, as described in "Windows 32 Draw Context Structure," beginning on page 846.

Q3Win32DCDrawContext_GetDC

You can use the `Q3Win32DCDrawContext_GetDC` function to retrieve the Microsoft Windows draw context associated with a Windows 32 draw context object.

```
TQ3Status Q3Win32DCDrawContext_GetDC (
                    TQ3DrawContextObject drawContext,
                    HDC *hdc);
```

`drawContext` A draw context object.

`hdc` Pointer to a Microsoft Windows draw context.

DESCRIPTION

`Q3Win32DCDrawContext_GetDC` returns in a structure pointed to by `hdc` the Microsoft Windows draw context that is associated with the draw context object specified by `drawContext`.

Q3Win32DCDrawContext_SetDC

You can use the `Q3Win32DCDrawContext_SetDC` function to set the Microsoft Windows draw context for a Windows 32 draw context object.

```
TQ3Status Q3Win32DCDrawContext_SetDC (
                    TQ3DrawContextObject drawContext,
                    const HDC hdc);
```

`drawContext` A draw context object.

`hdc` Pointer to a Microsoft Windows draw context (can be obtained from a window using the Windows `GetDC` function).

DESCRIPTION

`Q3Win32DCDrawContext_SetDC` sets the draw context object identified by `drawContext` to the draw context specified by `hdc`.

Managing Direct Draw Surface Draw Contexts

QuickDraw 3D provides routines that you can use to create and manipulate direct draw surface draw contexts.

Note
Windows 32 draw contexts are always implicitly single buffered. ◆

Q3DDSurfaceDrawContext_New

You can use the `Q3DDSurfaceDrawContext_New` function to create a new Direct Draw surface draw context object.

```
TQ3DrawContextObject Q3DDSurfaceDrawContext_New  (const
                     TQ3DDSurfaceDrawContextData *drawContextData);
```

`drawContextData`
Pointer to a Direct Draw surface draw context data structure.

return value A Direct Draw surface draw context object.

DESCRIPTION

`Q3DDSurfaceDrawContext_New` returns a Direct Draw surface draw context object if it is successful; otherwise it returns `NULL`. The application must set up the necessary data structure, as described in "Direct Draw Surface Draw Context Structure," beginning on page 847.

Q3DDSurfaceDrawContext_GetDirectDrawSurface

You can use the `Q3DDSurfaceDrawContext_GetDirectDrawSurface` function to retrieve the surface descriptor associated with a Direct Draw surface draw context object.

```
TQ3Status Q3DDSurfaceDrawContext_GetDirectDrawSurface (
                     TQ3DrawContextObject drawContext,
                     TQ3DDSurfaceDescriptor *ddSurfaceDescriptor);
```

`drawContext` A draw context object.

`ddSurfaceDescriptor`
A surface descriptor (see "Direct Draw Surface Draw Context Structure," beginning on page 847).

DESCRIPTION

`Q3DDSurfaceDrawContext_GetDirectDrawSurface` returns in `ddSurfaceDescriptor` the Direct Draw surface descriptor that is associated with the draw context object specified by `drawContext`.

Q3DDSurfaceDrawContext_SetDirectDrawSurface

You can use the `Q3DDSurfaceDrawContext_SetDirectDrawSurface` function to set the surface descriptor associated with a Direct Draw surface draw context object.

```
TQ3Status Q3DDSurfaceDrawContext_SetDirectDrawSurface (
                    TQ3DrawContextObject drawContext,
                    const TQ3DDSurfaceDescriptor *ddSurfaceDescriptor);
```

`drawContext` A draw context object.

`ddSurfaceDescriptor`
A surface descriptor (see "Direct Draw Surface Draw Context Structure," beginning on page 847).

DESCRIPTION

`Q3DDSurfaceDrawContext_SetDirectDrawSurface` sets the draw context object identified by `drawContext` to use the surface descriptor specified by `ddSurfaceDescriptor`.

Draw Context Errors, Warnings, and Notices

The following errors, warnings, and notices may be returned by draw context object routines. A list of general QuickDraw 3D errors is given in "QuickDraw 3D Errors, Warnings, and Notices" (page 87).

```
kQ3ErrorBadDrawContextType
kQ3ErrorBadDrawContextFlag
kQ3ErrorBadDrawContext
kQ3ErrorUnsupportedPixelDepth
kQ3WarningInvalidPaneDimensions
kQ3NoticeDrawContextNotSetUsingInternalDefaults
```

CHAPTER 12

Draw Context Objects

CHAPTER 13

View Objects

This chapter describes view objects (or views) and the functions you can use to manipulate them. You use a view to specify the camera, the group of lights, the draw context, and the renderer that you want QuickDraw 3D to use when rendering an image of a model. You also use views when picking and performing some other operations on a model.

To use this chapter, you should already be familiar with cameras, light groups, draw contexts, and renderers. See the chapters "Camera Objects," "Group Objects," "Draw Context Objects," and "Renderer Objects" for information on creating and manipulating these four kinds of objects. You must create and configure instances of these objects before you can attach them to a view.

You can also attach one or more kinds of shaders to a view to achieve special visual effects. Once again, you must create and configure a shader before attaching it to a view. See the chapter "Shader Objects" for information on the available kinds of shaders.

This chapter begins by describing view objects and their features. Then it shows how to create and attach objects to views. The section "View Objects Routines," beginning on page 876 provides a complete description of the routines you can use to create and manipulate view objects.

QuickDraw 3D provides one subclass of the view class, the user interface view class. A user interface view is a type of view that allows the user to interact (using interface elements such as a 3D cursor or widgets) with the three-dimensional objects displayed in the view. See the chapter "Pointing Device Manager" for information on user interface views and the functions you can use to create and manipulate them.

About View Objects

A **view object** (or, more briefly, a **view**) is a type of QuickDraw 3D object that maintains the information necessary to render a single scene or image of a model. A view also maintains the information necessary to perform picking, calculate a bounding box or sphere, and write data to a file. A view is essentially a collection of a single camera, a (possibly empty) group of lights, a draw context, and a renderer. As you've seen, a camera defines a point of view onto a three-dimensional model and a method of projecting the model onto a two-dimensional view plane. The group of lights provides illumination on the objects in the model. The draw context defines the destination of the two-dimensional image, and the renderer determines the method of generating the image from the model.

In addition to these four types of objects that are necessary to render a single image of a model, a view can also contain one or more kinds of shaders. The QuickDraw 3D shading architecture provides a powerful way to modify aspects of an image. QuickDraw 3D supports many kinds of shaders, which are applied at different stages of the process of generating an image of a model. This chapter describes how to attach a shader to a view. For a complete description of the QuickDraw 3D shading architecture and for information on creating an instance of a specific kind of shader, see the chapter "Shader Objects."

A view is of type `TQ3ViewObject`, which is one of the four main subclasses of QuickDraw 3D objects. The structure of a view object is opaque; you must create and manipulate views solely using functions supplied by QuickDraw 3D (for example, `Q3View_New`).

Using View Objects

QuickDraw 3D supplies routines that you can use to create view objects, attach cameras, renderers, and other objects to them, and render images in those view objects. This section describes how to accomplish these tasks.

▲ **WARNING**
After instantiating a view object you can no longer register a new object; doing so will put the view memory structure out of sync. For example, if you register or unregister a shape object while a view is instantiated, the system will crash because the view stacks have become invalid. ▲

Creating and Configuring a View

You create a view object by calling the function `Q3View_New`. If successful, `Q3View_New` returns a new empty view object. You must then configure the view object by specifying a renderer, a camera, a group of lights, and a model. Listing 1-9 (page 67) illustrates how to create and configure a view. Only one object of each of these types can be associated with a view object at a given time. You can, however, have multiple view objects in your application, each associated with a different window.

Note
The group of lights is optional. A view, however, must contain a camera, a renderer, and a draw context. ◆

Rendering an Image

Once you have created and configured a view, you can use it to render an image of a model. To do so, you need to enter into the rendering state by calling the `Q3View_StartRendering` function. Then you specify the model to be drawn and call `Q3View_EndRendering`. Because the renderer might not have had sufficient memory to complete the rendering when you call `Q3View_EndRendering`, you might need to respecify the model, to give the renderer another pass at the model's data. As a result, you almost always call `Q3View_StartRendering` and `Q3View_EndRendering` in a **rendering loop,** shown in outline in Listing 13-1.

Listing 13-1 Rendering a model

```
Q3View_StartRendering(myView);
do {
    /*submit the model here*/
} while (Q3View_EndRendering(myView) ==
                                        kQ3ViewStatusRetraverse);
```

The `Q3View_EndRendering` function returns a view status value that indicates the status of the rendering process. If `Q3View_EndRendering` returns the value `kQ3ViewStatusRetraverse`, you should reenter your rendering loop. If `Q3View_EndRendering` returns `kQ3ViewStatusDone`, `kQ3ViewStatusError`, or `kQ3ViewStatusCancelled`, you should exit the loop.

As you know, QuickDraw 3D supports immediate mode, retained mode, and mixed mode rendering. You use a rendering loop for all these rendering modes, but they differ in how you create and draw the objects in a model. To use retained mode rendering, you let QuickDraw 3D allocate memory to hold the data associated with a particular object or group of objects. For example, to render a box in retained mode, you must first create the box by calling the `Q3Box_New` function. Then you draw the box by calling the `Q3Geometry_Submit` function, as illustrated in Listing 13-2.

Listing 13-2 Creating and rendering a retained object

```
TQ3BoxData                  myBoxData;
TQ3GeometryObject           myBox;

Q3Point3D_Set(&myBoxData.origin, 1.0, 1.0, 1.0);
Q3Vector3D_Set(&myBoxData.orientation, 0, 2.0, 0);
Q3Vector3D_Set(&myBoxData.minorAxis, 2.0, 0, 0);
Q3Vector3D_Set(&myBoxData.majorAxis, 0, 0, 2.0);
myBox = Q3Box_New(&myBoxData);

Q3View_StartRendering(myView);
do {
    Q3Geometry_Submit(myBox, myView);
} while (Q3View_EndRendering(myView) ==
                                        kQ3ViewStatusRetraverse);
```

In general, you use retained mode rendering when much of the model remains unchanged from frame to frame. For retained mode rendering, you can use the following routines inside a rendering loop:

```
Q3Style_Submit
Q3Geometry_Submit
Q3Transform_Submit
Q3Group_Submit
```

To use immediate mode rendering, you allocate memory for an object yourself and draw the object using an immediate mode drawing routine, as illustrated in Listing 13-3.

Listing 13-3 Creating and rendering an immediate object

```
TQ3BoxData                      myBoxData;

Q3Point3D_Set(&myBoxData.origin, 1.0, 1.0, 1.0);
Q3Vector3D_Set(&myBoxData.orientation, 0, 2.0, 0);
Q3Vector3D_Set(&myBoxData.minorAxis, 2.0, 0, 0);
Q3Vector3D_Set(&myBoxData.majorAxis, 0, 0, 2.0);

Q3View_StartRendering(myView);
do {
    Q3Box_Submit(myBoxData, myView);
} while (Q3View_EndRendering(myView) ==
                                            kQ3ViewStatusRetraverse);
```

In general, you use immediate mode when your application does not need to retain the geometric data for subsequent use.

View Objects Reference

This section describes the QuickDraw 3D routines that you can use to manage view objects.

View Objects Routines

This section describes the routines you can use to manage views.

Creating and Configuring Views

QuickDraw 3D provides routines for creating a new view and for getting or setting a view's renderer, camera, light group, and draw context.

Q3View_New

You can use the `Q3View_New` function to create a new view object.

```
TQ3ViewObject Q3View_New (void);
```

DESCRIPTION

The `Q3View_New` function returns, as its function result, a new view object. Before you can render a model in that view, you must first set the view's renderer, camera, and draw context. You can also set the view's group of lights. `Q3View_New` returns `NULL` if it cannot create a new view object.

Q3View_GetRenderer

You can use the `Q3View_GetRenderer` function to get the renderer associated with a view.

```
TQ3Status Q3View_GetRenderer (
                    TQ3ViewObject view,
                    TQ3RendererObject *renderer);
```

`view` A view.

`renderer` On exit, the renderer object currently associated with the specified view.

DESCRIPTION

The `Q3View_GetRenderer` function returns, in the `renderer` parameter, the renderer currently associated with the view specified by the `view` parameter. The reference count of that renderer is incremented.

Q3View_SetRenderer

You can use the `Q3View_SetRenderer` function to set the renderer associated with a view.

```
TQ3Status Q3View_SetRenderer (
                    TQ3ViewObject view,
                    TQ3RendererObject renderer);
```

`view` A view.

`renderer` A renderer object.

DESCRIPTION

The `Q3View_SetRenderer` function attaches the renderer specified by the `renderer` parameter to the view specified by the `view` parameter. The reference count of the specified renderer is incremented. In addition, if some other renderer was already attached to the specified view, the reference count of that renderer is decremented.

SEE ALSO

For information on creating and manipulating renderers, see the chapter "Renderer Objects."

Q3View_SetRendererByType

You can use the `Q3View_SetRendererByType` function to set the renderer associated with a view by specifying its type.

```
TQ3Status Q3View_SetRendererByType (
                    TQ3ViewObject view,
                    TQ3ObjectType type);
```

`view` A view.

`type` A renderer type.

DESCRIPTION

The `Q3View_SetRendererByType` function attaches the renderer having the type specified by the `type` parameter to the view specified by the `view` parameter. The reference count of the specified render is incremented. In addition, if some other renderer was already attached to the specified view, the reference count of that renderer is decremented.

Q3View_GetCamera

You can use the `Q3View_GetCamera` function to get the camera associated with a view.

```
TQ3Status Q3View_GetCamera (
                    TQ3ViewObject view,
                    TQ3CameraObject *camera);
```

`view` A view.

`camera` On exit, the camera object currently associated with the specified view.

DESCRIPTION

The `Q3View_GetCamera` function returns, in the `camera` parameter, the camera currently associated with the view specified by the `view` parameter. The reference count of that camera is incremented.

Q3View_SetCamera

You can use the `Q3View_SetCamera` function to set the camera associated with a view.

```
TQ3Status Q3View_SetCamera (
                    TQ3ViewObject view,
                    TQ3CameraObject camera);
```

`view` A view.

`camera` A camera object.

DESCRIPTION

The `Q3View_SetCamera` function attaches the camera specified by the `camera` parameter to the view specified by the `view` parameter. The reference count of the specified camera is incremented. In addition, if some other camera was already attached to the specified view, the reference count of that camera is decremented.

SEE ALSO

For information on creating and manipulating cameras, see the chapter "Camera Objects."

Q3View_GetLightGroup

You can use the `Q3View_GetLightGroup` function to get the light group associated with a view.

```
TQ3Status Q3View_GetLightGroup (
                    TQ3ViewObject view,
                    TQ3GroupObject *lightGroup);
```

`view` A view.

`lightGroup` On exit, the light group currently associated with the specified view.

DESCRIPTION

The `Q3View_GetLightGroup` function returns, in the `lightGroup` parameter, the light group currently associated with the view specified by the `view` parameter. The reference count of that light group is incremented.

Q3View_SetLightGroup

You can use the `Q3View_SetLightGroup` function to set the light group associated with a view.

```
TQ3Status Q3View_SetLightGroup (
                    TQ3ViewObject view,
                    TQ3GroupObject lightGroup);
```

`view` A view.

`lightGroup` A light group.

DESCRIPTION

The `Q3View_SetLightGroup` function attaches the light group specified by the `lightGroup` parameter to the view specified by the `view` parameter. The reference count of the specified light group is incremented. In addition, if some other light group was already attached to the specified view, the reference count of that light group is decremented.

SEE ALSO

For information on creating and manipulating light groups, see the chapters "Light Objects" and "Group Objects."

Q3View_GetDrawContext

You can use the `Q3View_GetDrawContext` function to get the draw context associated with a view.

```
TQ3Status Q3View_GetDrawContext (
                    TQ3ViewObject view,
                    TQ3DrawContextObject *drawContext);
```

`view` A view.

`drawContext` On exit, the draw context currently associated with the specified view.

DESCRIPTION

The `Q3View_GetDrawContext` function returns, in the `drawContext` parameter, the draw context currently associated with the view specified by the `view` parameter. The reference count of that draw context is incremented.

Q3View_SetDrawContext

You can use the `Q3View_SetDrawContext` function to set the draw context associated with a view.

```
TQ3Status Q3View_SetDrawContext (
                    TQ3ViewObject view,
                    TQ3DrawContextObject drawContext);
```

`view` A view.

`drawContext` A draw context object.

DESCRIPTION

The `Q3View_SetDrawContext` function attaches the draw context specified by the `drawContext` parameter to the view specified by the `view` parameter. The reference count of the specified draw context is incremented. In addition, if some other draw context was already attached to the specified view, the reference count of that draw context is decremented.

SEE ALSO

For information on creating and manipulating draw contexts, see the chapter "Draw Context Objects."

Rendering in a View

QuickDraw 3D provides routines that you can use to manage the process of rendering in a view. The view must already exist and be fully configured before you call these routines.

Q3View_StartRendering

You can use the `Q3View_StartRendering` function to start rendering an image of a model.

```
TQ3Status Q3View_StartRendering (TQ3ViewObject view);
```

`view` A view.

DESCRIPTION

The `Q3View_StartRendering` function begins the process of rendering an image of a model in the view specified by the `view` parameter. After calling `Q3View_StartRendering`, you specify the model to be drawn (for instance, by calling `Q3Geometry_Submit`). When you have completely specified that model, you should call `Q3View_EndRendering` to complete the rendering of the image. Because the renderer attached to the specified view might need to reprocess the model data, you should always call `Q3View_StartRendering` and `Q3View_EndRendering` in a rendering loop.

Calling `Q3View_StartRendering` automatically clears the buffer into which the rendered image is drawn.

SPECIAL CONSIDERATIONS

You should not call `Q3View_StartRendering` while rendering is already occurring.

ERRORS

`kQ3ErrorRenderingIsActive`

SEE ALSO

See "Rendering an Image" (page 873) for more information about a rendering loop.

Q3View_EndRendering

You can use the `Q3View_EndRendering` function to stop rendering an image of a model.

```
TQ3ViewStatus Q3View_EndRendering (TQ3ViewObject view);
```

`view` A view.

DESCRIPTION

The `Q3View_EndRendering` function returns, as its function result, a view status value that indicates the current state of the rendering of an image of a model in the view specified by the `view` parameter. `Q3View_EndRendering` returns one of these four values:

```
typedef enum TQ3ViewStatus {
    kQ3ViewStatusDone,
    kQ3ViewStatusRetraverse,
    kQ3ViewStatusError,
    kQ3ViewStatusCancelled
} TQ3ViewStatus;
```

If `Q3View_EndRendering` returns `kQ3ViewStatusDone`, the rendering of the image has been completed and the specified view is no longer in rendering mode. At that point, it is safe to exit your rendering loop. If double-buffering is active, the front buffer is updated with the rendered image.

IMPORTANT

If the renderer associated with the specified view relies on a hardware accelerator for some or all of its operation, `Q3View_EndRendering` may return `kQ3ViewStatusDone` even though the rendering has not yet completed. (When a hardware accelerator is present, rendering occurs asynchronously.) If you must know when the rendering has actually finished, call the `Q3View_Sync` function. ▲

If `Q3View_EndRendering` returns `kQ3ViewStatusRetraverse`, the rendering of the image has not yet been completed. You should respecify the model by reentering your rendering loop.

If `Q3View_EndRendering` returns `kQ3ViewStatusError`, the rendering of the image has failed because the renderer associated with the view encountered an error in processing the model. You should exit the rendering loop.

If `Q3View_EndRendering` returns `kQ3ViewStatusCancelled`, the rendering of the image has been canceled. You should exit the rendering loop.

SPECIAL CONSIDERATIONS

You should call `Q3View_EndRendering` only if rendering is already occurring.

SEE ALSO

See "Rendering an Image" (page 873) for a sample rendering loop.

Q3View_Cancel

You can use the `Q3View_Cancel` function to cancel the rendering, picking, bounding, or writing operation currently occurring in a view.

```
TQ3Status Q3View_Cancel (TQ3ViewObject view);
```

`view` A view.

DESCRIPTION

The `Q3View_Cancel` function interrupts the process of rendering an image of a model, submitting objects for picking, calculating a bounding box or sphere, or writing data to a file in accordance with the view specified by the `view` parameter. Any subsequent calls to `_Submit` routines for the specified view will fail, and `Q3View_EndRendering` (or the similar call for picking, bounding, or writing) will return `kQ3ViewStatusCancelled` when it is next executed. Note that you must still call `Q3View_EndRendering` (or the similar call for picking, bounding, or writing) after you have called `Q3View_Cancel`.

You can call `Q3View_Cancel` at any time. If the specified view is not in the submitting state, `Q3View_Cancel` returns `kQ3Failure`.

Q3View_Flush

You can use the `Q3View_Flush` function to flush buffered graphics to a rasterizer.

```
TQ3Status Q3View_Flush (TQ3ViewObject view);
```

`view` A view.

DESCRIPTION

The `Q3View_Flush` function is a non-blocking call that flushes all buffered graphics to a rasterizer (if one is implemented). It may or may not update the draw context, depending on the type of renderer. The `Q3View_Flush` function may be called only between calls to `Q3View_StartRendering` and `Q3View_EndRendering`.

Q3View_Sync

You can use the `Q3View_Sync` function to flush buffered graphics to a rasterizer and also update the draw context.

```
TQ3Status Q3View_Sync (TQ3ViewObject view);
```

`view` A view.

DESCRIPTION

The `Q3View_Sync` function is a blocking call that flushes all buffered graphics to a rasterizer and updates the draw context. Calling this function guarantees that the image is updated on return. You may call it only after calling `Q3View_EndRendering`.

Picking in a View

QuickDraw 3D provides routines that you can use to manage the process of picking in a view. The view must already exist and be fully configured before you call these routines.

Q3View_StartPicking

You can use the `Q3View_StartPicking` function to start picking in a view.

```
TQ3Status Q3View_StartPicking (
                    TQ3ViewObject view,
                    TQ3PickObject pick);
```

`view` A view.

`pick` A pick object.

DESCRIPTION

The `Q3View_StartPicking` function begins the process of picking in the view specified by the `view` parameter, using the pick object specified by the `pick` parameter. After calling `Q3View_StartPicking`, you specify the model (for instance, by calling `Q3Geometry_Submit`). When you have completely specified that model, you should call `Q3View_EndPicking` to complete the picking operation. The renderer attached to the specified view might need to reprocess the model data, so you should always call `Q3View_StartPicking` and `Q3View_EndPicking` in a picking loop.

SPECIAL CONSIDERATIONS

You should not call `Q3View_StartPicking` while picking is already occurring.

Q3View_EndPicking

You can use the `Q3View_EndPicking` function to end picking in a view.

```
TQ3ViewStatus Q3View_EndPicking (TQ3ViewObject view);
```

`view` A view.

DESCRIPTION

The `Q3View_EndPicking` function returns, as its function result, a view status value that indicates the current state of the picking in the view specified by the `view` parameter. `Q3View_EndPicking` returns one of these four values:

```
typedef enum TQ3ViewStatus {
    kQ3ViewStatusDone,
    kQ3ViewStatusRetraverse,
    kQ3ViewStatusError,
    kQ3ViewStatusCancelled
} TQ3ViewStatus;
```

If `Q3View_EndPicking` returns `kQ3ViewStatusDone`, the picking has been completed and the specified view is no longer in picking mode. At that point, it is safe to exit your picking loop.

If `Q3View_EndPicking` returns `kQ3ViewStatusRetraverse`, the picking has not yet been completed. You should respecify the model by reentering your picking loop.

If `Q3View_EndPicking` returns `kQ3ViewStatusError`, the picking has failed because the renderer associated with the view encountered an error in processing the model. You should exit the picking loop.

If `Q3View_EndPicking` returns `kQ3ViewStatusCancelled`, the picking has been canceled. You should exit the picking loop.

SPECIAL CONSIDERATIONS

You should call `Q3View_EndPicking` only if picking is already occurring.

Writing in a View

QuickDraw 3D provides routines that you can use to manage the process of writing a view's data to a file. The view must already exist and be fully configured before you call these routines.

Q3View_StartWriting

You can use the `Q3View_StartWriting` function to start writing to a file.

```
TQ3Status Q3View_StartWriting (
                    TQ3ViewObject view,
                    TQ3FileObject file);
```

`view` A view.

`file` A file object.

DESCRIPTION

The `Q3View_StartWriting` function begins the process of writing in the view specified by the `view` parameter, using the file object specified by the `file` parameter. After calling `Q3View_StartWriting`, you specify the model (for instance, by calling `Q3Geometry_Submit`). When you have completely specified that model, you should call `Q3View_EndWriting` to complete the write operation. The renderer attached to the specified view might need to reprocess the model data, so you should always call `Q3View_StartWriting` and `Q3View_EndWriting` in a writing loop.

SPECIAL CONSIDERATIONS

You should not call `Q3View_StartWriting` while writing is already occurring.

Q3View_EndWriting

You can use the `Q3View_EndWriting` function to end writing to a file.

```
TQ3ViewStatus Q3View_EndWriting (TQ3ViewObject view);
```

`view` A view.

DESCRIPTION

The `Q3View_EndWriting` function returns, as its function result, a view status value that indicates the current state of the writing in the view specified by the `view` parameter. `Q3View_EndWriting` returns one of these four values:

```
typedef enum TQ3ViewStatus {
    kQ3ViewStatusDone,
    kQ3ViewStatusRetraverse,
    kQ3ViewStatusError,
    kQ3ViewStatusCancelled
} TQ3ViewStatus;
```

If `Q3View_EndWriting` returns `kQ3ViewStatusDone`, the writing has been completed and the specified view is no longer in writing mode. At that point, it is safe to exit your writing loop.

If `Q3View_EndWriting` returns `kQ3ViewStatusRetraverse`, the writing has not yet been completed. You should respecify the model by reentering your writing loop.

If `Q3View_EndWriting` returns `kQ3ViewStatusError`, the writing has failed because the renderer associated with the view encountered an error in processing the model. You should exit the writing loop.

If `Q3View_EndWriting` returns `kQ3ViewStatusCancelled`, the writing has been canceled. You should exit the writing loop.

SPECIAL CONSIDERATIONS

You should call `Q3View_EndWriting` only if writing is already occurring.

Bounding in a View

As described in the chapters "Geometric Objects" and "Group Objects", QuickDraw 3D provides routines that you can use to compute the bounding box and bounding sphere of an object or a group of objects in a model. Computing an object's bounding box or bounding sphere requires applying to

it all the transforms in the current view transform stack. QuickDraw 3D provides routines that you must call before and after computing an object's bounds.

QuickDraw 3D also provides a routine that you can use to determine whether a bounding box is visible in a view. You might use that routine to avoid specifying portions of a model that aren't visible.

Q3View_StartBoundingBox

You can use the `Q3View_StartBoundingBox` function to start computing an object's bounding box.

```
TQ3Status Q3View_StartBoundingBox (
                    TQ3ViewObject view,
                    TQ3ComputeBounds computeBounds);
```

`view` A view.

`computeBounds`
A constant that specifies how the bounding box should be computed. See the following description for details.

DESCRIPTION

The `Q3View_StartBoundingBox` function begins the process of calculating a bounding box in the view specified by the `view` parameter. After calling `Q3View_StartBoundingBox`, you specify the model (for instance, by calling `Q3Geometry_Submit`). When you have completely specified that model, you should call `Q3View_EndBoundingBox` to complete the bounding operation. The renderer attached to the specified view might need to reprocess the model data, so you should always call `Q3View_StartBoundingBox` and `Q3View_EndBoundingBox` in a bounding loop.

The `computeBounds` parameter determines the algorithm that QuickDraw 3D uses to calculate the bounding box. You should set `computeBounds` to one of these constants:

```
typedef enum TQ3ComputeBounds {
    kQ3ComputeBoundsExact,
    kQ3ComputeBoundsApproximate
} TQ3ComputeBounds;
```

If you set `computeBounds` to `kQ3ComputeBoundsExact`, the vertices of the geometric object are transformed into world space, and then the world space bounding box is computed from the transformed vertices. This method of calculating a bounding box produces the most precise bounding box but is slower than using the `kQ3ComputeBoundsApproximate` method.

If you set `computeBounds` to `kQ3ComputeBoundsApproximate`, a local bounding box is computed from the vertices of the geometric object, and then that bounding box is transformed into world space. The transformed bounding box is returned as the approximate bounding box of the geometric object. This method of calculating a bounding box is faster than using the `kQ3ComputeBoundsExact` method but produces a bounding box that might be larger than that computed by the exact method.

Q3View_EndBoundingBox

You can use the `Q3View_EndBoundingBox` function to stop computing an object's bounding box.

```
TQ3ViewStatus Q3View_EndBoundingBox (
                    TQ3ViewObject view,
                    TQ3BoundingBox *result);
```

`view` A view.

`result` On exit, the bounding box for the objects specified in the bounding loop.

DESCRIPTION

The `Q3View_EndBoundingBox` function returns, as its function result, a view status value that indicates the current state of the bounding box calculation of the objects in the view specified by the `view` parameter. `Q3View_EndBoundingBox` returns one of these four values:

```
typedef enum TQ3ViewStatus {
    kQ3ViewStatusDone,
    kQ3ViewStatusRetraverse,
    kQ3ViewStatusError,
    kQ3ViewStatusCancelled
} TQ3ViewStatus;
```

If `Q3View_EndBoundingBox` returns `kQ3ViewStatusDone`, the bounding box calculation has completed. At that point, it is safe to exit your bounding loop. The `result` parameter contains the bounding box.

If `Q3View_EndBoundingBox` returns `kQ3ViewStatusRetraverse`, the bounding box calculation has not yet completed. You should respecify the model by reentering your bounding loop.

If `Q3View_EndBoundingBox` returns `kQ3ViewStatusError`, the bounding box calculation has failed. You should exit the bounding loop.

If `Q3View_EndBoundingBox` returns `kQ3ViewStatusCancelled`, the bounding box calculation has been canceled. You should exit the bounding loop.

SPECIAL CONSIDERATIONS

You should call `Q3View_EndBoundingBox` only if bounding box calculation is already occurring.

Q3View_StartBoundingSphere

You can use the `Q3View_StartBoundingSphere` function to start computing an object's bounding sphere.

```
TQ3Status Q3View_StartBoundingSphere (
                    TQ3ViewObject view,
                    TQ3ComputeBounds computeBounds);
```

`view` A view.

`computeBounds`
A constant that specifies how the bounding sphere should be computed. See the following description for details.

DESCRIPTION

The `Q3View_StartBoundingSphere` function begins the process of calculating a bounding sphere in the view specified by the `view` parameter. After calling `Q3View_StartBoundingSphere`, you specify the model (for instance, by calling `Q3Geometry_Submit`). When you have completely specified that model, you should call `Q3View_EndBoundingSphere` to complete the bounding operation. The renderer attached to the specified view might need to reprocess the model data, so you should always call `Q3View_StartBoundingSphere` and `Q3View_EndBoundingSphere` in a bounding loop.

The `computeBounds` parameter determines the algorithm that QuickDraw 3D uses to calculate the bounding sphere. You should set `computeBounds` to one of these constants:

```
typedef enum TQ3ComputeBounds {
    kQ3ComputeBoundsExact,
    kQ3ComputeBoundsApproximate
} TQ3ComputeBounds;
```

If you set `computeBounds` to `kQ3ComputeBoundsExact`, the vertices of the geometric object are transformed into world space, and then the world space bounding sphere is computed from the transformed vertices. This method of calculating a bounding sphere produces the most precise bounding sphere but is slower than using the `kQ3ComputeBoundsApproximate` method.

If you set `computeBounds` to `kQ3ComputeBoundsApproximate`, a local bounding sphere is computed from the vertices of the geometric object, and then that bounding sphere is transformed into world space. The transformed bounding sphere is returned as the approximate bounding sphere of the geometric object. This method of calculating a bounding sphere is faster than using the `kQ3ComputeBoundsExact` method but produces a bounding sphere that might be larger than that computed by the exact method.

Q3View_EndBoundingSphere

You can use the `Q3View_EndBoundingSphere` function to stop computing an object's bounding sphere.

```
TQ3ViewStatus Q3View_EndBoundingSphere (
                    TQ3ViewObject view,
                    TQ3BoundingSphere *result);
```

`view` A view.

`result` On exit, the bounding sphere for the objects specified in the bounding loop.

DESCRIPTION

The `Q3View_EndBoundingSphere` function returns, as its function result, a view status value that indicates the current state of the bounding sphere calculation of the objects in the view specified by the `view` parameter. `Q3View_EndBoundingBox` returns one of these four values:

```
typedef enum TQ3ViewStatus {
    kQ3ViewStatusDone,
    kQ3ViewStatusRetraverse,
    kQ3ViewStatusError,
    kQ3ViewStatusCancelled
} TQ3ViewStatus;
```

If `Q3View_EndBoundingSphere` returns `kQ3ViewStatusDone`, the bounding sphere calculation has completed. At that point, it is safe to exit your bounding loop. The `result` parameter contains the bounding sphere.

If `Q3View_EndBoundingSphere` returns `kQ3ViewStatusRetraverse`, the bounding sphere calculation has not yet completed. You should respecify the model by reentering your bounding loop.

If `Q3View_EndBoundingSphere` returns `kQ3ViewStatusError`, the bounding sphere calculation has failed. You should exit the bounding loop.

If `Q3View_EndBoundingSphere` returns `kQ3ViewStatusCancelled`, the bounding sphere calculation has been canceled. You should exit the bounding loop.

SPECIAL CONSIDERATIONS

You should call `Q3View_EndBoundingSphere` only if bounding sphere calculation is already occurring.

Q3View_IsBoundingBoxVisible

You can use the `Q3View_IsBoundingBoxVisible` function to determine whether a bounding box is visible in a view (that is, whether it lies in the viewing frustum).

```
TQ3Boolean Q3View_IsBoundingBoxVisible (
                    TQ3ViewObject view,
                    const TQ3BoundingBox *bbox);
```

`view` A view.

`bbox` A bounding box.

DESCRIPTION

The `Q3View_IsBoundingBoxVisible` function returns, as its function result, a Boolean value that indicates whether the bounding box specified by the `bbox` parameter is visible in the view specified by the `view` parameter (`kQ3True`) or is not visible (`kQ3False`). `Q3View_IsBoundingBoxVisible` transforms the specified bounding box by the view's local-to-world transform and then determines whether the box lies in the viewing frustum.

Setting Idle Methods

QuickDraw 3D provides a function that you can use to set a view's idle method. QuickDraw 3D executes your idle method occasionally during lengthy operations. See "Application-Defined Routines" (page 909) for information on writing an idle method.

IMPORTANT

Your application's callback method may be called during a hardware interrupt, and therefore it should not use Macintosh Toolbox routines. To overcome this limitation, an interrupt-level render completion function can set a global variable, requesting Toolbox calls, that the client polls at noninterrupt time. ▲

Q3View_SetIdleMethod

You can use the `Q3View_SetIdleMethod` function to set a view's idle method.

```
TQ3Status Q3View_SetIdleMethod (
                    TQ3ViewObject view,
                    TQ3ViewIdleMethod idleMethod,
                    const void *idlerData);
```

`view`	A view.
`idleMethod`	A pointer to an idle method.
`idlerData`	A pointer to an application-defined block of data. This pointer is passed to the idle method when it is executed.

DESCRIPTION

The `Q3View_SetIdleMethod` function sets the idle method of the view specified by the `view` parameter to the function specified by the `idleMethod` parameter. The `idlerData` parameter is passed to your callback routine whenever it is executed.

SPECIAL CONSIDERATIONS

Because your callback function may be called at hardware interrupt level, be careful about using Macintosh Toolbox routines. To call the Toolbox, you may want to set a global variable that you can later poll at noninterrupt level.

Writing Custom Data

QuickDraw 3D provides a function that you can use to write custom objects. In general, you should call this function only within your custom write method.

Q3View_SubmitWriteData

You can use the `Q3View_SubmitWriteData` function to submit for writing the data associated with a custom object.

```
TQ3Status Q3View_SubmitWriteData (
                    TQ3ViewObject view,
                    TQ3Size size,
                    void *data,
                    void *deleteData);
```

`view` A view.

`size` The number of bytes of data to write. This value should be aligned on 4-byte boundaries.

`data` A pointer to a buffer of data to be submitted for writing.

`deleteData` A pointer to a data-deletion method. This method is called after your custom write method exits (whether or not the write method succeeds or fails). The value of the `data` parameter is passed as a parameter to your method.

DESCRIPTION

The `Q3View_SubmitWriteData` function submits the data specified by the `data` and `size` parameters for writing in the view specified by the `view` parameter. You can call `Q3View_SubmitWriteData` in a custom object-traversal method to write the data of a custom object. `Q3View_SubmitWriteData` calls the write method associated with that custom object type to actually write the data to a file object. When the write method returns, `Q3View_SubmitWriteData` executes the data-deletion method specified by the `deleteData` parameter.

SPECIAL CONSIDERATIONS

You should call this function only within a custom object-traversal method. See the chapter "File Objects" for more information about traversal methods.

Pushing and Popping the Graphics State

QuickDraw 3D maintains a graphics state during rendering that contains camera and lighting information, a transformation matrix stack, an attributes stack, and a style stack. When it is traversing a hierarchical scene database, QuickDraw 3D automatically pushes and pops graphics states onto and off the graphics state stack.

QuickDraw 3D provides routines that you can use to push and pop a graphics state during the rendering of an image or other view operation. You can push a graphics state by calling `Q3Push_Submit`. Subsequent rendering may alter the graphics state by drawing materials, styles, and transforms. You can restore a saved graphics state by calling `Q3Pop_Submit`. You're likely to use these functions only if you want to simulate the traversal of a hierarchical structure when operating in immediate mode.

Q3Push_Submit

You can use the `Q3Push_Submit` function to push a graphics state onto the graphics state stack.

```
TQ3Status Q3Push_Submit (TQ3ViewObject view);
```

`view` A view.

DESCRIPTION

The `Q3Push_Submit` function pushes the current graphics state of the view specified by the `view` parameter onto the graphics state stack. There must be a matching call to `Q3Pop_Submit` before the next call to `Q3View_EndRendering`.

SPECIAL CONSIDERATIONS

You should call `Q3Push_Submit` only in a submitting loop.

Q3Pop_Submit

You can use the `Q3Pop_Submit` function to pop a graphics state off the graphics state stack.

```
TQ3Status Q3Pop_Submit (TQ3ViewObject view);
```

`view` A view.

DESCRIPTION

The `Q3Pop_Submit` function pops the graphics state of the view specified by the `view` parameter off the graphics state stack. Every call to `Q3Pop_Submit` must match a previous call to `Q3Push_Submit`.

SPECIAL CONSIDERATIONS

You should call `Q3Pop_Submit` only in a submitting loop.

Getting a View's Transforms

QuickDraw 3D provides routines that you can use to get matrix representations of the transforms associated with a view.

IMPORTANT
You should call these routines only between calls to `Q3View_StartRendering` and `Q3View_EndRendering` (or similar submitting loops). If you call them at any other time, they return `kQ3Failure`. ▲

Q3View_GetLocalToWorldMatrixState

You can use the `Q3View_GetLocalToWorldMatrixState` function to get a view's local-to-world transform matrix.

```
TQ3Status Q3View_GetLocalToWorldMatrixState (
                    TQ3ViewObject view,
                    TQ3Matrix4x4 *matrix);
```

`view` A view.

`matrix` On exit, a 4-by-4 matrix representing the local-to-world transform of the specified view.

DESCRIPTION

The `Q3View_GetLocalToWorldMatrixState` function returns, in the `matrix` parameter, a 4-by-4 matrix that represents the local-to-world transform of the view specified by the `view` parameter.

Q3View_GetWorldToFrustumMatrixState

You can use the `Q3View_GetWorldToFrustumMatrixState` function to get a view's world-to-frustum transform matrix.

```
TQ3Status Q3View_GetWorldToFrustumMatrixState (
                    TQ3ViewObject view,
                    TQ3Matrix4x4 *matrix);
```

`view` A view.

`matrix` On exit, a 4-by-4 matrix representing the world-to-frustum transform of the specified view.

DESCRIPTION

The `Q3View_GetWorldToFrustumMatrixState` function returns, in the `matrix` parameter, a 4-by-4 matrix that represents the world-to-frustum transform of the view specified by the `view` parameter.

SPECIAL CONSIDERATIONS

You should call the `Q3View_GetWorldToFrustumMatrixState` function only from within a rendering loop. Its behavior when called outside a rendering loop is unpredictable.

Q3View_GetFrustumToWindowMatrixState

You can use the `Q3View_GetFrustumToWindowMatrixState` function to get a view's frustum-to-window transform matrix.

```
TQ3Status Q3View_GetFrustumToWindowMatrixState (
                    TQ3ViewObject view,
                    TQ3Matrix4x4 *matrix);
```

`view` A view.

`matrix` On exit, a 4-by-4 matrix representing the frustum-to-window transform of the specified view.

DESCRIPTION

The `Q3View_GetFrustumToWindowMatrixState` function returns, in the `matrix` parameter, a 4-by-4 matrix that represents the frustum-to-window transform of the view specified by the `view` parameter. The window is either the pixmap associated with a pixmap draw context or the window associated with a window draw context (for example, the Macintosh draw context). If, in a window system draw context, a part of a window (a pane) has been associated with the view, this function returns the matrix that maps the view frustum to that part of the window.

The z value of a point p_w in window space obtained by applying the transform returned by `Q3View_GetFrustumToWindowMatrixState` to a point p_f in the frustum space is the z value of point p_f (which ranges from 0.0 to 1.0, inclusive). You might use the z value of a transformed point to determine whether that point would be clipped (if the z value is less than 0 or greater than 1.0, the original point lies outside the viewing frustum).

Managing a View's Style States

QuickDraw 3D provides routines that you can use to get information about the style state of a view.

Note
For information about styles and style types, see the chapter "Style Objects." ◆

Q3View_GetBackfacingStyleState

You can use the `Q3View_GetBackfacingStyleState` function to get the current backfacing style of a view.

```
TQ3Status Q3View_GetBackfacingStyleState (
                    TQ3ViewObject view,
                    TQ3BackfacingStyle *backfacingStyle);
```

`view` A view.

`backfacingStyle`
On exit, the current backfacing style of the specified view.

DESCRIPTION

The `Q3View_GetBackfacingStyleState` function returns, in the `backfacingStyle` parameter, the current backfacing style of the view specified by the `view` parameter.

Q3View_GetInterpolationStyleState

You can use the `Q3View_GetInterpolationStyleState` function to get the current interpolation style of a view.

```
TQ3Status Q3View_GetInterpolationStyleState (
                    TQ3ViewObject view,
                    TQ3InterpolationStyle *interpolationType);
```

`view` A view.

`interpolationType`
On exit, the current interpolation style of the specified view.

DESCRIPTION

The `Q3View_GetInterpolationStyleState` function returns, in the `interpolationType` parameter, the current interpolation style of the view specified by the `view` parameter.

Q3View_GetFillStyleState

You can use the `Q3View_GetFillStyleState` function to get the current fill style of a view.

```
TQ3Status Q3View_GetFillStyleState (
                    TQ3ViewObject view,
                    TQ3FillStyle *fillStyle);
```

`view` A view.

`fillStyle` On exit, the current fill style of the specified view.

DESCRIPTION

The `Q3View_GetFillStyleState` function returns, in the `fillStyle` parameter, the current fill style of the view specified by the `view` parameter.

Q3View_GetHighlightStyleState

You can use the `Q3View_GetHighlightStyleState` function to get the current highlight style of a view.

```
TQ3Status Q3View_GetHighlightStyleState (
                    TQ3ViewObject view,
                    TQ3AttributeSet *highlightStyle);
```

`view` A view.

`highlightStyle`
On exit, the current highlight style of the specified view.

DESCRIPTION

The `Q3View_GetHighlightStyleState` function returns, in the `highlightStyle` parameter, the current highlight style of the view specified by the `view` parameter. You are responsible for disposing of the returned attribute set (by calling `Q3Object_Dispose`) when you are done using it.

Q3View_GetSubdivisionStyleState

You can use the `Q3View_GetSubdivisionStyleState` function to get the current subdivision style of a view.

```
TQ3Status Q3View_GetSubdivisionStyleState (
                    TQ3ViewObject view,
                    TQ3SubdivisionStyleData *subdivisionStyle);
```

`view` A view.

`subdivisionStyle`
On exit, the current subdivision style of the specified view.

DESCRIPTION

The `Q3View_GetSubdivisionStyleState` function returns, in the `subdivisionStyle` parameter, the current subdivision style of the view specified by the `view` parameter.

Q3View_GetOrientationStyleState

You can use the `Q3View_GetOrientationStyleState` function to get the current frontfacing direction style of a view.

```
TQ3Status Q3View_GetOrientationStyleState (
                    TQ3ViewObject view,
                    TQ3OrientationStyle *fontFacingDirectionStyle);
```

`view` A view.

`fontFacingDirectionStyle`
On exit, the current frontfacing direction style of the specified view.

DESCRIPTION

The `Q3View_GetOrientationStyleState` function returns, in the `fontFacingDirectionStyle` parameter, the current frontfacing direction style of the view specified by the `view` parameter.

Q3View_GetReceiveShadowsStyleState

You can use the `Q3View_GetReceiveShadowsStyleState` function to get the current shadow-receiving style of a view.

```
TQ3Status Q3View_GetReceiveShadowsStyleState (
                    TQ3ViewObject view,
                    TQ3Boolean *receives);
```

`view` A view.

`receives` On exit, the current shadow-receiving style of the specified view.

DESCRIPTION

The `Q3View_GetReceiveShadowsStyleState` function returns, in the `receives` parameter, the current shadow-receiving style of the view specified by the `view` parameter.

Q3View_GetPickIDStyleState

You can use the `Q3View_GetPickIDStyleState` function to get the current picking ID style of a view.

```
TQ3Status Q3View_GetPickIDStyleState (
                    TQ3ViewObject view,
                    unsigned long *pickIDStyle);
```

`view` A view.

`pickIDStyle` On exit, the current picking ID style of the specified view.

DESCRIPTION

The `Q3View_GetPickIDStyleState` function returns, in the `pickIDStyle` parameter, the current picking ID style of the view specified by the `view` parameter.

Q3View_GetPickPartsStyleState

You can use the `Q3View_GetPickPartsStyleState` function to get the current picking parts style of a view.

```
TQ3Status Q3View_GetPickPartsStyleState (
                    TQ3ViewObject view,
                    TQ3PickParts *pickPartsStyle);
```

`view` A view.

`pickPartsStyle`
On exit, the current picking parts style of the specified view.

DESCRIPTION

The `Q3View_GetPickPartsStyleState` function returns, in the `pickPartsStyle` parameter, the current picking parts style of the view specified by the `view` parameter.

Q3View_GetAntiAliasStyleState

You can use the `Q3View_GetAntiAliasStyleState` function to determine the state of antialiasing in a view.

```
TQ3Status Q3View_GetAntiAliasStyleState(
                    TQ3ViewObject view,
                    TQ3AntiAliasStyleData *antiAliasData);
```

`view` A view.

`antiAliasData` Pointer to an antialias style data structure.

DESCRIPTION

The `Q3View_GetAntiAliasStyleState` function must be called within a rendering loop. It returns the current state of the antialiasing style of the specified `TQ3ViewObject` view in a `TQ3AntiAliasStyleData` structure whose pointer is passed as `antiAliasData`.

Managing a View's Attribute Set

QuickDraw 3D provides routines that you can use to manage a view's attribute set.

Q3View_GetDefaultAttributeSet

You can use the `Q3View_GetDefaultAttributeSet` function to get the default attribute set associated with a view.

```
TQ3Status Q3View_GetDefaultAttributeSet (
                    TQ3ViewObject view,
                    TQ3AttributeSet *attributeSet);
```

`view` A view.

`attributeSet` On exit, the default attribute set associated with the specified view.

DESCRIPTION

The `Q3View_GetDefaultAttributeSet` function returns, in the `attributeSet` parameter, the default attribute set of the view specified by the `view` parameter. QuickDraw 3D supplies a default set of attributes for every view so that you can safely render a view without having to set a value for each attribute. The default attribute values are defined by constants:

```
#define kQ3ViewDefaultAmbientCoefficient1.0
#define kQ3ViewDefaultDiffuseColor      0.5, 0.5, 0.5
#define kQ3ViewDefaultSpecularColor     0.5, 0.5, 0.5
#define kQ3ViewDefaultSpecularControl   4.0
#define kQ3ViewDefaultTransparency      1.0, 1.0, 1.0
```

```
#define kQ3ViewDefaultHighlightColor    1.0, 0.0, 0.0
#define kQ3ViewDefaultSubdivisionMethod kQ3SubdivisionMethodScreenSpace
#define kQ3ViewDefaultSubdivisionC1     20.0
#define kQ3ViewDefaultSubdivisionC2     20.0
```

Q3View_SetDefaultAttributeSet

You can use the `Q3View_SetDefaultAttributeSet` function to set the default attribute set associated with a view.

```
TQ3Status Q3View_SetDefaultAttributeSet (
                    TQ3ViewObject view,
                    TQ3AttributeSet attributeSet);
```

`view` A view.

`attributeSet` The default attribute set to be associated with the specified view.

DESCRIPTION

The `Q3View_SetDefaultAttributeSet` function sets the default attribute set of the view specified by the `view` parameter to the set specified in the `attributeSet` parameter.

Q3View_GetAttributeSetState

You can use the `Q3View_GetAttributeSetState` function to get the current attribute set associated with a view.

```
TQ3Status Q3View_GetAttributeSetState (
                    TQ3ViewObject view,
                    TQ3AttributeSet *attributeSet);
```

`view` A view.

`attributeSet` On exit, the attribute set currently associated with the specified view.

DESCRIPTION

The `Q3View_GetAttributeSetState` function returns, in the `attributeSet` parameter, the current attribute set of the view specified by the `view` parameter.

Q3View_GetAttributeState

You can use the `Q3View_GetAttributeState` function to get the state of a view's attribute.

```
TQ3Status Q3View_GetAttributeState (
                    TQ3ViewObject view,
                    TQ3AttributeType attributeType,
                    void *data);
```

`view` A view.

`attributeType` An attribute type.

`data` On exit, a pointer to the attribute data associated with the specified attribute type.

DESCRIPTION

The `Q3View_GetAttributeState` function returns, in the `data` parameter, a pointer to the attribute data associated with the attribute type specified by the `attributeType` parameter in the attribute set of the view specified by the `view` parameter. If the value `NULL` is returned in the `data` parameter, there is no attribute of the specified type in the view's attribute set.

Application-Defined Routines

QuickDraw 3D allows you to specify idle methods that QuickDraw 3D can call occasionally during lengthy operations. Two tools, `TQ3ViewIdleProgressMethod` and `TQ3ViewEndFrameMethod`, provide progress and end-of-frame information. They can be called only from a renderer plug-in module.

TQ3ViewIdleMethod

You can define an idle method to receive occasional callbacks to your application during lengthy operations.

```
typedef TQ3Status (*TQ3ViewIdleMethod) (
                    TQ3ViewObject view,
                    const void *idleData);
```

`view` A view.

`idleData` A pointer to an application-defined block of data.

DESCRIPTION

Your `TQ3ViewIdleMethod` function is called occasionally during lengthy operations, such as rendering a complex model. You can use an idle method to provide a means for the user to cancel the lengthy operation (for example, by clicking a button or pressing a key sequence such as Command-period).

If your idle method returns `kQ3Success`, QuickDraw 3D continues its current operation. If your idle method returns `kQ3Failure`, QuickDraw 3D cancels its current operation and returns `kQ3ViewStatusCancelled` the next time you call `Q3View_EndRendering` or a similar function. You should not call `Q3View_Cancel` (or any other QuickDraw 3D routine) inside your idle method.

There is currently no way to indicate how often you want your idle method to be called. You can read the time maintained by the Operating System if you need to determine the amount of time that has elapsed since your idle method was last called.

SPECIAL CONSIDERATIONS

You must not call any QuickDraw 3D routines inside your idle method. In particular, you must not change any of the settings of the view being rendered or call `Q3View_StartRendering` on that same view.

Some renderers (particularly those that use hardware accelerators) might not support idle methods.

TQ3ViewIdleProgressMethod

You can use the `TQ3ViewIdleProgressMethod` function to to register callback routines that the view can call during long operations. It helps provide data for a user interface indicator showing progress, and may also be used to interrupt long renderings or traversals. Within the idler callback code, the application can check for a cancel button or command key combination that lets the user interrupt rendering.

`TQ3ViewIdleProgressMethod` can be called only from a renderer plug-in module.

```
TQ3Status Q3View_SetIdleProgressMethod(
                    TQ3ViewObject view,
                    TQ3ViewIdleProgressMethod idleMethod,
                    const void *idleData);
```

`view` A view.

`idleMethod` An idle method (see below).

`idleData` A pointer to an application-defined block of data.

```
typedef TQ3Status (*TQ3ViewIdleProgressMethod)(
                    TQ3ViewObject    view,
                    const void       *idlerData,
                    unsigned long    current,
                    unsigned long    completed);
```

`view` A view.

`idlerData` A pointer to an application-defined block of data. This pointer is passed to the idle method when it is executed.

`current` Numerator of progress fraction. Its value is always less than the value of `completed`.

`completed` Denominator of progress fraction. The value of `current/completed` gives the degree of completion.

DESCRIPTION

`TQ3ViewIdleProgressMethod` registers a callback that also returns progress information. This information is supplied by the renderer, and may or may not

be based on real time. If a renderer doesn't support the progress method, your method will be called with `current` and `completed` both set to 0. Otherwise, you are certain to get called at least twice:

- once `idleMethod(view, 0, n)` Initialize, show dialog
- 0 or more times `idleMethod(view, 1..n-1, n)` Update progress
- once `idleMethod(view, n, n)` Exit, hide dialog

There is no way to set timer intervals when you want to be called—it is up to the application's idler callback to check clock times to see how long ago the application was called. `TQ3ViewIdleProgressMethod` returns `kQ3Failure` to cancel rendering, `kQ3Success` to continue. It does not post errors.

SPECIAL CONSIDERATIONS

Because your callback function may be called at hardware interrupt level, be careful about using Macintosh Toolbox routines. To call the Toolbox, you may want to set a global variable that you can later poll at noninterrupt level.

▲ **WARNING**
It is not legal to call QD3D routines inside an idler callback. ▲

TQ3ViewEndFrameMethod

You can use the `TQ3ViewEndFrameMethod` function to determine when an asynchronous renderer has completed rendering a frame.

`TQ3ViewEndFrameMethod` can be called only from a renderer plug-in module.

```
TQ3Status Q3View_SetEndFrameMethod(
                    TQ3ViewObject view,
                    TQ3ViewEndFrameMethod endFrame,
                    void *endFrameData);
```

`view` A view.

`endFrame` An end-of-frame method (see below).

`endFrameData` A pointer to an application-defined block of data.

```
typedef void (*TQ3ViewEndFrameMethod)(
                    TQ3ViewObject view,
                    void *endFrameData);
```

`view` A view.

`endFrameData` A pointer to an application-defined block of data.

DESCRIPTION

`TQ3ViewEndFrameMethod` provides an alternative to `Q3View_Sync` for determining when an asynchronous renderer has completed rendering a frame. With `Q3View_Sync`, the application asks a renderer to finish rendering a frame and blocks until the frame is complete. With `TQ3ViewEndFrameMethod`, the renderer tells the application that it has completed a frame.

IMPORTANT

If `Q3View_Sync` is called before `TQ3ViewEndFrameMethod` has been called, `TQ3ViewEndFrameMethod` will never be called. If `Q3View_Sync` is called after `TQ3ViewEndFrameMethod` has been called, `Q3View_Sync` will return immediately because the frame has already been completed. ▲

View Errors, Warnings, and Notices

The following errors, warnings, and notices may be returned by view object routines. A list of general QuickDraw 3D errors is given in "QuickDraw 3D Errors, Warnings, and Notices" (page 87).

`kQ3ErrorViewNotStarted`
`kQ3ErrorViewIsStarted`
`kQ3ErrorRendererNotSet`
`kQ3ErrorRenderingIsActive`
`kQ3ErrorImmediateModeUnderflow`
`kQ3ErrorDisplayNotSet`
`kQ3ErrorCameraNotSet`
`kQ3ErrorDrawContextNotSet`
`kQ3ErrorNonInvertibleMatrix`
`kQ3ErrorRenderingNotStarted`
`kQ3ErrorPickingNotStarted`
`kQ3ErrorBoundsNotStarted`
`kQ3ErrorDataNotAvailable`
`kQ3ErrorNothingToPop`
`kQ3WarningViewTraversalInProgress`
`kQ3WarningNonInvertibleMatrix`
`kQ3NoticeViewSyncCalledAgain`

CHAPTER 13

View Objects

CHAPTER 14

Shader Objects

This chapter describes shader objects (or shaders) and the functions you can use to manipulate them. You use shaders to provide shading and other effects to the objects in a model. For example, you can use a texture shader to apply a texture to the surface of an object in a model.

To use this chapter, you should already be familiar with views and lights, described in the chapters "View Objects" and "Light Objects" earlier in this book.

This chapter begins by describing shader objects and their features. Then it shows how to create and manipulate shaders. The section "Shader Objects Reference," beginning on page 928 provides a complete description of shader objects and the routines you can use to create and manipulate them.

About Shader Objects

A **shader object** (or, more briefly, a **shader**) is a type of QuickDraw 3D object that you can use to manipulate visual effects that depend on the illumination provided by a view's group of lights, the color and other material properties (such as the reflectance and texture) of surfaces in a model, and the position and orientation of the lights and objects in a model. Shaders that affect the surfaces of geometric objects based on their material properties, position, and orientation (and other factors) are **surface-based shaders.** QuickDraw 3D supplies several surface-based shaders, and you can define your own custom surface-based shaders to create other special effects. For instance, you can define a custom surface-based shader to handle custom attributes you have attached to surfaces or parts of surfaces.

The application of surface-based shaders occurs within the **QuickDraw 3D shading architecture,** an environment in which shaders can be applied at

various stages in the imaging pipeline. This architecture provides well-defined entry points at specific locations along the imaging pipeline. At each such location, you can invoke a shader. This capability allows you to create both two-dimensional and three-dimensional visual effects.

The QuickDraw 3D shading architecture is implemented using an object-based class hierarchy. For each location in the imaging pipeline at which a shader can be invoked, a subclass of the shader object has been defined. The following sections describe the available classes of shader objects.

Surface-Based Shaders

Several of the base classes of shaders apply shading effects to the surfaces of geometric objects.

- **Surface shaders** are applied when calculating the appearance of a surface. A geometric object (or group of geometric objects) can be associated with a surface shader, which is called to evaluate the shading effect for each face, vertex, or pixel of the object. QuickDraw 3D currently defines one subclass of surface shaders:
  - **Texture shaders** apply shading to an object using a texture. See "Textures" (page 922) for more information on textures and texture shaders.
- **Illumination shaders** determine the effects of the view's group of lights on the objects in a model. QuickDraw 3D currently defines three subclasses of illumination shaders. See "Illumination Models" (page 916) for more information on these illumination models.
  - The **Lambert illumination shader** implements a Lambert illumination model.
  - The **Phong illumination shader** implements a Phong illumination model.
  - The **null illumination shader** draws objects using only the diffuse colors of those objects, ignoring the view's group of lights.

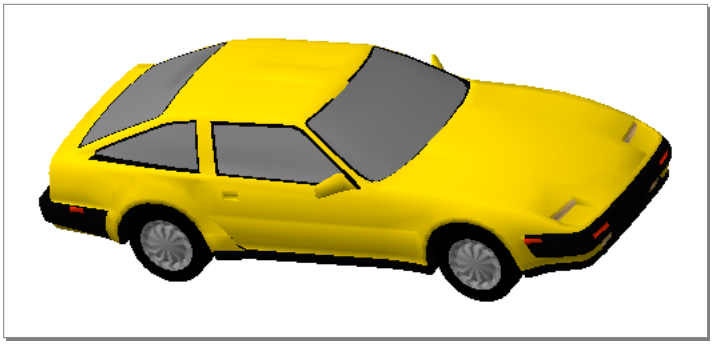
Illumination Models

As you've seen, an illumination shader determines the effects of a view's group of lights on the objects in a model. In order for the lights to have any effect, you must attach an illumination shader to the view. QuickDraw 3D provides three types of illumination shaders.

Lambert Illumination

The Lambert illumination shader implements an illumination model based on the diffuse reflection (also called the Lambertian reflection) of a surface. **Diffuse reflection** is characteristic of light reflected from a dull, nonshiny surface. Objects illuminated solely by diffusely reflected light exhibit an equal light intensity from all viewing directions. Figure 14-1 shows an object illuminated using the Lambert illumination shader. See also Color Plate 4 at the beginning of this book.

Figure 14-1 Effects of the Lambert illumination shader



For a point on a surface, the Lambert illumination provided by i distinct lights is given by the following equation:

$$I_{Lambert} = I_a k_a O_d + \sum_i (N \bullet L_i) I_i k_d O_d$$

Here, I_a is the intensity of the ambient light, and k_a is the ambient coefficient. O_d is the diffuse color of the surface of the object being illuminated. N is the surface normal vector at the point whose illumination is being evaluated, and L_i is a normalized vector indicating the direction to the ith light source. Notice that if the dot product $(N \cdot L_i)$ is 0 for a particular light (that is, if N and L_i are perpendicular), that light contributes nothing to the illumination of the point. I_i is the intensity of the ith light source, and k_d is the **diffuse coefficient** of the surface being illuminated (that is, the level of diffuse reflection of the surface).

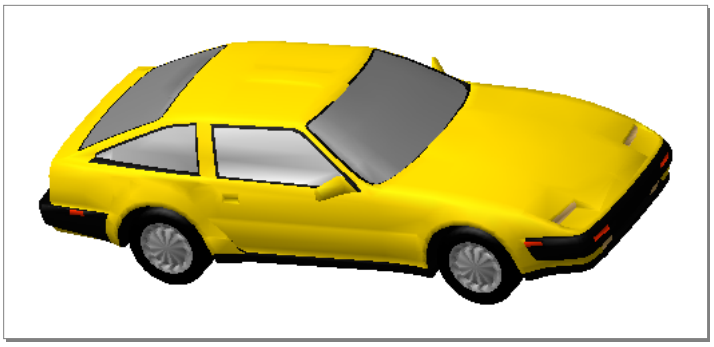
As you can see, the intensity of the light reflected by a point on a surface depends solely on the ambient light and the diffuse reflection of the surface at that point.

Note
QuickDraw 3D does not currently provide a way to set the value of the diffuse coefficient of a surface directly. Instead, you must use the product $k_d O_d$ as the surface's diffuse color. You specify a diffuse color by inserting an attribute of type `kQ3AttributeTypeDiffuseColor` into the surface's attribute set. ◆

Phong Illumination

The Phong illumination shader implements an illumination model based on both diffuse reflection and specular reflection of a surface. **Specular reflection** is characteristic of light reflected from a shiny surface, where a bright highlight appears from certain viewing directions. Figure 14-2 shows an object illuminated using the Phong illumination shader. See also Color Plate 4 at the beginning of this book.

Figure 14-2 Effects of the Phong illumination shader



For a point on a surface, the Phong illumination provided by *i* distinct lights is given by the following equation:

$$I_{Phong} = I_a k_a O_d + \sum_i [((N \bullet L_i) I_i k_d O_d) + ((R \bullet V)^n k_s)]$$

Notice that the Phong illumination equation is simply the Lambert illumination equation with an additional summand to account for specular reflection. Here, R is the direction of reflection and V is the direction of viewing. The exponent n is the specular reflection exponent, and k_s is the specular reflection coefficient. The **specular reflection exponent** determines how quickly the specular reflection diminishes as the viewing direction moves away from the direction of reflection. In other words, the specular reflection exponent determines the size of the **specular highlight** (a bright area on the surface of the object caused by specular reflection). When the value of n is small, the size of the specular highlight is large; as n increases, the size of the specular highlight shrinks.

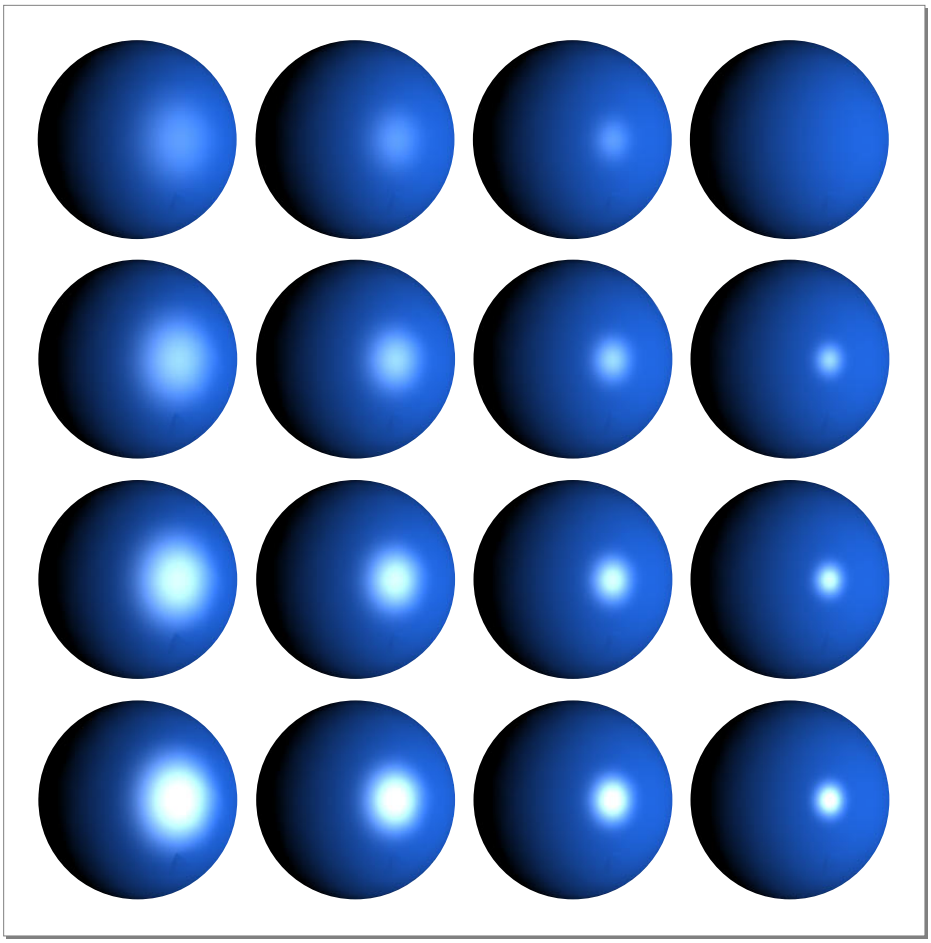
Note
Note that setting the specular reflection exponent to 0 results in no specular reflection (because $n^0 = 1$ for any number n). Moreover, values between 0 and 1 reduce the amount of specular reflection. In general, the specular reflection exponent should be a value greater than or equal to 1. ◆

The **specular coefficient** (or **specular reflection coefficient**), symbolized by k_s in the equation above, indicates the level of the object's specular reflection. It controls the overall brightness of the specular highlight, independent of the brightness of the light sources and the direction of viewing.

Figure 14-3 shows an object illuminated using a variety of values for the specular reflection exponent and the specular coefficient. In this figure, the specular reflection exponent increases from left to right, resulting in a smaller specular highlight. In addition, the specular coefficient increases from top to bottom, resulting in a brighter specular highlight.

Note
A surface's specular reflection coefficient is also called its **specular control.** You specify a specular reflection coefficient by inserting an attribute of type `kQ3AttributeTypeSpecularControl` into the surface's attribute set. ◆

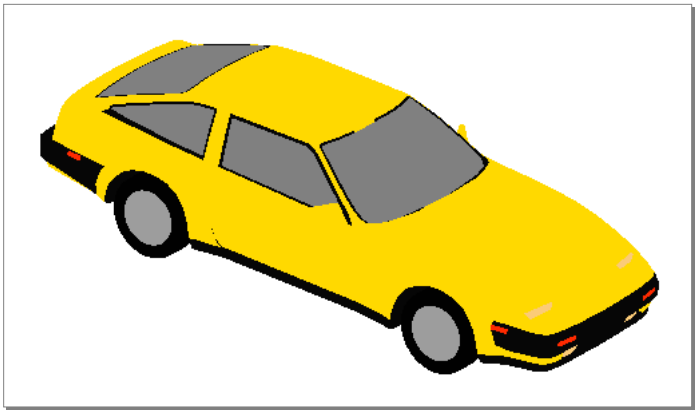
Figure 14-3 Phong illumination with various specular exponents and coefficients



Null Illumination

The null illumination shader ignores the lights in a view's light group and configures the renderer to draw all objects using only the diffuse colors of those objects. The net effect of the this shader is to draw objects as if the only light source was an ambient light at full intensity. Figure 14-4 shows an object illuminated using the null illumination shader.

Figure 14-4 Effects of the null illumination shader



For any point on a surface, the null illumination is given by the following equation:

$$I_{null} = O_d$$

Here, O_d is the diffuse color of the surface of the object being illuminated. As you can see, when the null illumination shader is active, all facets of an object are drawn the same color (unless different facets have attribute sets that override the diffuse color of the object).

Textures

As indicated earlier, QuickDraw 3D supports texture shaders that allow you to perform **texture mapping,** a technique wherein a predefined image (the texture) is mapped onto the surface of an object in a model. For instance, you can create a wood-grain image and map it onto objects in a model to give those objects a wooden appearance. Similarly, you can digitize an image of a person and apply it, using a texture shader, to the face of an object to create a picture, in the model, of that person. In general, you'll use texture shaders to create realistic-looking surfaces (such as wood, stone, or cloth) in your models.

You create a texture shader by calling `Q3TextureShader_New`, passing it a **texture object** (or, more briefly, a **texture**). QuickDraw 3D provides a number of functions that you can use to create and manipulate texture objects. Currently QuickDraw 3D supports one subclass of texture objects, **pixmap texture objects,** which are images defined by pixmaps. You call `Q3PixmapTexture_New` to create a new texture object from a pixmap.

Note
See the chapter "Geometric Objects" for information on pixmaps. ◆

Once you've created a texture from a pixmap, you need to attach the texture to surfaces in your model. See "Using Texture Shaders" (page 923) for details.

Using Shader Objects

QuickDraw 3D supplies routines that you use to create and configure shader objects. You can make a shader's effects appear in a rendered image in several ways. You can submit the shader inside a rendering loop, or you can add the shader to a group and submit the group inside a rendering loop. These ways of applying a shader are all equally good, and which of them you use depends on the circumstances. For instance, if you put a shader object into an unordered display group, it will affect only the objects following it in the group.

Using Illumination Shaders

You create an illumination shader by calling the `_New` function for the type of illumination model you want to use. For example, to use Phong illumination, you can call the `Q3PhongIllumination_New` function.

Once you've created an illumination shader, you apply it to the objects in a model by submitting the shader inside of a submitting loop, or by adding it to a group that is submitted in a submitting loop. For instance, to apply Phong illumination to all the objects in a model, you can call the function `Q3Shader_Submit` in your rendering loop, as shown in Listing 14-1.

Listing 14-1 Applying an illumination shader

```
Q3View_StartRendering(myView);
do {
    Q3Shader_Submit(myPhongShader, myView);

    /*submit styles, groups, and other objects here*/

    myViewStatus = Q3View_EndRendering(myView);
} while (myViewStatus == kQ3ViewStatusRetraverse);
```

Using Texture Shaders

You create a texture shader by calling the `Q3TextureShader_New` function, to which you pass a texture object. QuickDraw 3D currently supports only pixmap texture objects, which you create by calling the `Q3PixMapTexture_New` function.

Once you've created a texture shader, you can apply it to all the objects in a model by submitting the shader inside of a rendering loop, as shown in Listing 14-2.

Listing 14-2 Applying a texture shader in a submitting loop

```
Q3View_StartRendering(myView);
do {
    Q3Shader_Submit(myTextureShader, myView);
```

```
    /*submit styles, groups, and other objects here*/

    myViewStatus = Q3View_EndRendering(myView);
} while (myViewStatus == kQ3ViewStatusRetraverse);
```

You can apply the shader to the objects in a group by adding it to a group that is submitted in a rendering loop, as shown in Listing 14-3. (The `myGroup` group is an ordered display group.)

Listing 14-3 Applying a texture shader in a group

```
Q3Group_AddObject(myGroup, myTextureShader);

Q3View_StartRendering(myView);
do {
    Q3Group_Submit(myGroup, myView);
    myViewStatus = Q3View_EndRendering(myView);
} while (myViewStatus == kQ3ViewStatusRetraverse);
```

You can also apply a texture shader to all the objects in a model by adding the shader as an attribute of type `kQ3AttributeTypeSurfaceShader` to the view's attribute set. Similarly, you can attach the texture shader to a part of a geometric object as an attribute. For example, you can attach a texture shader to the face of a cube or a mesh to have that face shaded with a texture. Listing 14-4 illustrates how to create a texture shader and use it to shade a triangle. Note that the function `MyCreateShadedTriangle` defined in Listing 14-4 sets up a custom surface parameterization for the triangle, because there is no standard surface parameterization for a triangle.

Listing 14-4 Applying a texture shader as an attribute

```
TQ3GeometryObject MyCreateShadedTriangle (TQ3StoragePixmap myPixmap)
{
    TQ3ShaderObject                 myShader;
    TQ3TextureObject                myTexture;
    TQ3TriangleData                 myTriData;
    TQ3GeometryObject               myTriangle;
```

```
TQ3Param2D                          myParam2D;
TQ3Vertex3D                         myVertices[3] = {
        { { 0.5,  0.5, 0.0}, NULL },
        { {-0.5,  0.5, 0.0}, NULL },
        { {-0.5, -0.5, 0.0}, NULL }};

/*Create a new texture from the pixmap passed in.*/
myTexture = Q3PixmapTexture_New(&myPixmap);
if (myTexture == NULL)
    return (NULL);
Q3Object_Dispose(myPixmap.image);

/*Create a new texture shader from the texture.*/
myShader = Q3TextureShader_New(myTexture);
if (myShader == NULL)
    return (NULL);
Q3Object_Dispose(myTexture);

/*Configure triangle data.*/
/*First, attach uv values to the three vertices.*/
myParam2D.u = 0;
myParam2D.v = 0;
myVertices[0].attributeSet = Q3AttributeSet_New();
Q3AttributeSet_Add(myVertices[0].attributeSet, kQ3AttributeTypeShadingUV,
                                            &myParam2D);
myParam2D.u = 0;
myParam2D.v = 1;
myVertices[1].attributeSet = Q3AttributeSet_New();
Q3AttributeSet_Add(myVertices[1].attributeSet, kQ3AttributeTypeShadingUV,
                                            &myParam2D);
myParam2D.u = 1;
myParam2D.v = 1;
myVertices[2].attributeSet = Q3AttributeSet_New();
Q3AttributeSet_Add(myVertices[2].attributeSet, kQ3AttributeTypeShadingUV,
                                            &myParam2D);

/*Define the triangle, using the vertices and uv values just set up.*/
myTriData.vertices[0] = myVertices[0];
myTriData.vertices[1] = myVertices[1];
myTriData.vertices[2] = myVertices[2];
```

```
    /*Attach a texture surface shader as an attribute.*/
    myTriData.triangleAttributeSet = Q3AttributeSet_New();
    Q3AttributeSet_Add(myTriData.triangleAttributeSet,
                                        kQ3AttributeTypeSurfaceShader, &myShader);

    myTriangle = Q3Triangle_New(&myTriData);
    Q3Object_Dispose(myVertices[0].attributeSet);
    Q3Object_Dispose(myVertices[1].attributeSet);
    Q3Object_Dispose(myVertices[2].attributeSet);

    return(myTriangle);
}
```

The function `MyCreateShadedTriangle` defined in Listing 14-4 creates a texture from the pixmap it is passed and then creates a new texture shader from that texture. `MyCreateShadedTriangle` then attaches *uv* parameterization values to each of the three triangle vertices and defines the triangle data. Finally, `MyCreateShadedTriangle` creates a triangle and returns it to its caller. When the triangle is drawn (perhaps by being submitted in a rendering loop), it will have the specified texture mapped onto it.

Creating Storage Pixmaps

The data passed to the `Q3PixmapTexture_New` function (as in Listing 14-4 (page 924)) is a storage pixmap, of type `TQ3StoragePixmap`. The `image` field of a storage pixmap specifies a storage object that contains the pixmap data to be applied as a texture. You can call either `Q3MemoryStorage_New` or `Q3MemoryStorage_NewBuffer` to create a storage object. Which function you use depends on whether (1) you want QuickDraw 3D to maintain the image data in an internal buffer or (2) you want to maintain the data in your own buffer.

To let QuickDraw 3D manage the pixmap data, you can assign the `image` field of a storage pixmap using code like this:

```
myStoragePixmap.image = Q3MemoryStorage_New(myBuffer, mySize);
```

This code asks QuickDraw 3D to allocate a buffer internally, of the specified size. Once `Q3MemoryStorage_New` returns successfully, you can dispose of the buffer `myBuffer`, because QuickDraw 3D has copied the texture pixmap data into its own internal memory.

If you prefer, you can maintain the pixmap data in your application's memory partition and avoid the overhead of having the data copied to internal QuickDraw 3D memory. (This is especially useful if you want to animate a texture by changing the texture pixmap data from frame to frame.) To do this, you create a storage object by calling the `Q3MemoryStorage_NewBuffer` function, like this:

```
myStoragePixmap.image = Q3MemoryStorage_NewBuffer
                                        (myBuffer, mySize, mySize);
```

In this case, you should *not* dispose of the data buffer. You can change the pixmap data by calling `Q3MemoryStorage_SetBuffer`.

```
Q3MemoryStorage_SetBuffer
               (myStoragePixmap.image, myBuffer, mySize, mySize);
```

You need to call `Q3MemoryStorage_SetBuffer` to force QuickDraw 3D to update any caches.

Note
You can also change the data of a storage object created by a call to `Q3MemoryStorage_New`, by calling `Q3MemoryStorage_Set`. ◆

Handling *uv* Values Outside the Valid Range

As you've seen, a *uv* parameterization defines how to map one object (for example, a pixmap) onto another (typically a surface). The standard surface parameterizations defined by QuickDraw 3D all use *u* and *v* parametric values that are in the **valid range** 0.0 to 1.0. A custom surface parameterization, however, is free to define some other range of *u* and *v* values. When this happens, you need to indicate how you want QuickDraw 3D to handle *uv* values outside the valid range.

Currently, QuickDraw 3D supports two boundary-handling methods: wrapping and clamping. To **wrap** a shader effect is to replicate the entire effect across the mapped area. For example, to wrap a texture is to replicate the texture across the entire mapped area, as many times as are necessary to fill the mapped area. To **clamp** a shader effect is to replicate the *boundaries* of the effect across the portion of the mapped area that lies outside the valid range 0.0 to 1.0.

You can specify the boundary-handling methods of the *u* and *v* directions independently. You can call the `Q3Shader_SetUBoundary` function to indicate how to handle values in the *u* parametric direction that lie outside the valid range, and you can call the `Q3Shader_SetVBoundary` function to indicate how to handle values in the *v* parametric direction that lie outside the valid range. The default boundary-handling method is to wrap in both the *u* and *v* parametric directions.

Shader Objects Reference

This section describes the constants, data structures, and routines you can use to create and manipulate shaders and textures.

Constants

This section describes the constants that you use to specify *uv* boundary-handling methods.

Boundary-Handling Methods

You use a boundary-handling method specifier to indicate how you want a shader to handle *uv* values that are outside the valid range (namely, 0 to 1). For example, you pass one of these constants to the `Q3Shader_SetUBoundary` function to indicate how to handle values in the *u* parametric direction that lie outside the valid range.

Note
For a fuller description of boundary-handling methods, see "Handling uv Values Outside the Valid Range," beginning on page 927. ◆

```
typedef enum TQ3ShaderUVBoundary {
    kQ3ShaderUVBoundaryWrap,
    kQ3ShaderUVBoundaryClamp
} TQ3ShaderUVBoundary;
```

Constant descriptions

`kQ3ShaderUVBoundaryWrap`
: Values outside the valid range are to be wrapped. To wrap a shader effect is to replicate the entire effect across the mapped area. For example, for a texture shader, wrapping causes the entire image to be replicated across the surface onto which the texture is mapped.

`kQ3ShaderUVBoundaryClamp`
: Values outside the valid range are to be clamped. To clamp a shader effect is to replicate the boundaries of the effect across the portion of the mapped area that lies outside the valid range. For example, for a texture shader, clamping causes boundaries of the image to be smeared across the portion of the surface onto which the texture is mapped that lies outside the valid range.

Shader Objects Routines

This section describes the routines you can use to manage shaders and textures.

Managing Shaders

QuickDraw 3D provides routines that you can use to manage shaders.

Q3Shader_GetType

You can use the `Q3Shader_GetType` function to get the type of a shader object.

```
TQ3ObjectType Q3Shader_GetType (TQ3ShaderObject shader);
```

`shader` A shader object.

DESCRIPTION

The `Q3Shader_GetType` function returns, as its function result, the type of the shader object specified by the `shader` parameter. The types of shader objects currently supported by QuickDraw 3D are defined by these constants:

```
kQ3ShaderTypeSurface
kQ3ShaderTypeIllumination
```

If the specified shader object is invalid or is not one of these types, `Q3Shader_GetType` returns the value `kQ3ObjectTypeInvalid`.

Q3Shader_Submit

You can use the `Q3Shader_Submit` function to submit a shader in a view.

```
TQ3Status Q3Shader_Submit (
                    TQ3ShaderObject shader,
                    TQ3ViewObject view);
```

`shader` A shader.

`view` A view.

DESCRIPTION

The `Q3Shader_Submit` function submits the shader specified by the `shader` parameter for drawing or writing in the view specified by the `view` parameter.

SPECIAL CONSIDERATIONS

You should call this function only in a submitting loop.

Managing Shader Characteristics

QuickDraw 3D provides routines for getting and setting characteristics that define how a shader affects a surface.

Q3Shader_GetUVTransform

You can use the `Q3Shader_GetUVTransform` function to get the current transform in *uv* parametric space.

```
TQ3Status Q3Shader_GetUVTransform (
                    TQ3ShaderObject shader,
                    TQ3Matrix3x3 *uvTransform);
```

`shader` A shader.

`uvTransform` On exit, a pointer to the current transform in *uv* parametric space.

DESCRIPTION

The `Q3Shader_GetUVTransform` function returns, in the `uvTransform` parameter, the current transform in *uv* parametric space for the shader specified by the `shader` parameter.

Q3Shader_SetUVTransform

You can use the `Q3Shader_SetUVTransform` function to set the transform in *uv* parametric space.

```
TQ3Status Q3Shader_SetUVTransform (
                    TQ3ShaderObject shader,
                    const TQ3Matrix3x3 *uvTransform);
```

`shader` A shader.

`uvTransform` A pointer to the desired transform in *uv* parametric space.

DESCRIPTION

The `Q3Shader_SetUVTransform` function sets the transform in *uv* parametric space for the shader specified by the `shader` parameter to the transform specified by the `uvTransform` parameter. For example, a texture shader that relies on *uv*

values to index a texture mapping can rotate, scale, or translate the texture by setting appropriate values in the *uv* transform.

Q3Shader_GetUBoundary

You can use the `Q3Shader_GetUBoundary` function to get the current boundary-handling method for *u* values that are outside the range 0 to 1.

```
TQ3Status Q3Shader_GetUBoundary (
                    TQ3ShaderObject shader,
                    TQ3ShaderUVBoundary *uBoundary);
```

`shader` A shader.

`uBoundary` On exit, a value that indicates the current method of handling *u* values that are outside the range 0 to 1. See "Boundary-Handling Methods" (page 928) for a description of the values that can be returned.

DESCRIPTION

The `Q3Shader_GetUBoundary` function returns, in the `uBoundary` parameter, the current method used by the shader specified by the `shader` parameter of handling *u* values that are outside the range 0 to 1. If `Q3Shader_GetUBoundary` completes successfully, the `uBoundary` parameter contains one of these values:

```
typedef enum TQ3ShaderUVBoundary {
    kQ3ShaderUVBoundaryWrap,
    kQ3ShaderUVBoundaryClamp
} TQ3ShaderUVBoundary;
```

Q3Shader_SetUBoundary

You can use the `Q3Shader_SetUBoundary` function to set the current boundary-handling method for *u* values that are outside the range 0 to 1.

```
TQ3Status Q3Shader_SetUBoundary (
                    TQ3ShaderObject shader,
                    TQ3ShaderUVBoundary uBoundary);
```

`shader` A shader.

`uBoundary` A value that indicates the desired method of handling *u* values that are outside the range 0 to 1. See "Boundary-Handling Methods" (page 928) for a description of the values that you can pass in this parameter.

DESCRIPTION

The `Q3Shader_SetUBoundary` function sets the boundary-handling method for *u* values to be used by the shader specified by the `shader` parameter to the method specified by the `uBoundary` parameter.

Q3Shader_GetVBoundary

You can use the `Q3Shader_GetVBoundary` function to get the current boundary-handling mode for *v* values that are outside the range 0 to 1.

```
TQ3Status Q3Shader_GetVBoundary (
                    TQ3ShaderObject shader,
                    TQ3ShaderUVBoundary *vBoundary);
```

`shader` A shader.

`vBoundary` On exit, a value that indicates the current method of handling *v* values that are outside the range 0 to 1. See "Boundary-Handling Methods" (page 928) for a description of the values that can be returned.

DESCRIPTION

The `Q3Shader_GetVBoundary` function returns, in the `vBoundary` parameter, the current method used by the shader specified by the `shader` parameter of handling *v* values that are outside the range 0 to 1. If `Q3Shader_GetVBoundary` completes successfully, the `vBoundary` parameter contains one of these values:

```
typedef enum TQ3ShaderUVBoundary {
    kQ3ShaderUVBoundaryWrap,
    kQ3ShaderUVBoundaryClamp
} TQ3ShaderUVBoundary;
```

Q3Shader_SetVBoundary

You can use the `Q3Shader_SetVBoundary` function to set the current boundary-handling mode for *v* values that are outside the range 0 to 1.

```
TQ3Status Q3Shader_SetVBoundary (
                    TQ3ShaderObject shader,
                    TQ3ShaderUVBoundary vBoundary);
```

`shader` A shader.

`vBoundary` A value that indicates the desired method of handling *v* values that are outside the range 0 to 1. See "Boundary-Handling Methods" (page 928) for a description of the values that you can pass in this parameter.

DESCRIPTION

The `Q3Shader_SetVBoundary` function sets the boundary-handling method for *v* values to be used by the shader specified by the `shader` parameter to the method specified by the `vBoundary` parameter.

Managing Surface Shaders

QuickDraw 3D provides routines that you can use to manage surface shaders.

Q3SurfaceShader_GetType

You can use the `Q3SurfaceShader_GetType` function to get the type of a surface shader.

```
TQ3ObjectType Q3SurfaceShader_GetType (TQ3SurfaceShaderObject shader);
```

`shader` A surface shader.

DESCRIPTION

The `Q3SurfaceShader_GetType` function returns, as its function result, the type of the surface shader specified by the `shader` parameter. The types of surface shaders currently supported by QuickDraw 3D are defined by these constants:

```
kQ3SurfaceShaderType_Texture
```

If the specified surface shader is invalid or is not one of these types, `Q3SurfaceShader_GetType` returns the value `kQ3ObjectTypeInvalid`.

Managing Texture Shaders

QuickDraw 3D provides routines that you can use to create and manage texture shaders.

Q3TextureShader_New

You can use the `Q3TextureShader_New` function to create a new texture shader.

```
TQ3ShaderObject Q3TextureShader_New (TQ3TextureObject texture);
```

`texture` A texture object.

DESCRIPTION

The `Q3TextureShader_New` function returns, as its function result, a new texture shader that uses the texture specified by the `texture` parameter. If

Q3TextureShader_New cannot create a new texture shader, it returns the value NULL.

Q3TextureShader_GetTexture

You can use the Q3TextureShader_GetTexture function to get the texture associated with a texture shader.

```
TQ3Status Q3TextureShader_GetTexture (
                    TQ3ShaderObject shader,
                    TQ3TextureObject *texture);
```

shader	A texture shader.
texture	On exit, the texture object currently associated with the specified texture shader.

DESCRIPTION

The Q3TextureShader_GetTexture function returns, in the texture parameter, the texture object currently associated with the texture shader specified by the shader parameter.

Q3TextureShader_SetTexture

You can use the Q3TextureShader_SetTexture function to set the texture associated with a texture shader.

```
TQ3Status Q3TextureShader_SetTexture (
                    TQ3ShaderObject shader,
                    TQ3TextureObject texture);
```

shader	A texture shader.
texture	The texture object to be associated with the specified texture shader.

DESCRIPTION

The `Q3TextureShader_SetTexture` function sets the texture object associated with the texture shader specified by the `shader` parameter to the texture specified by the `texture` parameter.

Managing Illumination Shaders

QuickDraw 3D provides routines that you can use to create and manage illumination shaders. QuickDraw 3D supplies two types of illumination shaders, Lambert illumination shaders and Phong illumination shaders.

Q3LambertIllumination_New

You can use the `Q3LambertIllumination_New` function to create a new illumination shader that provides Lambert illumination.

```
TQ3ShaderObject Q3LambertIllumination_New (void);
```

DESCRIPTION

The `Q3LambertIllumination_New` function returns, as its function result, a new illumination shader that implements a Lambert illumination model. See “Illumination Models” (page 916) for information on the Lambert illumination algorithm.

Q3PhongIllumination_New

You can use the `Q3PhongIllumination_New` function to create a new illumination shader that provides Phong illumination.

```
TQ3ShaderObject Q3PhongIllumination_New (void);
```

DESCRIPTION

The `Q3PhongIllumination_New` function returns, as its function result, a new illumination shader that implements a Phong illumination model. See "Illumination Models" (page 916) for information on the Phong illumination algorithm.

Q3NULLIllumination_New

You can use the `Q3NULLIllumination_New` function to create a new null illumination shader.

```
TQ3ShaderObject Q3NULLIllumination_New (void);
```

DESCRIPTION

The `Q3NULLIllumination_New` function returns, as its function result, a new null illumination shader.

Q3IlluminationShader_GetType

You can use the `Q3IlluminationShader_GetType` function to get the type of an illumination shader.

```
TQ3ObjectType Q3IlluminationShader_GetType (
                    TQ3ShaderObject shader);
```

`shader` An illumination shader.

DESCRIPTION

The `Q3IlluminationShader_GetType` function returns, as its function result, the type of the illumination shader specified by the `shader` parameter. The types of illumination shaders currently supported by QuickDraw 3D are defined by these constants:

```
kQ3IlluminationTypeLambert
kQ3IlluminationTypePhong
kQ3IlluminationTypeNULL
```

If the specified illumination shader is invalid or is not one of these types, `Q3IlluminationShader_GetType` returns the value `kQ3ObjectTypeInvalid`.

Managing Textures

QuickDraw 3D provides routines that you can use to get information about the characteristics of a texture. You can get the dimensions of a texture, as well as the number of channels and the number of bits per channel. You cannot, however, reset any of these texture characteristics (they are determined at the time you create a texture object). You can also get the current alpha and RGB channels of a texture. You can reset these characteristics to achieve special effects.

Note
To create a texture object, you need to create an instance of some subclass of the texture class. For example, you can create a pixmap texture object by calling `Q3PixmapTexture_New`. See "Managing Pixmap Textures" (page 941) for information on creating and manipulating pixmap textures. ◆

Q3Texture_GetType

You can use the `Q3Texture_GetType` function to get the type of a texture object.

```
TQ3ObjectType Q3Texture_GetType (TQ3TextureObject texture);
```

`texture` A texture object.

DESCRIPTION

The `Q3Texture_GetType` function returns, as its function result, the type of the texture object specified by the `texture` parameter. The type of texture objects currently supported by QuickDraw 3D is defined by these constants:

```
kQ3TextureTypePixmap
kQ3TextureTypeMipmap
```

If the specified texture object is invalid or is not of this type, `Q3Texture_GetType` returns the value `kQ3ObjectTypeInvalid`.

Q3Texture_GetWidth

You can use the `Q3Texture_GetWidth` function to get the width of a texture.

```
TQ3Status Q3Texture_GetWidth (
                    TQ3TextureObject texture,
                    unsigned long *width);
```

`texture` A texture object.

`width` On exit, the width of the specified texture.

DESCRIPTION

The `Q3Texture_GetWidth` function returns, in the `width` parameter, the width of the texture specified by the `texture` parameter.

Q3Texture_GetHeight

You can use the `Q3Texture_GetHeight` function to get the height of a texture.

```
TQ3Status Q3Texture_GetHeight (
                    TQ3TextureObject texture,
                    unsigned long *height);
```

`texture` A texture object.

`height` On exit, the height of the specified texture.

DESCRIPTION

The `Q3Texture_GetHeight` function returns, in the `height` parameter, the height of the texture specified by the `texture` parameter.

Managing Pixmap Textures

QuickDraw 3D provides routines that you can use to create and manipulate pixmap textures.

Q3PixmapTexture_New

You can use the `Q3PixmapTexture_New` function to create a new pixmap texture.

```
TQ3TextureObject Q3PixmapTexture_New (const TQ3StoragePixmap *pixmap);
```

`pixmap` A storage pixmap.

DESCRIPTION

The `Q3PixmapTexture_New` function returns, as its function result, a new texture object that uses the storage pixmap specified by the `pixmap` parameter. If `Q3PixmapTexture_New` cannot create a new pixmap texture object, it returns the value `NULL`.

Q3PixmapTexture_GetPixmap

You can use the `Q3PixmapTexture_GetPixmap` function to get the pixmap associated with a pixmap texture object.

```
TQ3Status Q3PixmapTexture_GetPixmap (
                    TQ3TextureObject texture,
                    TQ3StoragePixmap *pixmap);
```

`texture` A pixmap texture object.

`pixmap` On exit, the storage pixmap currently associated with the specified pixmap texture object.

DESCRIPTION

The `Q3PixmapTexture_GetPixmap` function returns, in the `pixmap` parameter, the pixmap currently associated with the pixmap texture object specified by the `texture` parameter.

Q3PixmapTexture_SetPixmap

You can use the `Q3PixmapTexture_SetPixmap` function to set the pixmap associated with a pixmap texture object.

```
TQ3Status Q3PixmapTexture_SetPixmap (
                    TQ3TextureObject texture,
                    const TQ3StoragePixmap *pixmap);
```

`texture` A pixmap texture object.

`pixmap` The storage pixmap to be associated with the specified pixmap texture object.

DESCRIPTION

The `Q3PixmapTexture_SetPixmap` function sets the pixmap to be associated with the pixmap texture object specified by the `texture` parameter to the pixmap specified by the `pixmap` parameter.

Managing Mipmap Textures

QuickDraw 3D provides routines that you can use to create and manipulate mipmap textures. A mipmap is stored in a structure of type `TQ3Mipmap`, which may contain up to 32 images of type `TQ3MipmapImage`:

```
typedef struct TQ3Mipmap {
    TQ3StorageObject    image;
    TQ3Boolean          useMipmapping;
```

```
    TQ3PixelType        pixelType;
    TQ3Endian           bitOrder;
    TQ3Endian           byteOrder;
    unsigned long       reserved;       /* NULL */
    TQ3MipmapImage      mipmaps[32];
} TQ3Mipmap;

typedef struct TQ3MipmapImage {
    unsigned long       width;
    unsigned long       height;
    unsigned long       rowBytes;
    unsigned long       offset;
} TQ3MipmapImage;
```

Field descriptions

`image`	A storage object containing the texture map; if `useMipmapping` is `kQ3True`, it contains the mipmap data.
`useMipmapping`	`kQ3True` if mipmapping should be used and all mipmaps are provided.
`mipmaps`	Images of type `TQ3MipmapImage`. The actual number of mipmaps is determined by the size of the first mipmap.
`width`	Width of the mipmap; must be a power of 2.
`height`	Height of the mipmap; must be a power of 2.
`rowBytes`	Rowbytes of the mipmap.
`offset`	Offset from the image base to this mipmap.

Q3MipmapTexture_New

You can use the `Q3MipmapTexture_New` function to create a new mipmap texture.

```
TQ3TextureObject Q3MipmapTexture_New (const TQ3Mipmap *mipmap);
```

`mipmap` A mipmap.

DESCRIPTION

The `Q3MipmapTexture_New` function returns, as its function result, a new texture object that uses the mipmap specified by the `mipmap` parameter. If `Q3MipmapTexture_New` cannot create a new mipmap texture object, it returns the value `NULL`.

Q3MipmapTexture_GetMipmap

You can use the `Q3MipmapTexture_GetMipmap` function to get the mipmap associated with a mipmap texture object.

```
TQ3Status Q3MipmapTexture_GetMipmap (
                    TQ3TextureObject texture,
                    TQ3Mipmap *mipmap);
```

`texture` A mipmap texture object.

`mipmap` On exit, the mipmap currently associated with the specified mipmap texture object.

DESCRIPTION

The `Q3MipmapTexture_GetMipmap` function returns, in the `mipmap` parameter, the mipmap currently associated with the mipmap texture object specified by the `texture` parameter.

Q3MipmapTexture_SetMipmap

You can use the `Q3MipmapTexture_SetMipmap` function to set the mipmap associated with a mipmap texture object.

```
TQ3Status Q3MipmapTexture_SetMipmap (
                    TQ3TextureObject texture,
                    const TQ3Mipmap *mipmap);
```

`texture` A mipmap texture object.

`mipmap`	The mipmap to be associated with the specified mipmap texture object.

DESCRIPTION

The `Q3MipmapTexture_SetMipmap` function sets the mipmap to be associated with the mipmap texture object specified by the `texture` parameter to the mipmap specified by the `mipmap` parameter.

CHAPTER 14

Shader Objects

CHAPTER 15

Pick Objects

This chapter describes pick objects and the functions you can use to manipulate them. You use pick objects to get a list of objects in a view that intersect a specified geometric object (for example, objects the user has selected in an image on the screen).

To use this chapter, you should already be familiar with the QuickDraw 3D class hierarchy, described in the chapter "QuickDraw 3D Objects." For information about views, see the chapter "View Objects." You do not, however, need to know how to create or manipulate views.

This chapter begins by describing pick objects and their features. Then it shows how to create and use pick objects. The section "Pick Objects Reference," beginning on page 961 provides a complete description of pick objects and the routines you can use to create and manipulate them.

About Pick Objects

Picking is the process of identifying the objects in a view that are close to a specified geometric object. You might, for example, want to determine which objects in a view, if any, are sufficiently close to a particular ray. You'll use picking primarily to allow users to select objects in a view. Picking thereby provides the foundation for user interaction with three-dimensional models. You can, however, use picking for other purposes. You might, for example, use picking to determine which objects in a model are visible from a particular camera location.

Screen-space picking (or **window picking**) involves testing whether the projections of three-dimensional objects onto the screen intersect or are close enough to a specified two-dimensional object on the screen.

QuickDraw 3D returns information about the picked geometric objects as they are defined in three-dimensional space. For example, you might want to know the distance of a picked object from some point. The distance reported by QuickDraw 3D is always a three-dimensional world-space distance, not a two-dimensional screen-space distance.

You perform a picking operation by creating a **pick object** (or, more briefly, a **pick**). QuickDraw 3D provides a variety of routines that you can use to create pick objects, depending on the desired picking method. For example, you can call `Q3WindowPointPick_New` to create a pick object that selects objects in a view whose projections onto the screen are close enough to a particular point. The geometric object used in any picking method is the **pick geometry.**

To get the objects in the model that are close to the pick geometry, you must submit the entire model. The code you use to do this is similar to the rendering loop you use when drawing a model and therefore is called the **picking loop.** (A picking loop is a type of submitting loop.) In a picking loop, however, instead of drawing the model, you pick the model by calling routines such as `Q3DisplayGroup_Submit`. See Listing 15-1 (page 956) for code that illustrates a picking loop.

Once you've completely specified the model within a picking loop, QuickDraw 3D can return to your application a list of all objects in the model that are close to the pick geometry. This list is the **hit list.** You can search through the returned hit list for individual items and obtain information about those items. You can also specify an order in which you want the items in the hit list to be sorted, and you can indicate in advance the kinds of objects you want QuickDraw 3D to put into the hit list. For example, you can indicate that you want QuickDraw 3D to put only entire objects into the hit list or that you want QuickDraw 3D to put only *parts of* objects (that is, its component vertices, edges, or faces) into the hit list.

Types of Pick Objects

A pick object is of type `TQ3PickObject`, which is one of the basic types of QuickDraw 3D objects. QuickDraw 3D defines several subtypes of pick objects, which are distinguished from one another by the pick geometry.

QuickDraw 3D provides two types of screen-space pick objects: **window-point pick objects** and **window-rectangle pick objects.** These pick objects test for closeness between the pick geometry (a point or rectangle in a window) and the screen projections of the objects in the model. In general, you'll use one of these

two screen-space pick objects when using picking as the basis of user interaction.

Note
There are many optimizations that can be used to determine whether an object in a model is suitably close to a pick geometry without having to perform all the projections that otherwise would be required. QuickDraw 3D uses these optimizations whenever appropriate. ◆

Hit Identification

Once you have created a pick object and specified the model within a picking loop, QuickDraw 3D determines which, if any, of the objects in the model are suitably close to the pick geometry specified in the pick object. QuickDraw 3D uses hit-tests that are appropriate to the specific pick object and the objects in the model being tested. For example, if you're using a window-point pick object and your model contains a triangle, QuickDraw 3D tests whether the pick geometry—a point—is inside the two-dimensional screen projection of the triangle. If it is, QuickDraw 3D adds the triangle to the hit list.

For the window point pick geometry, QuickDraw 3D allows you to specify two tolerance values, which indicate how close a pick geometry must be to an object in a model for a hit to occur. A pick object's **vertex tolerance** indicates how close two points must be for a hit to occur. A pick object's **edge tolerance** indicates how close a point must be to a line for a hit to occur. Edge and vertex tolerances apply to mesh shape parts; edge tolerances apply to lines and polylines; and point tolerances apply to vertices.

Table 15-1 lists the hit-tests that QuickDraw 3D uses for window-space pick objects. The tolerances for these picks are floating-point values that specify

units in the window coordinate system. QuickDraw 3D adds an object in a view to the hit list if the specified condition is fulfilled.

Table 15-1 Hit-tests for window-space pick objects

Object	Point pick objects	Rectangle pick objects
Marker	The pick point is inside the marker bitmap and on an active pixel. (No tolerance is used.)	The pick rectangle intersects the marker bitmap and covers an active pixel in the bitmap.
Point	The distance from the pick point to the screen projection of the point is less than or equal to the vertex tolerance.	The screen projection of the point is within the pick rectangle.
Line	The distance from the pick point to the closest point on the screen projection of the line is less than or equal to the edge tolerance.	The screen projection of the line intersects the pick rectangle.
Triangle	The pick point is inside of the screen projection of the triangle.	The screen projection of the triangle intersects the pick rectangle or lies completely within it.
Polygon	The pick point is inside of the screen projection of the polygon.	The screen projection of the polygon intersects the pick rectangle or lies completely within it.
Mesh	For object picking, the pick point is inside of the screen projection of any element of the mesh. For mesh vertex, edge, or face picking, the criteria for points, line, and triangles apply, respectively.	For object picking, the screen projection of any element of the mesh intersects the pick rectangle or lies completely within it. For mesh vertex, edge, or face picking, the criteria for points, line, and triangles apply, respectively

IMPORTANT

If the view within which picking is occurring is associated with a pixmap draw context, you need to transform the window-space pick coordinates (usually obtained from the mouse coordinates) to the pixmap's coordinate space. You can use original QuickDraw's `MapPt` function to do this. ▲

Hit Sorting

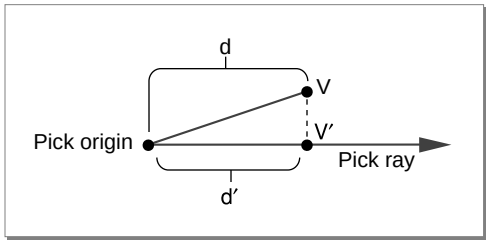
In some cases, you can have QuickDraw 3D sort a hit list before returning it to your application. The sorting is based on either increasing or decreasing distance from some point, the **pick origin.** As a result, hit-list sorting is possible only when the pick geometry has a clearly defined pick origin. Pick objects whose pick geometries have a pick origin are called **metric pick objects** (or **metric picks**). Window-point picking uses metric pick objects. With window-rectangle pick objects, however, there is no clearly defined pick origin. As a result, window-rectangle pick objects are not metric: you cannot have the hit list sorted by distance.

With a metric pick, distances are measured along the ray from the pick origin to the point of intersection on the picked object. If that ray intersects a picked object more than once, QuickDraw 3D always returns the hit that's closest to the pick origin.

Recall that you can have QuickDraw 3D put either entire objects or parts of objects into a hit list. When you are hit-testing parts of objects—vertices, edges, and faces—you need to keep in mind that the tolerance values can complicate the process of calculating distances (and hence the process of sorting hits). For example, a window point might be equally distant from both a vertex and an edge, at least within the tolerance values associated with the window-point pick object. To establish a unique sorting order in such cases, QuickDraw 3D gives priority to vertices, then to edges, and finally to faces.

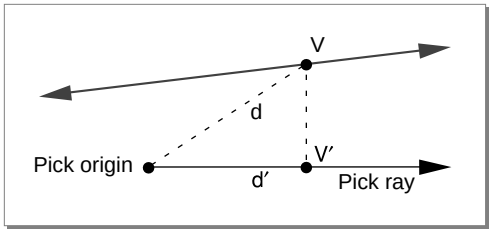
Note that the distances used to establish a sort order might not be the same distances reported to your application when you retrieve hit information. Consider, for example, the situation illustrated in Figure 15-1. Here, the vertex V is within the current vertex tolerance of the window point pick object and therefore qualifies as a hit. QuickDraw 3D uses the distance d' from the pick origin to the closest point on the pick ray (that is, V') as the basis for sorting vertex V in the hit list. However, when reporting the distance from the pick origin to the picked vertex V, QuickDraw 3D gives the actual distance d.

Figure 15-1 Determining a vertex sorting distance



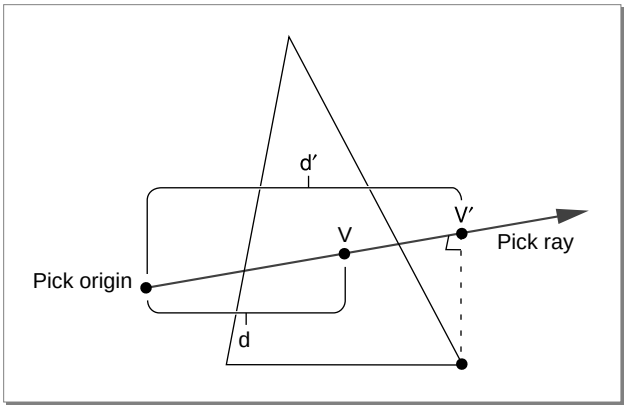
QuickDraw 3D calculates distances to edges and faces in an analogous manner. If the pick ray passes within the current edge tolerance of an edge, the sorting distance is set to the distance *d′* from the pick ray origin to the projection onto the pick ray of the point on the edge that is closest to the pick ray. See Figure 15-2.

Figure 15-2 Determining an edge sorting distance



If the pick ray intersects a face, the sorting distance is set to the distance from the pick ray origin to the projection onto the pick ray of the face vertex that is closest to the pick ray. See Figure 15-3.

Figure 15-3 Determining a face sorting distance



Note
The sorting distance *d′* is not always less than the actual distance *d* to the hit object. In Figure 15-3, for example, *d′* is greater than *d*. ◆

Hit Information

When you create a pick object, you specify (in the `mask` field of a pick data structure) a **hit information mask** value that indicates the kind of information you want returned about objects in the model. For example, you could use this code to request information about surface normals and the distance from the pick origin:

```
TQ3PickData             myPickData;
myPickData.mask = kQ3PickDetailMaskNormal |
                        kQ3PickDetailMaskDistance;
```

Once you've created the hit list, you can obtain information about a particular hit in the list by calling the `Q3Pick_GetPickDetailData` function. You pass this function a pick object, an index to a hit within the hit list, and the desired `pickDetailValue` from one of the bit values defined by the `TQ3PickDetailMasks` data type:

```
typedef enum TQ3PickDetailMasks {
    kQ3PickDetail_None                          = 0,
    kQ3PickDetailMask_PickID                    = 1 << 0,
    kQ3PickDetailMask_Path                      = 1 << 1,
    kQ3PickDetailMask_Object                    = 1 << 2,
    kQ3PickDetailMask_LocalToWorldMatrix        = 1 << 3,
    kQ3PickDetailMask_XYZ                       = 1 << 4,
    kQ3PickDetailMask_Distance                  = 1 << 5,
    kQ3PickDetailMask_Normal                    = 1 << 6,
    kQ3PickDetailMask_ShapePart                 = 1 << 7,
    kQ3PickDetailMask_PickPart                  = 1 << 8,
    kQ3PickDetailMask_UV                        = 1 << 9,
} TQ3PickDetailMasks;

typedef unsigned long       TQ3PickDetail;
```

QuickDraw 3D returns the specified pick detail data. Before using this information you should call `Q3Pick_GetPickDetailValidMask` to see what information QuickDraw 3D has returned. The values in the `mask` field of an initial pick data structure and the `validMask` value returned by this call can differ.

You need to pay attention to what information is returned in part because some kinds of information are not available for some combinations of pick object types and picked object types. For example, you cannot get information about a surface normal for a hit on a point (because points do not have normals). Similarly, you cannot get a distance value for a window-rectangle pick object (because rectangles have no origin from which to measure). Table 15-2 indicates the kinds of information you can receive about each type of picked object.

IMPORTANT
QuickDraw 3D can always return information for the `pickID`, `path`, `object`, and `localToWorldMatrix` data types. As a result, those fields are omitted from Table 15-2. ▲

Table 15-2 Pick geometries and information types supported by view objects

View object	`xyzPoint`	`distance`	`normal`	`shapePart`
Marker				
Point	Point Rectangle	Point		
Line	Point	Point		
Triangle	Point	Point	Point	
Polygon	Point	Point	Point	
Decomposition	Point	Point	Point	
Mesh	Point	Point	Point	Point

Using Pick Objects

A pick object contains all the information necessary to calculate geometric intersections between the pick geometry and the objects in a model. To create a pick object, you need to fill out data structures with the appropriate information, including

- how the hits are to be sorted
- how many hits to return
- what information should be returned about any hits
- whether to pick whole objects or parts of objects
- how much tolerance to allow when calculating hits
- the pick geometry

The following sections illustrate how to perform these tasks.

Handling Object Picking

Listing 15-1 illustrates how to create, use, and dispose of pick objects. It defines a function, `MyHandleClickInWindow`, that takes a window pointer and an event record and handles mouse clicks in that window.

Listing 15-1 Picking objects

```
TQ3Status MyHandleClickInWindow (CGrafPtr myWindow, EventRec myEvent)
{
    TQ3WindowPointPickData          myWPPickData;
    TQ3PickObject                   myPickObject;
    unsigned long                   myNumHits;
    unsigned long                   myIndex;
    Point                           myPoint;
    TQ3Point2D                      my2DPoint;
    TQ3ViewObject                   myView;

    /*Get the window coordinates of a mouse click.*/
    SetPort(myWindow);
    myPoint = myEvent.where;                /*get location of mouse click*/
    GlobalToLocal(&myPoint);                /*convert to window coordinates*/
    my2DPoint.x = myPoint.h;                /*configure a 2D point*/
    my2DPoint.y = myPoint.v;

    /*Set up picking data structures.*/
    /*Set sorting type: objects nearer to pick origin are returned first.*/
    myWPPickData.data.sort = kQ3PickSortNearToFar;
    myWPPickData.data.mask = kQ3PickDetailMaskPickID | kQ3PickDetailMaskXYZ |
                                                    kQ3PickDetailMaskObject;
    myWPPickData.data.numHitsToReturn = kQ3ReturnAllHits;
```

```
myWPPickData.point = my2DPoint;
myWPPickData.vertexTolerance = 2.0;
myWPPickData.edgeTolerance = 2.0;

/*Create a new window-point pick object.*/
myPickObject = Q3WindowPointPick_New(&myWPPickData);

myView = MyGetViewFromWindow(myWindow);         /*increments reference count*/

/*Pick a group object.*/
Q3View_StartPicking(myView, myPickObject);
do {
    Q3DisplayGroup_Submit(gGroup, myView);
} while (Q3View_EndPicking(myView) == kQ3ViewStatusRetraverse);

/*See whether any hits occurred.*/
if (Q3Pick_GetNumHits(myPickObject, &myNumHits) == kQ3Failure || myNumHits==0) {
    Q3Object_Dispose(myPickObject);
    return;
}

/* Process each hit */
for (myIndex = 0; myIndex = myNumHits; myIndex++) {
    TQ3Point3D      xyzPoint;
    unsigned long   pickID;
    TQ3Object       object;

    /* Get validMask first */
    if (Q3Pick_GetPickDetailValidMask(myPickObject, myIndex, &validMask)
                == kQ3Failure) {
            break;
    }

    if (! ((validMask & kQ3PickDetailMaskXYZ)     &&
               (validMask & kQ3PickDetailMaskPickID)  &&
               (validMask & kQ3PickDetailMaskObject))) {
            continue;
    }

    /* Get world space intersection, pick ID, and geometry object reference */
    object = NULL;
```

```
        status = Q3Pick_GetPickDetailData (myPickObject, myIndex,
                                           kQ3PickDetailMaskXYZ, &xyzPoint);
        status = Q3Pick_GetPickDetailData (myPickObject, myIndex,
                                           kQ3PickDetailMaskPickID, &pickID);
        status = Q3Pick_GetPickDetailData (myPickObject, myIndex,
                                           kQ3PickDetailMaskObject, &object);

        /* Operate on xyzPoint, pickID, and object */
        ...

        if (object != NULL) {
                Q3Object_Dispose(object);
        }
    }

    /*Dispose of all hits in the hit list.*/
    Q3Pick_EmptyHitList(myPickObject);

    /*Dispose of the pick object.*/
    Q3Object_Dispose(myPickObject);

    /*Dispose of the view object.*/
    Q3Object_Dispose(myView);
}
```

Note that the call to `Q3Pick_EmptyHitList` is redundant, because disposing of a pick object (by calling `Q3Object_Dispose`) also disposes of its associated hit list. The call is included in Listing 15-1 simply to illustrate how to call `Q3Pick_EmptyHitList`. You would, however, need to call to `Q3Pick_EmptyHitList` if you wanted to reuse the associated pick object in another pick operation.

Handling Mesh Part Picking

When a model includes a mesh, you can decide whether the entire mesh only or parts of the mesh also are eligible for picking. You do this by specifying an appropriate hit information mask. For example, to allow mesh parts to be selected, you can set up the hit information mask like this:

```
myPickData.mask = kQ3PickDetailMaskShapePart |
                  kQ3PickDetailMaskObject |
                  kQ3PickDetailMaskDistance;
```

This line of code indicates that you want QuickDraw 3D to return information about objects and any distinguishable parts of objects, as well as the distances from the objects to the pick origin. (To prevent mesh parts from being selected, you simply omit adding in the `kQ3PickDetailMaskShapePart` mask.)

You can determine whether data returned by `Q3Pick_GetPickDetailData` applies to a shape part by inspecting the `validMask` bit. If the value of the bit is 1, the data contains information about a shape part. Currently the only available shape parts are mesh parts. Listing 15-2 illustrates how to use the `shapePart` field to determine the type of mesh part selected and to perform some operation on the selected mesh part.

Listing 15-2 Picking mesh parts

```
Q3Pick_GetPickDetailValidMask(myPickObject, myIndex);
Q3Pick_GetPickDetailData(myPickObject, myIndex, shapePart);

if (shapePart != NULL) {
    switch(Q3Object_GetLeafType(shapePart)) {
        case kQ3MeshPartTypeMeshFacePart:
            Q3MeshFacePart_GetFace(shapePart, &myFace);
            MyDoPickFace(object, myFace);
            break;
        case kQ3MeshPartTypeMeshEdgePart:
            Q3MeshEdgePart_GetEdge(shapePart, &myEdge);
            MyDoPickEdge(object, myEdge);
            break;
        case kQ3MeshPartTypeMeshVertexPart:
            Q3MeshVertexPart_GetVertex(shapePart, &myVertex);
            MyDoPickVertex(object, myVertex);
            break;
    }
}
```

This code branches on the type of the mesh part indicated by the `shapePart` field. For each defined type of mesh part, the code calls a QuickDraw 3D routine to retrieve the corresponding mesh face, edge, or vertex. Then it calls an application-defined routine (for example, `MyDoPickFace`) to handle the mesh part selection.

Picking in Immediate Mode

Picking IDs are particularly useful when picking in immediate mode. Listing 15-3 shows how to create a triangle, attach a picking ID to it, and then process hits.

Listing 15-3 Picking in immediate mode

```
void MyImmediateModePickID (TQ3ViewObject view, WindowPtr window)
{
    TQ3WindowRectPickData               myPickData;
    TQ3TriangleData                     myTriangleData;
    TQ3PickObject                       myPick;
    TQ3ViewStatus                       myViewStatus;
    unsigned long                       pickID;
    Rect                                myPortRect;
    Point                               myCenter;
    unsigned long                       myNumHits;

    /*Set up a triangle.*/
    Q3Point3D_Set(&myTriangleData.vertices[0].point, -1.0, -0.5, 0.0);
    Q3Point3D_Set(&myTriangleData.vertices[1].point,  1.0,  0.0, 0.0);
    Q3Point3D_Set(&myTriangleData.vertices[2].point, -0.5,  1.5, 0.0);
    myTriangleData.vertices[0].attributeSet = NULL;
    myTriangleData.vertices[1].attributeSet = NULL;
    myTriangleData.vertices[2].attributeSet = NULL;
    myTriangleData.triangleAttributeSet = NULL;

    /*Set up TQ3WindowPointPickData structure.*/
    myPickData.data.sort = kQ3PickSortNone;
    myPickData.data.mask = kQ3PickDetailMaskPickID | kQ3PickDetailMaskObject;
    myPickData.data.numHitsToReturn = kQ3ReturnAllHits;

    myPortRect = ((GrafPtr) window)->myPortRect;
    myCenter.h = (myPortRect.right  - myPortRect.left)/2.0;
    myCenter.v = (myPortRect.bottom - myPortRect.top) /2.0;

    Q3Point2D_Set(&myPickData.rect.min, myCenter.h - 5, myCenter.v - 5);
    Q3Point2D_Set(&myPickData.rect.max, myCenter.h + 5, myCenter.v + 5);
```

```
    /*Create the window rectangle window pick.*/
    myPick = Q3WindowRectPick_New(&myPickData);

    /*Submit the pick ID and triangle in immediate mode.*/
    Q3View_StartPicking(view, myPick);
    do
    {
        Q3PickIDStyle_Submit(kPickID, view);
        Q3Triangle_Submit(&myTriangleData, view);
        myViewStatus = Q3View_EndPicking(view);
    } while (myViewStatus == kQ3ViewStatusRetraverse);

    Q3Pick_GetNumHits(myPick, &myNumHits);
    if (numHits == 1)
    {
        /*Get the pickID data (for first and only hit) and check if it's
            the expected pick ID.*/
        Q3Pick_GetPickDetailData(myPick, 0);
        if (pickID == kPickID)
        {
            /*picked on triangle with pick ID*/
        }
    }

    Q3Object_Dispose(myPick);
}
```

Pick Objects Reference

This section describes the constants, data structures, and routines provided by QuickDraw 3D that you can use to manage pick objects.

Constants

QuickDraw 3D provides constants that you can use to specify how to sort hit lists, what kinds of information you want returned about the items in a hit list, and what features of an object you want information about.

Hit List Sorting Values

You specify a **hit list sorting value** to determine the kind of sorting (relative to the pick origin) that is to be done on the hit list.

```
typedef enum TQ3PickSort {
    kQ3PickSortNone,
    kQ3PickSortNearToFar,
    kQ3PickSortFarToNear
} TQ3PickSort;
```

Constant descriptions

`kQ3PickSortNone` No sorting is to be done on the hit list. There is no meaning to the order of hits in the list.

`kQ3PickSortNearToFar`
The hit list is sorted according to increasing distance from the origin of the pick point. Objects nearer to the origin are returned before objects farther away.

`kQ3PickSortFarToNear`
The hit list is sorted according to decreasing distance from the origin of the pick point. Objects farther away from the origin are returned before objects nearer to it.

Hit Information Masks

The `Q3Pick_GetPickDetailValidMask` function returns a mask for all types of `TQ3PickDetail` information that is relevant to a hit with the specified index for the given pick object. The `Q3Pick_GetPickDetailValidMask` call should be followed with a call to `Q3Pick_GetPickDetailData` for each corresponding type of `pickDetail` value set in the returned `pickDetailValidMask`. If a bit in `pickDetailValidMask` is 0, it means that either the pick detail type wasn't specified when the pick was created, or if it was specified then it was meaningless for the type of pick object or the geometry intersected.

The hit detail masks are values of type `TQ3PickDetailMasks`. See "Hit Detail Data" (page 967) for a more complete description of the information these masks specify.

```
typedef enum TQ3PickDetailMasks {
    kQ3PickDetailNone                                   = 0,
    kQ3PickDetailMaskPickID                             = 1 << 0,
```

```
    kQ3PickDetailMaskPath                                   = 1 << 1,
    kQ3PickDetailMaskObject                                 = 1 << 2,
    kQ3PickDetailMaskLocalToWorldMatrix                     = 1 << 3,
    kQ3PickDetailMaskXYZ                                    = 1 << 4,
    kQ3PickDetailMaskDistance                               = 1 << 5,
    kQ3PickDetailMaskNormal                                 = 1 << 6,
    kQ3PickDetailMaskShapePart                              = 1 << 7,
    kQ3PickDetailMaskPart                                   = 1 << 8,
    kQ3PickDetailMaskUV                                     = 1 << 9,
    } TQ3PickDetailMasks;
```

Constant descriptions

`kQ3PickDetailNone`
: No pick detail. This mask results in faster picking, because various calculations do not need to be performed.

`kQ3PickDetailMaskPickID`
: The picking ID of the picked object.

`kQ3PickDetailMaskPath`
: The path through the model's group hierarchy to the picked object.

`kQ3PickDetailMaskObject`
: A reference to the object handle of the picked object.

`kQ3PickDetailMaskLocalToWorldMatrix`
: The matrix that transforms the local coordinate system of the picked object to the world coordinate system. Note that the local-to-world transform matrix for a multiply-referenced object differs for each reference to the object.

`kQ3PickDetailMaskXYZ`
: The point of intersection between the picked object and the pick geometry in world space.

`kQ3PickDetailMaskDistance`
: The distance between the intersected geometry and the origin of the pick geometry.

`kQ3PickDetailMaskNormal`
: The surface normal of the picked object at the point of intersection with the pick geometry. The magnitude of this normal should always be returned as a normalized vector.

`kQ3PickDetailMaskShapePart`
The reference to the shape part object of the picked object.

`kQ3PickDetailMaskPart`
The object, edge, or vertex.

`kQ3PickDetailMaskUV`
The surface parameterization of the picked object.

Pick Parts Masks

QuickDraw 3D defines **pick parts masks** to indicate the kinds of objects it has placed in the hit list. You use the face, vertex, and edge values to pick parts of meshes. To pick any other object, use the value `kQ3PickPartsObject`.

```
typedef enum TQ3PickPartsMasks {
    kQ3PickPartsObject                                  = 0,
    kQ3PickPartsMaskFace                                = 1 << 0,
    kQ3PickPartsMaskEdge                                = 1 << 1,
    kQ3PickPartsMaskVertex                              = 1 << 2
} TQ3PickPartsMasks;

typedef unsigned long       TQ3PickParts;
```

Constant descriptions

`kQ3PickPartsObject`
The hit list contains only whole objects.

`kQ3PickPartsMaskFace`
The hit list contains faces.

`kQ3PickPartsMaskEdge`
The hit list contains edges.

`kQ3PickPartsMaskVertex`
The hit list contains vertices.

Data Structures

This section describes the data structures you need to use for creating pick objects and retrieving the information returned in a hit list.

Pick Data Structure

You use a **pick data structure** to specify information when creating a pick object for subsequent picking. The pick data structure, common to all pick types, is defined by the `TQ3PickData` data type.

```
typedef struct TQ3PickData {
    TQ3PickSort                         sort;
    TQ3PickDetail                       mask;
    unsigned long                       numHitsToReturn;
} TQ3PickData;
```

Field descriptions

`sort` A hit list sorting value that determines the kind of sorting, if any, that is to be done on the hit list.

`mask` A hit information mask that determines the type of pick detail information to be returned for the items in a hit list.

`numHitsToReturn` The maximum number of hits to return. QuickDraw 3D discards any hits that would exceed this limit, but only *after* all possible hits have been found and placed into the sort order determined by the `sort` field. You can specify the constant `kQ3ReturnAllHits` to request that all hits be returned.

Window-Point Pick Data Structure

You use a **window-point pick data structure** to specify information when creating a pick object for subsequent window-point picking. A window-point pick data structure is defined by the `TQ3WindowPointPickData` data type.

```
typedef struct TQ3WindowPointPickData {
    TQ3PickData                         data;
    TQ3Point2D                          point;
    float                               vertexTolerance;
    float                               edgeTolerance;
} TQ3WindowPointPickData;
```

Field descriptions

`data` A pick data structure specifying basic information about the window-point pick object.

`point`	A point, in local window coordinates, where each unit equals 1/72 inch.
`vertexTolerance`	The vertex tolerance.
`edgeTolerance`	The edge tolerance.

Vertex and edge tolerances are discussed in "Hit Identification," beginning on page 949.

Window-Rectangle Pick Data Structure

You use a **window-rectangle pick data structure** to specify information when creating a pick object for subsequent window-rectangle picking. A window-rectangle pick data structure is defined by the `TQ3WindowRectPickData` data type.

```
typedef struct TQ3WindowRectPickData {
    TQ3PickData                 data;
    TQ3Area                     rect;
} TQ3WindowRectPickData;
```

Field descriptions

`data`	A pick data structure specifying basic information about the window-rectangle pick object.
`rect`	A rectangle, in local window coordinates, where each unit equals 1/72 inch.

Hit Path Structure

You use a **hit path structure** to get group information about the path through a model hierarchy to a specific picked object. A hit path structure is defined by the `TQ3HitPath` data type.

```
typedef struct TQ3HitPath {
    TQ3GroupObject              rootGroup;
    unsigned long               depth;
    TQ3GroupPosition            *positions;
} TQ3HitPath;
```

Field descriptions

`rootGroup`	The root group that was picked.

`depth`	The number of positions in the path. If the picked object is not in the model hierarchy, this field contains the value 0.
`positions`	A pointer to an array of group positions. This array is allocated by QuickDraw 3D.

Hit Detail Data

As described in "Hit Information Masks," beginning on page 962, `Q3Pick_GetPickDetailData` returns data for specific pick detail masks. The values of the returned data for each mask are shown in Table 15-3.

Table 15-3 Pick detail return data

Mask	Meaning
`PickID`	The style pick ID in the group of the picked object. The picking ID is a 32-bit value specified by your application. See the chapter "Style Objects" for more information about picking IDs. Picking IDs are especially useful for immediate mode picking. See Listing 15-3 (page 960) for a sample routine that uses picking IDs.
`Path`	The path through the model hierarchy to the picked object, from the root group of the hierarchy to the leaf object. See "Hit Path Structure" (page 966) for information about a path. For immediate mode picking, this field is not valid.
`Object`	A reference to the picked geometry object. For immediate mode picking, this field is not valid.
`LocalToWorldMatrix`	The matrix that transforms the local coordinates of the picked object to world-space coordinates. This matrix is copied from the graphics state in effect at the time the object is hit. If there are multiple references to an object, this matrix may be different for each individual reference.
`XYZ`	For window-point picking, the point (in world-space coordinates) at which the picked object and the pick geometry intersect. For all other types of picking, this field is undefined.

Table 15-3 Pick detail return data (continued)

Mask	Meaning
`Distance`	For window-point picking, the distance (in world space) from the origin of the picking ray to the point of intersection with the picked object. (This is effectively the distance from the camera to the intersection point, in world space.) For all other types of picking, this field is undefined.
`Normal`	The surface normal of the picked object at the point of intersection with the pick geometry. This field is valid only for window-point picking.
`ShapePart`	The shape part object, if any, that was picked. If the picked object has no distinguishable shape parts, this field contains the value `NULL`. If the value of this field is not `NULL`, you can call the `Q3ShapePart_GetType` function to get the type of this shape part object, or `Q3Object_GetLeafType` to get the leaf type of this shape part.
`Part`	The object, edge, or vertex.
`UV`	The surface parameterization of the picked object.

Pick Objects Routines

This section describes the routines you can use to manage pick objects and hit lists.

Managing Pick Objects

QuickDraw 3D provides a number of general routines for managing pick objects of any kind.

Q3Pick_GetType

You can use the `Q3Pick_GetType` function to get the type of a pick object.

```
TQ3ObjectType Q3Pick_GetType (TQ3PickObject pick);
```

`pick` A pick object.

DESCRIPTION

The `Q3Pick_GetType` function returns, as its function result, the type of the pick object specified by the `pick` parameter. The types of pick objects currently supported by QuickDraw 3D are defined by these constants:

```
kQ3PickTypeWindowPoint
kQ3PickTypeWindowRect
```

If the specified pick object is invalid or is not one of these types, `Q3Pick_GetType` returns the value `kQ3ObjectTypeInvalid`.

Q3Pick_GetData

You can use the `Q3Pick_GetData` function to get the basic data associated with a pick object.

```
TQ3Status Q3Pick_GetData (
                    TQ3PickObject pick,
                    TQ3PickData *data);
```

`pick` A pick object.

`data` On entry, a pointer to a pick data structure.

DESCRIPTION

The `Q3Pick_GetData` function returns, through the `data` parameter, basic information about the pick object specified by the `pick` parameter. See "Pick Data Structure" (page 965) for a description of a pick data structure. Your application is responsible for allocating memory for the pick data structure before calling `Q3Pick_GetData` and for disposing of that memory when you're finished using that structure.

Q3Pick_SetData

You can use the `Q3Pick_SetData` function to set the basic data associated with a pick object.

```
TQ3Status Q3Pick_SetData (
                    TQ3PickObject pick,
                    const TQ3PickData *data);
```

`pick` A pick object.

`data` A pointer to a pick data structure.

DESCRIPTION

The `Q3Pick_SetData` function sets the data associated with the pick object specified by the `pick` parameter to the data specified by the `data` parameter.

Q3Pick_GetVertexTolerance

You can use the `Q3Pick_GetVertexTolerance` function to get the current vertex tolerance of a pick object.

```
TQ3Status Q3Pick_GetVertexTolerance (
                    TQ3PickObject pick,
                    float *vertexTolerance);
```

`pick` A pick object.

`vertexTolerance`
On exit, the current vertex tolerance of the specified pick object.

DESCRIPTION

The `Q3Pick_GetVertexTolerance` function returns, in the `vertexTolerance` parameter, the current vertex tolerance of the pick object specified by the `pick` parameter. If the specified pick object does not support a vertex tolerance, `Q3Pick_GetVertexTolerance` generates an error. Tolerances are discussed in "Hit Identification," beginning on page 949.

Q3Pick_SetVertexTolerance

You can use the `Q3Pick_SetVertexTolerance` function to set the vertex tolerance of a pick object.

```
TQ3Status Q3Pick_SetVertexTolerance (
                    TQ3PickObject pick,
                    float vertexTolerance);
```

`pick` A pick object.

`vertexTolerance`
The desired vertex tolerance of the specified pick object.

DESCRIPTION

The `Q3Pick_SetVertexTolerance` function sets the vertex tolerance of the pick object specified by the `pick` parameter to the tolerance specified by the `vertexTolerance` parameter. If the specified pick object does not support a vertex tolerance, `Q3Pick_SetVertexTolerance` generates an error. Tolerances are discussed in "Hit Identification," beginning on page 949.

Q3Pick_GetEdgeTolerance

You can use the `Q3Pick_GetEdgeTolerance` function to get the current edge tolerance of a pick object.

```
TQ3Status Q3Pick_GetEdgeTolerance (
                    TQ3PickObject pick,
                    float *edgeTolerance);
```

`pick` A pick object.

`edgeTolerance`
On exit, the current edge tolerance of the specified pick object.

DESCRIPTION

The `Q3Pick_GetEdgeTolerance` function returns, in the `edgeTolerance` parameter, the current edge tolerance of the pick object specified by the `pick` parameter. If the specified pick object does not support an edge tolerance, `Q3Pick_GetEdgeTolerance` generates an error. Tolerances are discussed in "Hit Identification," beginning on page 949.

Q3Pick_SetEdgeTolerance

You can use the `Q3Pick_SetEdgeTolerance` function to set the edge tolerance of a pick object.

```
TQ3Status Q3Pick_SetEdgeTolerance (
                    TQ3PickObject pick,
                    float edgeTolerance);
```

`pick` A pick object.

`edgeTolerance`
The desired edge tolerance of the specified pick object.

DESCRIPTION

The `Q3Pick_SetEdgeTolerance` function sets the edge tolerance of the pick object specified by the `pick` parameter to the tolerance specified by the `edgeTolerance` parameter. If the specified pick object does not support an edge tolerance, `Q3Pick_SetEdgeTolerance` generates an error. Tolerances are discussed in "Hit Identification," beginning on page 949.

Q3Pick_GetPickDetailValidMask

You can use the `Q3Pick_GetPickDetailValidMask` function to return a mask designating the available `TQ3PickDetail` information. It should be followed with a call to `Q3Pick_GetPickDetailData` for each `pickDetail` bit set in the returned mask.

```
TQ3Status Q3Pick_GetPickDetailValidMask (
                    TQ3PickObject pick,
                    unsigned long index,
                    TQ3PickDetail *pickDetailValidMask);
```

`pick` A pick object.

`index` An index into a hit list.

`pickDetailValidMask`
The detail mask. For mask values, see "Hit Information," beginning on page 953.

DESCRIPTION

`Q3Pick_GetPickDetailValidMask` returns, in the `pickDetailValidMask` parameter, a mask designating available pick detail data for the pick object `pick` and hit index `index`. If a bit in `pickDetailValidMask` is 0, it means that either the pick detail type wasn't specified when the pick was created, or if it was specified then it was meaningless for the type of pick object or the geometry intersected.

The `index` parameter is a zero-based value within the maximum number of hits obtained by `callingQ3Pick_GetNumHits`.

Q3Pick_GetPickDetailData

You can use the `Q3Pick_GetPickDetailData` function to obtain pick detail data for a given bit out of the mask returned by `Q3Pick_GetPickDetailValidMask`.

```
TQ3Status Q3Pick_GetPickDetailData (
                    TQ3PickObject pick,
                    unsigned long index,
                    TQ3PickDetail pickDetailValue,
                    void *detailData);
```

`pick` A pick object.

`index` An index into a hit list.

`pickDetailValue`
A single-bit mask.

`detailData` The returned detail pick data.

DESCRIPTION

The `Q3Pick_GetPickDetailData` function returns, in the `detailData` parameter, the data corresponding to a `pickDetailValue` mask for the pick object `pick` and hit index `index`. See "Hit Detail Data" (page 967) for descriptions of the information retrurned.

The value of `pickDetailValue` can be only one bit from the set of possible `TQ3PickDetailMasks` values returned by `Q3Pick_GetPickDetailValidMask`. `TQ3PickDetailMasks` values may not be combined using `OR`.

Upon successful completion, `Q3Pick_GetPickDetailData` returns a function result of `kQ3Success`.

Q3HitPath_EmptyData

You must use the `Q3HitPath_EmptyData` function to dispose of the path data that QuickDraw 3D allocated internally as the result of a previous call to `Q3Pick_GetPickDetailData`.

```
TQ3Status Q3HitPath_EmptyData (TQ3HitPath *hitPath);
```

`hitPath` A hit path.

DESCRIPTION

The `Q3HitPath_EmptyData` function disposes of path data allocated internally as the result of a previous call to `Q3Pick_GetPickDetailData`. It returns `kQ3Success` if it completes successfully.

It is the responsibility of the application to dispose of the path data using `Q3HitPath_EmptyData` and the `object` and `shapePart` references using `Q3Object_Dispose`; otherwise undisposed memory or objects will be leaked.

Q3Pick_GetNumHits

You can use the `Q3Pick_GetNumHits` function to get the number of hits in the hit list of a pick object.

```
TQ3Status Q3Pick_GetNumHits (
                    TQ3PickObject pick,
                    unsigned long *numHits);
```

`pick` A pick object.

`numHits` On exit, the number of items in the hit list of the specified pick object.

DESCRIPTION

The `Q3Pick_GetNumHits` function returns, in the `numHits` parameter, the number of items in the hit list associated with the pick object specified by the `pick` parameter. This number never exceeds the maximum number of items specified in the pick object's data structure.

Q3Pick_EmptyHitList

You can use the `Q3Pick_EmptyHitList` function to empty a pick object's hit list.

```
TQ3Status Q3Pick_EmptyHitList (TQ3PickObject pick);
```

`pick` A pick object.

DESCRIPTION

The `Q3Pick_EmptyHitList` function disposes of all QuickDraw 3D-allocated memory occupied by the hit list associated with the pick object specified by the `pick` parameter. (This memory is also disposed of when the specified pick object is disposed of.) `Q3Pick_EmptyHitList` also sets the hit count of the specified pick object to 0.

Managing Shape Parts and Mesh Parts

QuickDraw 3D provides routines that you can use to get shape parts and mesh parts and to determine the shape objects that correspond to those parts. They use these types:

```
typedef TQ3ShapePartObject      TQ3MeshPartObject;
typedef TQ3MeshPartObject       TQ3MeshFacePartObject;
typedef TQ3MeshPartObject       TQ3MeshEdgePartObject;
typedef TQ3MeshPartObject       TQ3MeshVertexPartObject;
```

Q3ShapePart_GetShape

You can use the `Q3ShapePart_GetShape` function to get the shape object that contains a shape part object.

```
TQ3Status Q3ShapePart_GetShape (
                    TQ3ShapePartObject shapePartObject,
                    TQ3ShapeObject *shapeObject);
```

`shapePartObject`
A shape part object.

`shapeObject` On exit, the shape object that contains the specified shape part object.

DESCRIPTION

The `Q3ShapePart_GetShape` function returns, in the `shapeObject` parameter, the shape object that contains the shape part object specified by the `shapePartObject` parameter.

Q3ShapePart_GetType

You can use the `Q3ShapePart_GetType` function to get the type of a shape part object.

```
TQ3ObjectType Q3ShapePart_GetType (TQ3ShapePartObject shapePartObject);
```

`shapePartObject`
A shape part object.

DESCRIPTION

The `Q3ShapePart_GetType` function returns, as its function result, the type identifier of the shape part object specified by the `shapePartObject` parameter. If successful, `Q3ShapePart_GetType` returns this constant:

`kQ3ShapePartTypeMeshPart`

If the type cannot be determined or is invalid, `Q3ShapePart_GetType` returns the value `kQ3ObjectTypeInvalid`.

Q3MeshPart_GetType

You can use the `Q3MeshPart_GetType` function to get the type of a mesh part object.

```
TQ3ObjectType Q3MeshPart_GetType (TQ3MeshPartObject meshPartObject);
```

`meshPartObject`
A mesh part object.

DESCRIPTION

The `Q3MeshPart_GetType` function returns, as its function result, the type identifier of the mesh part object specified by the `meshPartObject` parameter. If successful, `Q3MeshPart_GetType` returns one of these constants:

```
kQ3MeshPartTypeMeshFacePart
kQ3MeshPartTypeMeshEdgePart
kQ3MeshPartTypeMeshVertexPart
```

If the type cannot be determined or is invalid, `Q3MeshPart_GetType` returns the value `kQ3ObjectTypeInvalid`.

Q3MeshPart_GetComponent

You can use the `Q3MeshPart_GetComponent` function to get the mesh component that contains a mesh part.

```
TQ3Status Q3MeshPart_GetComponent (
                    TQ3MeshPartObject meshPartObject,
                    TQ3MeshComponent *component);
```

`meshPartObject`
A mesh part object.

`component` On exit, the mesh component that contains the specified mesh part object.

DESCRIPTION

The `Q3MeshPart_GetComponent` function returns, in the `component` parameter, the mesh component that contains the mesh part object specified by the `meshPartObject` parameter.

Q3MeshFacePart_GetFace

You can use the `Q3MeshFacePart_GetFace` function to get the mesh face that corresponds to a mesh face part.

```
TQ3Status Q3MeshFacePart_GetFace (
                    TQ3MeshFacePartObject meshFacePartObject,
                    TQ3MeshFace *face);
```

`meshFacePartObject`
A mesh face part object.

`face` On exit, the mesh face that corresponds to the specified mesh face part object.

DESCRIPTION

The `Q3MeshFacePart_GetFace` function returns, in the `face` parameter, the mesh face that corresponds to the mesh face part object specified by the `meshFacePartObject` parameter.

Q3MeshEdgePart_GetEdge

You can use the `Q3MeshEdgePart_GetEdge` function to get the mesh edge that corresponds to a mesh edge part.

```
TQ3Status Q3MeshEdgePart_GetEdge (
                    TQ3MeshEdgePartObject meshEdgePartObject,
                    TQ3MeshEdge *edge);
```

`meshEdgePartObject`
A mesh edge part object.

`edge` On exit, the mesh edge that corresponds to the specified mesh face part object.

DESCRIPTION

The `Q3MeshEdgePart_GetEdge` function returns, in the `edge` parameter, the mesh edge that corresponds to the mesh edge part object specified by the `meshEdgePartObject` parameter.

Q3MeshVertexPart_GetVertex

You can use the `Q3MeshVertexPart_GetVertex` function to get the mesh vertex that corresponds to a mesh vertex part.

```
TQ3Status Q3MeshVertexPart_GetVertex (
                    TQ3MeshVertexPartObject meshVertexPartObject,
                    TQ3MeshVertex *vertex);
```

`meshVertexPartObject`
A mesh vertex part object.

`vertex` On exit, the mesh vertex that corresponds to the specified mesh vertex part object.

DESCRIPTION

The `Q3MeshVertexPart_GetVertex` function returns, in the `vertex` parameter, the mesh vertex that corresponds to the mesh vertex part object specified by the `meshVertexPartObject` parameter.

Picking With Window Points

QuickDraw 3D provides routines that you can use to pick with window points. The location of the point is in the resolution of the current draw context.

Q3WindowPointPick_New

You can use the `Q3WindowPointPick_New` function to create a new window-point pick object.

```
TQ3PickObject Q3WindowPointPick_New (
                    const TQ3WindowPointPickData *data);
```

`data` A pointer to a window-point pick data structure.

DESCRIPTION

The `Q3WindowPointPick_New` function returns, as its function result, a new window-point pick object having the characteristics specified by the `data` parameter. If `Q3WindowPointPick_New` fails, it returns `NULL`.

Q3WindowPointPick_GetPoint

You can use the `Q3WindowPointPick_GetPoint` function to get the point of a window-point pick object.

```
TQ3Status Q3WindowPointPick_GetPoint (
                    TQ3PickObject pick,
                    TQ3Point2D *point);
```

`pick` A window-point pick object.

`point` On exit, the current point of the specified window-point pick object.

DESCRIPTION

The `Q3WindowPointPick_GetPoint` function returns, in the `point` parameter, the current point of the window-point pick object specified by the `pick` parameter.

Q3WindowPointPick_SetPoint

You can use the `Q3WindowPointPick_SetPoint` function to set the point of a window-point pick object in screen space.

```
TQ3Status Q3WindowPointPick_SetPoint (
                    TQ3PickObject pick,
                    const TQ3Point2D *point);
```

`pick` A window-point pick object.

`point` The desired point for the specified window-point pick object.

DESCRIPTION

The `Q3WindowPointPick_SetPoint` function sets the point of the window-point pick object specified by the `pick` parameter to the point specified by the `point` parameter.

Q3WindowPointPick_GetData

You can use the `Q3WindowPointPick_GetData` function to get the data associated with a window-point pick object.

```
TQ3Status Q3WindowPointPick_GetData (
                    TQ3PickObject pick,
                    TQ3WindowPointPickData *data);
```

`pick` A window-point pick object.

`data` On exit, a pointer to a window-point pick data structure.

DESCRIPTION

The `Q3WindowPointPick_GetData` function returns, through the `data` parameter, information about the window-point pick object specified by the `pick` parameter. See "Window-Point Pick Data Structure" (page 965) for a description of a window-point pick data structure.

Q3WindowPointPick_SetData

You can use the `Q3WindowPointPick_SetData` function to set the data associated with a window-point pick object.

```
TQ3Status Q3WindowPointPick_SetData (
                    TQ3PickObject pick,
                    const TQ3WindowPointPickData *data);
```

`pick` A window-point pick object.

`data` A pointer to a window-point pick data structure.

DESCRIPTION

The `Q3WindowPointPick_SetData` function sets the data associated with the window-point pick object specified by the `pick` parameter to the data specified by the `data` parameter.

Picking With Window Rectangles

QuickDraw 3D provides routines that you can use to pick with window rectangles. The dimensions of the rectangle are in the resolution of the current draw context.

Q3WindowRectPick_New

You can use the `Q3WindowRectPick_New` function to create a new window-rectangle pick object.

```
TQ3PickObject Q3WindowRectPick_New (
                    const TQ3WindowRectPickData *data);
```

`data` A pointer to a window-rectangle pick data structure.

DESCRIPTION

The `Q3WindowRectPick_New` function returns, as its function result, a new window-rectangle pick object having the characteristics specified by the `data` parameter. If `Q3WindowRectPick_New` fails, it returns `NULL`.

Q3WindowRectPick_GetRect

You can use the `Q3WindowRectPick_GetRect` function to get the rectangle of a window-rectangle pick object.

```
TQ3Status Q3WindowRectPick_GetRect (
                    TQ3PickObject pick,
                    TQ3Area *rect);
```

`pick` A window-rectangle pick object.

`rect` On exit, the current rectangle of the specified window-rectangle pick object.

DESCRIPTION

The `Q3WindowRectPick_GetRect` function returns, in the `rect` parameter, the current rectangle of the window-rectangle pick object specified by the `pick` parameter.

Q3WindowRectPick_SetRect

You can use the `Q3WindowRectPick_SetRect` function to set the rectangle of a window-rectangle pick object.

```
TQ3Status Q3WindowRectPick_SetRect (
                    TQ3PickObject pick,
                    const TQ3Area *rect);
```

`pick` A window-rectangle pick object.

`rect` The desired rectangle for the window-rectangle pick object.

DESCRIPTION

The `Q3WindowRectPick_SetRect` function sets the rectangle of the window-rectangle pick object specified by the `pick` parameter to the rectangle specified by the `rect` parameter.

Q3WindowRectPick_GetData

You can use the `Q3WindowRectPick_GetData` function to get the data associated with a window-rectangle pick object.

```
TQ3Status Q3WindowRectPick_GetData (
                    TQ3PickObject pick,
                    TQ3WindowRectPickData *data);
```

`pick` A window-rectangle pick object.

`data` On exit, a pointer to a window-rectangle pick data structure.

DESCRIPTION

The `Q3WindowRectPick_GetData` function returns, through the `data` parameter, information about the window-rectangle pick object specified by the `pick` parameter. See "Window-Rectangle Pick Data Structure" (page 966) for the structure of a window-rectangle pick data structure.

Q3WindowRectPick_SetData

You can use the `Q3WindowRectPick_SetData` function to set the data associated with a window-rectangle pick object.

```
TQ3Status Q3WindowRectPick_SetData (
                    TQ3PickObject pick,
                    const TQ3WindowRectPickData *data);
```

`pick` A window-rectangle pick object.

`data` A pointer to a window-rectangle pick data structure.

DESCRIPTION

The `Q3WindowRectPick_SetData` function sets the data associated with the window-rectangle pick object specified by the `pick` parameter to the data specified by the `data` parameter.

Picking Warnings

The `kQ3WarningPickParamOutside` warning may be returned by picking routines. A list of general QuickDraw 3D errors is given in "QuickDraw 3D Errors, Warnings, and Notices" (page 87).

CHAPTER 15

Pick Objects

CHAPTER 16

Storage Objects

This chapter describes storage objects and the functions you can use to manipulate them. You use storage objects to represent a piece of storage accessible in a computer (for example, a file on disk, a block of memory, or some data on the Clipboard). A storage object connects a physical storage device to a file object. You use storage objects together with file objects to access the data on that storage device.

To use this chapter, you should already be familiar with the QuickDraw 3D class hierarchy, described in the chapter "QuickDraw 3D Objects." For information about file objects, see the chapter "File Objects." You do not, however, need to know how to create file objects or attach them to storage objects to read this chapter.

This chapter begins by describing storage objects and their features. Then it shows how to create and manipulate storage objects. The section "Storage Objects Reference," beginning on page 992 provides a complete description of storage objects and the routines you can use to create and manipulate them.

About Storage Objects

A **storage object** is a type of QuickDraw 3D object that you can use to represent a physical piece of storage in a computer. The piece of storage can be any data that is accessible in a linear, stream-based manner. QuickDraw 3D currently supports three basic types of data storage formats: data stored in memory, data stored in the data fork of a Macintosh file, and data stored in files accessed through the C programming language standard I/O library. QuickDraw 3D represents these data storage devices as storage objects.

To read data from (or write data to) a data storage device, you first need to create a storage object of the appropriate type. For example, to read data from a

Macintosh file, you can create a Macintosh storage object. You also need to create a file object (of type `TQ3FileObject`) and attach the file object to the storage object. Once you've created a storage object and a file object and attached them to one another, you can then read data from the file object by using file object reading calls. See the chapter "File Objects" for information on creating file objects, attaching them to storage objects, and reading or writing data using those file objects.

QuickDraw 3D distinguishes between storage objects and file objects primarily so that you can read and write stored data using a single set of functions. QuickDraw 3D supports only one class of file object, instances of which can be attached to any of the types of storage objects that it supports.

A storage object is of type `TQ3StorageObject`, which is a type of shared object. QuickDraw 3D provides three subclasses of the `TQ3StorageObject` type:

- A **memory storage object** (of type `kQ3StorageTypeMemory`) represents a dynamically allocated block of RAM. You can allocate the block of memory yourself, or you can have QuickDraw 3D allocate a block of memory on your behalf. Memory storage objects are available on all computer systems. QuickDraw 3D supports one subclass of the `kQ3StorageTypeMemory` storage object type:
- A **handle storage object** (of type `kQ3MemoryStorageTypeHandle`) represents a handle to a block of dynamically allocated RAM. On the Macintosh Operating System, QuickDraw 3D uses the `SetHandleSize` function when it needs to change the size of the memory block. On operating systems that do not support handles, QuickDraw 3D allocates and maintains the memory blocks internally.
- A **Macintosh storage object** (of type `kQ3StorageTypeMacintosh`) represents the data fork of a Macintosh file using a file reference number. Macintosh storage objects are available only on the Macintosh Operating System. QuickDraw 3D supports one subclass of the `kQ3StorageTypeMacintosh` storage object type:
- A **Macintosh FSSpec storage object** of type `kQ3MacintoshStorageTypeFSSpec` represents the data fork of a Macintosh file using a file system specification structure (of type `FSSpec`). QuickDraw 3D uses the Alias Manager to create cross-file references.
- A **UNIX® storage object** (of type `kQ3StorageTypeUnix`) represents a file using a structure of type `FILE`. This structure is accessed using the **standard I/O library,** a collection of functions that provide character I/O and

file-manipulation services for C programs on any operating system. The represented object can be a pipe, the standard input file, the standard output file, or any other `FILE` abstraction. QuickDraw 3D supports one subclass of the `kQ3StorageTypeUnix` storage object type:

- A **UNIX path name storage object** (of type `kQ3UnixStorageTypePath`) represents a file using a path name.

IMPORTANT

UNIX storage objects and UNIX path name storage objects can be used to represent any object accessible through the standard I/O library on *any* operating system. The names, which can therefore be confusing, derive from the origin of the standard I/O library on the UNIX operating system. ▲

For a description of pointers and handles, see the book *Inside Macintosh: Memory.* For a description of the Macintosh file-specification methods (that is, file reference numbers and file system specification structures), see the book *Inside Macintosh: Files.* For a description of the standard I/O library, see the documentation for any UNIX-based computer (for example, *A/UX Essentials* from Apple Computer, Inc., or *The UNIX Programming Environment* by Kernighan and Pike), or any book devoted specifically to C language programming (for example, *The C Programming Language* by Kernighan and Ritchie).

Using Storage Objects

As indicated earlier, you use storage objects to represent physical storage devices available on a computer. Most often, you'll simply create a new storage object associated with some part of a storage device (for instance, with some file on a disk drive) and then attach that storage object to a file object (by calling the `Q3File_SetStorage` function). If necessary, you can also get or set some of the information associated with a particular storage object. For example, you can determine the file reference number of the open file associated with a Macintosh storage object. This section describes how to perform these two tasks.

Creating a Storage Object

Creating a storage object essentially involves indicating to QuickDraw 3D the location and possibly also the size of the piece of physical storage you later want to read data from or write data to. Once you've created a storage object, you attach it to a file object and perform all I/O operations using file object functions. Listing 16-1 illustrates how to create a storage object connected to an open Macintosh file.

Listing 16-1 Creating a Macintosh storage object

```
myErr = FSpOpenDF(&myFSSpec, fsCurPerm, &myFRefNum);
if (!myErr)
    myStorageObj = Q3MacintoshStorage_New(myFRefNum);
```

Listing 16-2 illustrates how to open a file and create a UNIX storage object connected to that open file.

Listing 16-2 Creating a UNIX storage object

```
myFile = fopen("..:teacup.eb", "r");
if (myFile)
    myStorageObj = Q3UnixStorage_New(myFile);
```

Listing 16-3 illustrates how to allocate a block of memory and create a storage object connected to that block.

Listing 16-3 Creating a memory storage object

```
#define kBufferSize                         256

myBuffer = malloc(kBufferSize);
if (myBuffer)
    myStorageObj = Q3MemoryStorage_NewBuffer(myBuffer, 0, kBufferSize);
```

In the code shown in Listing 16-1 through Listing 16-3, your application specifically reserves the desired piece of the physical storage device, either by

opening a file or by allocating memory. In these cases, your application must also make sure to close the file or deallocate the memory block after you've closed or disposed of the associated storage object.

Note, however, that QuickDraw 3D provides two types of memory storage functions. The function `Q3MemoryStorage_NewBuffer` creates a new memory storage object using a specified buffer. The function `Q3MemoryStorage_New` creates a new memory storage object but copies the data in the specified buffer into its own internal memory. If you create a storage object by calling `Q3MemoryStorage_New`, you can dispose of the buffer once `Q3MemoryStorage_New` returns.

IMPORTANT
Whenever you create a storage object associated with an open file or an allocated memory block, you must close the file or dispose of the memory. ▲

▲ **WARNING**
When you open a piece of storage (that is, a file or a block of memory), you must not access that piece of storage once you've created a storage object to represent it. QuickDraw 3D assumes that it has exclusive access to all data in any part of a physical storage device associated with an open storage object. ▲

Getting and Setting Storage Object Information

QuickDraw 3D provides routines that you can use to get or set some of the information it maintains about storage objects. For example, you can get the file reference number of the Macintosh file associated with a Macintosh storage object by calling the function `Q3MacintoshStorage_Get`. Similarly, you can determine the starting address and size of a buffer associated with a memory storage object by calling `Q3MemoryStorage_GetBuffer`.

In general, the routines that get and set storage object information operate like the get and set routines for other types of QuickDraw 3D objects, but with several important differences:

- For memory storage objects created by a call to `Q3MemoryStorage_NewBuffer`, the returned address is the address of the actual buffer associated with the storage object, *not* the address of a copy of that buffer. In addition, that buffer may change locations in memory (but only if QuickDraw 3D allocated the

buffer on your behalf and writing data to the storage object causes QuickDraw 3D to resize the buffer).

- You cannot access subclass data using the get and set methods of a class. For example, you cannot use `Q3MemoryStorage_Get` or `Q3MemoryStorage_Set` with a handle storage object (of type `kQ3MemoryStorageTypeHandle`). Similarly, you cannot use `Q3UnixStorage_Get` or `Q3UnixStorage_Set` with a UNIX path name storage object (of type `kQ3UnixStorageTypePath`).
- You cannot use the get or set methods with a storage object that is open. A storage object is considered **open** whenever its associated storage is in use—for example, when an application is reading data from a file object attached to the storage object. (To be more specific, a storage object is open if it has been attached to a file object by a call to the `Q3File_SetStorage` function and that file object has been opened by a call to the `Q3File_OpenRead` or `Q3File_OpenWrite` function.) A storage object is considered **closed** at all other times. (Note that a storage object can be closed even though the associated file on disk is open to the operating system.)

Storage Objects Reference

This section describes the routines you can use to create and manipulate storage objects.

Storage Objects Routines

This section describes routines you can use to manage storage objects.

Managing Storage Objects

QuickDraw 3D provides several general routines for getting the type and size of storage objects. It also provides routines you can use to get and set the private data of a storage object.

Q3Storage_GetType

You can use the `Q3Storage_GetType` function to get the type of a storage object.

```
TQ3ObjectType Q3Storage_GetType (TQ3StorageObject storage);
```

`storage` A storage object.

DESCRIPTION

The `Q3Storage_GetType` function returns, as its function result, the type of the storage object specified by the `storage` parameter. The types of storage objects currently supported by QuickDraw 3D are defined by these constants:

```
kQ3StorageTypeMemory
kQ3StorageTypeMacintosh
kQ3StorageTypeUnix
kQ3StorageTypeWin32
```

If the specified storage object is invalid or is not one of these types, `Q3Storage_GetType` returns the value `kQ3ObjectTypeInvalid`.

ERRORS

```
kQ3ErrorInvalidObjectParameter
kQ3ErrorNULLParameter
```

Q3Storage_GetSize

You can use the `Q3Storage_GetSize` function to get the size of the data stored in a storage object.

```
TQ3Status Q3Storage_GetSize (
                    TQ3StorageObject storage,
                    unsigned long *size);
```

`storage` A storage object.

`size` On entry, a pointer to a buffer. On exit, the number of bytes of data stored in the specified storage object.

DESCRIPTION

The `Q3Storage_GetSize` function returns, through the `size` parameter, the number of bytes of data stored in the storage object specified by the `storage` parameter. That storage object must already be open when you call `Q3Storage_GetSize`.

ERRORS

```
kQ3ErrorInvalidObjectParameter
kQ3ErrorNULLParameter
kQ3ErrorStorageNotOpen
```

Q3Storage_GetData

You can use the `Q3Storage_GetData` function to get the data stored in a storage object.

```
TQ3Status Q3Storage_GetData (
                    TQ3StorageObject storage,
                    unsigned long offset,
                    unsigned long dataSize,
                    unsigned char *data,
                    unsigned long *sizeRead);
```

`storage` A storage object.

`offset` An offset into the private data associated with the specified storage object.

`dataSize` The number of bytes of data from the specified storage object to be returned in the specified buffer.

`data` On entry, a pointer to a buffer that is at least large enough to contain the number of bytes of data specified by the `dataSize` parameter. On exit, this buffer is filled with data from the specified storage object.

sizeRead	On exit, the number of bytes of data read from the specified storage object.

DESCRIPTION

The Q3Storage_GetData function returns, through the data parameter, some or all of the private data associated with the storage object specified by the storage parameter. The data to be returned begins at an offset specified by the offset parameter and extends for dataSize bytes from that location. On exit, the sizeRead parameter contains the number of bytes actually retrieved from the storage object's private data into the data buffer. If the value returned in the sizeRead parameter is less than the number of bytes requested in the dataSize parameter, then the end of the storage object's private data occurs at the distance offset + sizeRead from the beginning of the private data.

If the specified storage object is associated with a file object, that file object must be closed before you call Q3Storage_GetData.

Q3Storage_SetData

You can use the Q3Storage_SetData function to set the data stored in a storage object.

```
TQ3Status Q3Storage_SetData (
                    TQ3StorageObject storage,
                    unsigned long offset,
                    unsigned long dataSize,
                    const unsigned char *data,
                    unsigned long *sizeWritten);
```

storage	A storage object.
offset	An offset into the specified storage object.
dataSize	The number of bytes of data from the specified buffer to be written to the specified storage object.
data	On entry, a pointer to a buffer that contains the data you want to be written to the specified storage object.

`sizeWritten` On exit, the number of bytes of data written to the specified storage object.

DESCRIPTION

The `Q3Storage_SetData` function sets the data associated with the storage object specified by the `storage` parameter to the data specified by the `dataSize` and `data` parameters. The data is written to the storage object starting at the byte offset specified by the `offset` parameter. `Q3Storage_SetData` returns, in the `sizeWritten` parameter, the number of bytes of data written to the storage object. If the value returned in the `sizeWritten` parameter is less than the number of bytes requested in the `dataSize` parameter, then the end of the storage object's private data occurs at the distance `offset` + `sizeWritten` from the beginning of the private data.

Creating and Accessing Memory Storage Objects

QuickDraw 3D provides routines for creating and managing memory storage objects.

Q3MemoryStorage_New

You can use the `Q3MemoryStorage_New` function to create a new memory storage object.

```
TQ3StorageObject Q3MemoryStorage_New (
                    const unsigned char *buffer,
                    unsigned long validSize);
```

`buffer` A pointer to a buffer in memory, or `NULL`.

`validSize` The size, in bytes, of the valid metafile data contained in the specified buffer. If `buffer` is set to `NULL`, this parameter specifies the initial size and also the grow size of the buffer that QuickDraw 3D allocates internally.

DESCRIPTION

The `Q3MemoryStorage_New` function returns, as its function result, a new memory storage object associated with the data in the buffer specified by the `buffer` and `validSize` parameters. The data in the specified buffer is copied into internal QuickDraw 3D memory, so you can dispose of the buffer if `Q3MemoryStorage_New` returns successfully.

If you pass the value `NULL` in the `buffer` parameter, QuickDraw 3D allocates a buffer of `validSize` bytes, increases the buffer by that size whenever necessary, and later disposes of the buffer when the associated storage object is closed or disposed of. If `buffer` is set to `NULL` and `validSize` is set to 0, QuickDraw 3D uses a default initial buffer and grow size.

If `Q3MemoryStorage_New` cannot create a new storage object, it returns the value `NULL`.

ERRORS

`kQ3ErrorOutOfMemory`

Q3MemoryStorage_NewBuffer

You can use the `Q3MemoryStorage_NewBuffer` function to create a new memory storage object. The data you provide is not copied into QuickDraw 3D memory.

```
TQ3StorageObject Q3MemoryStorage_NewBuffer (
                    unsigned char *buffer,
                    unsigned long validSize,
                    unsigned long bufferSize);
```

`buffer` A pointer to a buffer in memory, or `NULL`.

`validSize` The size, in bytes, of the valid metafile data contained in the specified buffer. If `buffer` is set to `NULL`, this parameter specifies the initial size and also the grow size of the buffer that QuickDraw 3D allocates internally.

`bufferSize` The size, in bytes, of the specified buffer.

DESCRIPTION

The `Q3MemoryStorage_NewBuffer` function returns, as its function result, a new memory storage object associated with the buffer specified by the `buffer` and `validSize` parameters. The data in the specified buffer is not copied into internal QuickDraw 3D memory, so your application must not access that buffer until the associated storage object is closed or disposed of.

If you pass the value `NULL` in the `buffer` parameter, QuickDraw 3D allocates a buffer of `validSize` bytes, increases the buffer by that size whenever necessary, and later disposes of the buffer when the associated storage object is closed or disposed of. If `buffer` is set to `NULL` and `validSize` is set to 0, QuickDraw 3D uses a default initial buffer and grow size.

The `bufferSize` parameter specifies the size of the specified buffer. The `validSize` parameter specifies the size of the valid metafile data contained in the buffer. The value of the `validSize` parameter should always be less than or equal to the value of the `bufferSize` parameter. This allows you to maintain other data in the buffer following the valid metafile data.

If `Q3MemoryStorage_NewBuffer` cannot create a new storage object, it returns the value `NULL`.

ERRORS

`kQ3ErrorOutOfMemory`

Q3MemoryStorage_Set

You can use the `Q3MemoryStorage_Set` function to set the data of a memory storage object.

```
TQ3Status Q3MemoryStorage_Set (
                    TQ3StorageObject storage,
                    const unsigned char *buffer,
                    unsigned long validSize);
```

`storage` A memory storage object.

`buffer` A pointer to a contiguous block of memory to be associated with the specified storage object, or `NULL`.

`validSize` The size, in bytes, of the valid metafile data contained in the specified buffer. If `buffer` is set to `NULL`, this parameter specifies the initial size and also the grow size of the buffer that QuickDraw 3D allocates internally.

DESCRIPTION

The `Q3MemoryStorage_Set` function sets the data for the memory storage object specified by the `storage` parameter to the values specified in the `buffer` and `validSize` parameters. The data in the specified buffer is copied into internal QuickDraw 3D memory, so you can dispose of the buffer if `Q3MemoryStorage_Set` returns successfully.

If you pass the value `NULL` in the `buffer` parameter, QuickDraw 3D allocates a buffer of `validSize` bytes, increases the buffer by that size whenever necessary, and later disposes of the buffer when the associated storage object is closed or disposed of. If `buffer` is set to `NULL` and `validSize` is set to 0, and if the `buffer` parameter was set to `NULL` when the storage object was created, QuickDraw 3D uses a default initial buffer and grow size.

SPECIAL CONSIDERATIONS

You must not use `Q3MemoryStorage_Set` with an open memory storage object.

ERRORS

```
kQ3ErrorAccessRestricted
kQ3ErrorInvalidObjectParameter
```

Q3MemoryStorage_GetBuffer

You can use the `Q3MemoryStorage_GetBuffer` function to get the data of a memory storage object.

```
TQ3Status Q3MemoryStorage_GetBuffer (
                    TQ3StorageObject storage,
                    unsigned char **buffer,
                    unsigned long *validSize,
                    unsigned long *bufferSize);
```

`storage`	A memory storage object.
`buffer`	On entry, a pointer to a pointer. On exit, a pointer to a pointer to the block of memory associated with the specified storage object.
`validSize`	On exit, the size, in bytes, of the valid metafile data contained in the specified buffer.
`bufferSize`	On exit, the size, in bytes, of the block of memory whose address is returned through the `buffer` parameter.

DESCRIPTION

The `Q3MemoryStorage_GetBuffer` function returns, in the `buffer` and `bufferSize` parameters, the address and size of the block of memory currently associated with the memory storage object specified by the `storage` parameter. Note that the returned address is the address of the storage object's data, not of a *copy* of that data. As a result, the returned pointer may become a dangling pointer if the buffer holding the storage object's data is dynamically reallocated (perhaps because additional data was written to the object).

ERRORS

```
kQ3ErrorAccessRestricted
kQ3ErrorInvalidObjectParameter
```

Q3MemoryStorage_SetBuffer

You can use the `Q3MemoryStorage_SetBuffer` function to set the data of a memory storage object.

```
TQ3Status Q3MemoryStorage_SetBuffer (
                    TQ3StorageObject storage,
                    unsigned char *buffer,
                    unsigned long validSize,
                    unsigned long bufferSize);
```

`storage`	A memory storage object.

`buffer`	A pointer to a block of memory to be associated with the specified storage object, or `NULL`.
`validSize`	The size, in bytes, of the valid metafile data contained in the specified buffer. If the value of `buffer` is `NULL`, this parameter specifies the initial size and also the grow size of the buffer that QuickDraw 3D allocates internally.
`bufferSize`	The size, in bytes, of the specified buffer.

DESCRIPTION

The `Q3MemoryStorage_SetBuffer` function sets the buffer location, size, and valid size of the memory storage object specified by the `storage` parameter to the values specified in the `buffer`, `bufferSize`, and `validSize` parameters.

If you pass the value `NULL` in the `buffer` parameter, QuickDraw 3D allocates a buffer of `validSize` bytes, increases the buffer by that size whenever necessary, and later disposes of the buffer when the associated storage object is closed or disposed of. If `buffer` is set to `NULL` and `validSize` is set to 0, QuickDraw 3D uses a default initial buffer and grow size.

SPECIAL CONSIDERATIONS

You must not use `Q3MemoryStorage_SetBuffer` with an open memory storage object.

Q3MemoryStorage_GetType

You can use the `Q3MemoryStorage_GetType` function to get the type of a memory storage object.

```
TQ3ObjectType Q3MemoryStorage_GetType (TQ3StorageObject storage);
```

`storage`	A memory storage object.

DESCRIPTION

The `Q3MemoryStorage_GetType` function returns, as its function result, the type of the memory storage object specified by the `storage` parameter. The types of memory storage objects currently supported by QuickDraw 3D are defined by this constant:

```
kQ3MemoryStorageTypeHandle
```

If the specified memory storage object is invalid or is not of this type, `Q3MemoryStorage_GetType` returns the value `kQ3ObjectTypeInvalid`.

ERRORS

```
kQ3ErrorNoSubclass
kQ3ErrorInvalidObjectParameter
kQ3ErrorNULLParameter
```

Creating and Accessing Handle Storage Objects

QuickDraw 3D provides routines for creating and managing handle storage objects.

Q3HandleStorage_New

You can use the `Q3HandleStorage_New` function to create a new handle storage object.

```
TQ3StorageObject Q3HandleStorage_New (
                    Handle handle,
                    unsigned long validSize);
```

`handle` A handle to a buffer in memory, or `NULL`.

`validSize` The size, in bytes, of the specified buffer.

DESCRIPTION

The `Q3HandleStorage_New` function returns, as its function result, a new handle storage object associated with the buffer specified by the `handle` and `validSize` parameters. Your application must not access that buffer until the associated storage object is closed or disposed of. If `Q3HandleStorage_New` cannot create a new storage object, it returns the value `NULL`. If you pass the value `NULL` in the `handle` parameter, QuickDraw 3D allocates a buffer of the specified size and later disposes of that buffer when the associated storage object is closed or disposed of.

ERRORS

`kQ3ErrorOutOfMemory`

Q3HandleStorage_Get

You can use the `Q3HandleStorage_Get` function to get information about a handle storage object.

```
TQ3Status Q3HandleStorage_Get (
                    TQ3StorageObject storage,
                    Handle *handle,
                    unsigned long *validSize);
```

`storage` A handle storage object.

`handle` On entry, a pointer to a handle. On exit, a pointer to a handle to the block of memory associated with the specified storage object.

`validSize` On exit, the size, in bytes, of the block of memory whose address is returned through the `buffer` parameter.

DESCRIPTION

The `Q3HandleStorage_Get` function returns, in the `handle` and `validSize` parameters, the handle and size of the block of memory currently associated with the handle storage object specified by the `storage` parameter. Note that the

returned handle is a handle to the storage object's data, not of a *copy* of that data.

ERRORS

```
kQ3ErrorInvalidObjectParameter
kQ3ErrorNULLParameter
```

Q3HandleStorage_Set

You can use the `Q3HandleStorage_Set` function to set information about a handle storage object.

```
TQ3Status Q3HandleStorage_Set (
                    TQ3StorageObject storage,
                    Handle handle,
                    unsigned long validSize);
```

`storage`	A handle storage object.
`handle`	A handle to a contiguous block of memory to be associated with the specified storage object, or `NULL`.
`validSize`	The size, in bytes, of the specified buffer.

DESCRIPTION

The `Q3HandleStorage_Set` function sets the buffer location and size of the handle storage object specified by the `storage` parameter to the values specified in the `handle` and `validSize` parameters. If you pass the value `NULL` in the `handle` parameter, QuickDraw 3D allocates a buffer of the specified size and later disposes of that buffer when the associated storage object is closed or disposed of. If you pass `NULL` in `handle` and 0 in `validSize`, QuickDraw 3D allocates a buffer of a private default size.

SPECIAL CONSIDERATIONS

You must not use `Q3HandleStorage_Set` with an open handle storage object.

ERRORS

`kQ3ErrorInvalidObjectParameter`

Creating and Accessing Macintosh Storage Objects

QuickDraw 3D provides routines for creating and managing Macintosh storage objects.

Q3MacintoshStorage_New

You can use the `Q3MacintoshStorage_New` function to create a new Macintosh storage object.

```
TQ3StorageObject Q3MacintoshStorage_New (short fsRefNum);
```

`fsRefNum` A file reference number of the data fork of a Macintosh file. This file must already be open.

DESCRIPTION

The `Q3MacintoshStorage_New` function returns, as its function result, a new storage object associated with the Macintosh file specified by the `fsRefNum` parameter. The specified file is assumed to be open, and it must remain open as long as you use the returned storage object. In addition, you are responsible for closing the file once the associated storage object has been closed or disposed of. If `Q3MacintoshStorage_New` cannot create a new storage object, it returns the value `NULL`.

ERRORS

`kQ3ErrorOutOfMemory`

Q3MacintoshStorage_Get

You can use the `Q3MacintoshStorage_Get` function to get information about a Macintosh storage object.

```
TQ3Status Q3MacintoshStorage_Get (
                    TQ3StorageObject storage,
                    short *fsRefNum);
```

`storage` A Macintosh storage object.

`fsRefNum` On exit, the file reference number of the Macintosh file associated with the specified storage object.

DESCRIPTION

The `Q3MacintoshStorage_Get` function returns, in the `fsRefNum` parameter, the file reference number of the Macintosh file associated with the Macintosh storage object specified by the `storage` parameter.

Q3MacintoshStorage_Set

You can use the `Q3MacintoshStorage_Set` function to set information about a Macintosh storage object.

```
TQ3Status Q3MacintoshStorage_Set (
                    TQ3StorageObject storage,
                    short fsRefNum);
```

`storage` A Macintosh storage object.

`fsRefNum` A file reference number.

DESCRIPTION

The `Q3MacintoshStorage_Set` function sets the file reference number of the file associated with the Macintosh storage object specified by the `storage` parameter to the number specified by the `fsRefNum` parameter.

SPECIAL CONSIDERATIONS

You must not use `Q3MacintoshStorage_Set` with an open Macintosh storage object.

ERRORS

`kQ3ErrorStorageIsOpen`

Q3MacintoshStorage_GetType

You can use the `Q3MacintoshStorage_GetType` function to get the type of a Macintosh storage object.

```
TQ3ObjectType Q3MacintoshStorage_GetType (TQ3StorageObject storage);
```

`storage` A Macintosh storage object.

DESCRIPTION

The `Q3MacintoshStorage_GetType` function returns, as its function result, the type of the Macintosh storage object specified by the `storage` parameter. The types of Macintosh storage objects currently supported by QuickDraw 3D are defined by this constant:

```
kQ3MacintoshStorageTypeFSSpec
```

If the specified memory storage object is invalid or is not of this type, `Q3MacintoshStorage_GetType` returns the value `kQ3ObjectTypeInvalid`.

ERRORS

```
kQ3ErrorNoSubclass
kQ3ErrorInvalidObjectParameter
kQ3ErrorNULLParameter
```

Creating and Accessing FSSpec Storage Objects

QuickDraw 3D provides routines for creating and managing Macintosh storage objects specified using a file system specification structure.

Q3FSSpecStorage_New

You can use the `Q3FSSpecStorage_New` function to create a new memory storage object specified using a file system specification structure.

```
TQ3StorageObject Q3FSSpecStorage_New (const FSSpec *fs);
```

`fs` A file system specification structure specifying the name and location of a Macintosh file.

DESCRIPTION

The `Q3FSSpecStorage_New` function returns, as its function result, a new storage object associated with the Macintosh file specified by the `fs` parameter. The specified file is assumed to be closed. QuickDraw 3D opens the file, and, when the associated storage object is closed or disposed of, QuickDraw 3D closes the file. If `Q3FSSpecStorage_New` cannot create a new storage object, it returns the value `NULL`.

ERRORS

```
kQ3ErrorOutOfMemory
kQ3ErrorNULLParameter
```

Q3FSSpecStorage_Get

You can use the `Q3FSSpecStorage_Get` function to get information about an `FSSpec` storage object.

```
TQ3Status Q3FSSpecStorage_Get (
                    TQ3StorageObject storage,
                    FSSpec *fs);
```

storage	A Macintosh FSSpec storage object.
fs	On entry, a pointer to a file system specification structure. On exit, a pointer to the file system specification structure associated with the specified Macintosh FSSpec storage object.

DESCRIPTION

The Q3FSSpecStorage_Get function returns, through the fs parameter, the file system specification structure associated with the Macintosh FSSpec storage object specified by the storage parameter.

Q3FSSpecStorage_Set

You can use the Q3FSSpecStorage_Set function to set information about an FSSpec storage object.

```
TQ3Status Q3FSSpecStorage_Set (
                    TQ3StorageObject storage,
                    const FSSpec *fs);
```

storage	A Macintosh FSSpec storage object.
fs	A file system specification structure specifying the name and location of a Macintosh file.

DESCRIPTION

The Q3FSSpecStorage_Set function sets the file system specification structure of the file associated with the Macintosh FSSpec storage object specified by the storage parameter to the structure specified by the fs parameter.

SPECIAL CONSIDERATIONS

You must not use Q3FSSpecStorage_Set with an open Macintosh FSSpec storage object.

ERRORS

`kQ3ErrorStorageIsOpen`

Creating and Accessing UNIX Storage Objects

QuickDraw 3D provides routines for creating and managing UNIX storage objects.

Note
You need to link your application with the standard I/O library to use these functions. ◆

Q3UnixStorage_New

You can use the `Q3UnixStorage_New` function to create a new UNIX storage object.

```
TQ3StorageObject Q3UnixStorage_New (FILE *stdFile);
```

`stdFile` A pointer to a file. This file must already be open.

DESCRIPTION

The `Q3UnixStorage_New` function returns, as its function result, a new UNIX storage object associated with the file specified by the `stdFile` parameter. The specified file is assumed to be open, and it must remain open as long as you use the returned storage object. In addition, you are responsible for closing the file once the associated storage object has been closed or disposed of. If `Q3UnixStorage_New` cannot create a new storage object, it returns the value `NULL`.

ERRORS

`kQ3ErrorOutOfMemory`
`kQ3ErrorNULLParameter`

Q3UnixStorage_Get

You can use the `Q3UnixStorage_Get` function to get information about a UNIX storage object.

```
TQ3Status Q3UnixStorage_Get (TQ3StorageObject storage, FILE **stdFile);
```

`storage` A UNIX storage object.

`stdFile` On entry, a pointer to a `FILE` structure. On exit, a pointer to the `FILE` structure associated with the specified UNIX storage object.

DESCRIPTION

The `Q3UnixStorage_Get` function returns, through the `stdFile` parameter, the `FILE` structure associated with the UNIX storage object specified by the `storage` parameter.

ERRORS

```
kQ3ErrorAccessRestricted
kQ3ErrorInvalidObjectParameter
```

Q3UnixStorage_Set

You can use the `Q3UnixStorage_Set` function to set information about a UNIX storage object.

```
TQ3Status Q3UnixStorage_Set (TQ3StorageObject storage, FILE *stdFile);
```

`storage` A UNIX storage object.

`stdFile` A pointer to a `FILE` structure.

DESCRIPTION

The `Q3UnixStorage_Set` function sets the `FILE` structure associated with the UNIX storage object specified by the `storage` parameter to the structure specified by the `stdFile` parameter.

SPECIAL CONSIDERATIONS

You must not use `Q3UnixStorage_Set` with an open UNIX storage object.

ERRORS

```
kQ3ErrorAccessRestricted
kQ3ErrorInvalidObjectParameter
kQ3ErrorStorageIsOpen
```

Q3UnixStorage_GetType

You can use the `Q3UnixStorage_GetType` function to get the type of a UNIX storage object.

```
TQ3ObjectType Q3UnixStorage_GetType (TQ3StorageObject storage);
```

`storage` A UNIX storage object.

DESCRIPTION

The `Q3UnixStorage_GetType` function returns, as its function result, the type of the UNIX storage object specified by the `storage` parameter. The types of UNIX storage objects currently supported by QuickDraw 3D are defined by this constant:

```
kQ3UnixStorageTypePath
```

If the specified memory storage object is invalid or is not of this type, `Q3UnixStorage_GetType` returns the value `kQ3ObjectTypeInvalid`.

ERRORS

```
kQ3ErrorNoSubclass
kQ3ErrorInvalidObjectParameter
kQ3ErrorNULLParameter
```

Creating and Accessing UNIX Path Name Storage Objects

QuickDraw 3D provides routines for creating and managing UNIX storage objects specified using a path name.

Note
You need to link your application with the standard I/O library to use these functions. ◆

Q3UnixPathStorage_New

You can use the `Q3UnixPathStorage_New` function to create a new UNIX storage object specified using a path name.

```
TQ3StorageObject Q3UnixPathStorage_New (const char *pathName);
```

`pathName` A path name of a file. The path name is a null-terminated C string.

DESCRIPTION

The `Q3UnixPathStorage_New` function returns, as its function result, a new storage object associated with the file specified by the `pathName` parameter. The specified file is assumed to be closed. QuickDraw 3D opens the file (by calling `fopen`) and, when the associated storage object is closed or disposed of, QuickDraw 3D closes the file (by calling `fclose`). If `Q3UnixPathStorage_New` cannot create a new storage object, it returns the value `NULL`.

ERRORS

```
kQ3ErrorOutOfMemory
kQ3ErrorNULLParameter
```

Q3UnixPathStorage_Get

You can use the `Q3UnixPathStorage_Get` function to get information about a UNIX path name storage object.

```
TQ3Status Q3UnixPathStorage_Get (
                    TQ3StorageObject storage,
                    char *pathName);
```

`storage` A UNIX path name storage object.

`pathName` On entry, a pointer to a block of memory large enough to hold a string of size `kQ3StringMaximumLength`. The path name of the file associated with the specified storage object is copied into that block of memory. The path name is a null-terminated C string.

DESCRIPTION

The `Q3UnixPathStorage_Get` function returns, through the `pathName` parameter, a copy of the path name of the file associated with the UNIX path storage object specified by the `storage` parameter.

ERRORS

```
kQ3ErrorInvalidObjectParameter
kQ3ErrorNULLParameter
```

Q3UnixPathStorage_Set

You can use the `Q3UnixPathStorage_Set` function to set information about a UNIX path name storage object.

```
TQ3Status Q3UnixPathStorage_Set (
                    TQ3StorageObject storage,
                    const char *pathName);
```

`storage` A UNIX path name storage object.

`pathName` A pointer to the path name of a file. The path name is a null-terminated C string. (A file does not yet need to exist in that location.)

DESCRIPTION

The `Q3UnixPathStorage_Set` function sets the path name of the file associated with the UNIX path name storage object specified by the `storage` parameter to the string pointed to by the `pathName` parameter.

SPECIAL CONSIDERATIONS

You must not use `Q3UnixPathStorage_Set` with an open UNIX path name storage object.

ERRORS

```
kQ3ErrorAccessRestricted
kQ3ErrorInvalidObjectParameter
```

Creating and Accessing Windows Storage Objects

QuickDraw 3D provides routines for creating and managing Windows storage objects.

Q3Win32Storage_New

You use the `Q3Win32Storage_New` function to create a Windows storage object.

```
TQ3StorageObject Q3Win32Storage_New (const HANDLE hFile);
```

`hFile` A handle to a file.

DESCRIPTION

The `Q3Win32Storage_New` function returns, as its function result, a new storage object associated with the file specified by the `hFile` parameter. The specified

file is assumed to be closed. QuickDraw 3D opens the file and, when the associated storage object is closed or disposed of, QuickDraw 3D closes the file. If `Q3Win32Storage_New` cannot create a new storage object, it returns the value `NULL`.

EXAMPLES

The `HANDLE` type is a native Windows entity that's created using the Windows `CreateFile` call. The following illustrates a typical way of using this call to read an existing variable of type `HANDLE`:

```
hFile = CreateFile(
    pathName,               // pointer to name of the file
    GENERIC_READ,           // access (read-write) mode
    0,                      // share mode
    NULL,                   // pointer to security descriptor
    OPEN_EXISTING,          // how to create
    FILE_ATTRIBUTE_NORMAL,  // file attributes
    NULL                    // handle to file with attributes
                            // to copy
);
A3Assert(hFile != INVALID_HANDLE_VALUE);
A3Assert((srcStorage = Q3Win32Storage_New(hFile)) != NULL);
```

The following is a typical way of using the Windows `CreateFile` call to write to a variable of type `HANDLE`:

```
hFile = CreateFile(
    pathName,                   // pointer to name of the file
    GENERIC_WRITE,              // access (read-write) mode
    0,                          // share mode
    NULL,                       // pointer to security descriptor
    CREATE_NEW,                 // how to create
    FILE_ATTRIBUTE_NORMAL,      // file attributes
    NULL                        // handle to file with
                                // attributes to copy
);

A3Assert(hFile != INVALID_HANDLE_VALUE);
A3Assert((dstStorage = Q3Win32Storage_New(hFile)) != NULL);
```

Q3Win32Storage_Get

You can use the `Q3Win32Storage_Get` function to get the file associated with a Windows storage object.

```
TQ3Status Q3Win32Storage_Get (
                    TQ3StorageObject storage,
                    const HANDLE *hFile);
```

`storage` A Windows storage object.

`hFile` On exit, a handle to the file associated with the specified storage object.

DESCRIPTION

The `Q3Win32Storage_Get` function returns, through the `hFile` parameter, a handle to the file associated with the Windows storage object specified by the `storage` parameter.

Q3Win32Storage_Set

You can use the `Q3Win32Storage_Set` function to set the file associated with a Windows storage object.

```
TQ3Status Q3Win32Storage_Set (
                    TQ3StorageObject storage,
                    const HANDLE hFile);
```

`storage` A Windows storage object.

`hFile` A handle to a file.

DESCRIPTION

The `Q3Win32Storage_Set` function sets the file associated with the Windows storage object specified by the `storage` parameter to the file specified by the `hFile` parameter.

Storage Object Errors

The following errors may be returned by storage object routines. A list of general QuickDraw 3D errors is given in “QuickDraw 3D Errors, Warnings, and Notices” (page 87).

```
kQ3ErrorStorageInUse
kQ3ErrorStorageAlreadyOpen
kQ3ErrorStorageNotOpen
kQ3ErrorStorageIsOpen
```

CHAPTER 17

File Objects

This chapter describes file objects and the functions you can use to manipulate them. You use file objects, together with storage objects, to read and write data stored in the QuickDraw 3D Object Metafile format. A storage object connects a physical storage device to a file object.

To use this chapter, you should already be familiar with the QuickDraw 3D class hierarchy, described in the chapter "QuickDraw 3D Objects." You also need to know how to create and configure storage objects, as explained in the chapter "Storage Objects."

This chapter begins by describing file objects and their features. Then it shows how to create and manipulate file objects. The section "File Objects Reference," beginning on page 1029 provides a complete description of file objects and the routines you can use to create and manipulate them.

Note
For a discussion of file-oriented custom object methods, see "I/O Methods" (page 233). ◆

About File Objects

A **file object** (or, more briefly, a **file**) is a type of QuickDraw 3D object that you use to read and write data that conforms to the **QuickDraw 3D Object Metafile (3DMF),** a standard file format intended to facilitate the interchange of three-dimensional data among applications. You can use the 3DMF both as a 3D data storage format and as a 3D data interchange format. For example, when a user saves a 3D model created by your application, you can write the data to a file object. The data-writing methods of the file object and its associated storage object ensure that the data in the piece of storage associated with that storage object (for example, a file on disk or a block of memory) conforms to the 3DMF

specification. All other applications capable of handling 3DMF files can thus open and read that data.

By using file objects, you can insulate your application from having to know the actual details of the QuickDraw 3D Object Metafile standard. You use file object routines to read and write data in a piece of storage that conforms to the 3DMF and, if necessary, to get information about that storage. In all likelihood, you'll need to know about the details of the 3DMF only if you cannot use file objects to access 3DMF data. For instance, you would need to know the structure of the 3DMF if you wanted to read and write 3DMF files using a 3D graphics system other than QuickDraw 3D.

Note
See *Quickdraw 3D 1.5 Metafile Reference* for complete information about the structure of the QuickDraw 3D Object Metafile. ◆

File I/O

The relationship between file objects and storage objects is similar to that between view objects and draw context objects. A draw context object receives the raw data needed to draw an image on a particular window system, and the associated view object is an abstraction in which you perform all drawing. Similarly, a storage object receives the raw data read from or written to a particular piece of storage, and the associated file object is an abstraction in which you perform all I/O operations. View objects maintain information about the current state of the drawing, and file objects maintain information about the current state of I/O operations. Just as you must perform all drawing in a rendering loop, between calls to `Q3View_StartRendering` and `Q3View_EndRendering`, you must perform all file writing in a **writing loop,** between calls to `Q3View_StartWriting` and `Q3View_EndWriting`. See "Writing Data to a File Object," beginning on page 1028 for more information on writing 3DMF data.

A QuickDraw 3D file object is of type `TQ3FileObject`, which is a type of shared object. QuickDraw 3D currently provides no subclasses of the `TQ3FileObject` type.

File Types

As mentioned earlier, the data associated with a file object must conform to the QuickDraw 3D Object Metafile standard. That standard defines two general forms for the 3D data: text form and binary form. A **text file** is a stream of ASCII characters with meaningful labels for each type of object contained in the file (for example, `NURBCurve` for a NURB curve). A **binary file** is a stream of raw binary data, the type of which is indicated by more cryptic object type codes (for example, `nrbc` for a NURB curve). The text form is most useful when you're writing and debugging your application, but the binary form is usually smaller (requiring less storage space on disk or in memory) and can be read and written much faster.

IMPORTANT

Disk-based metafile data, whether a text file or a binary file, should be contained in a file of type '`3DMF`'. ▲

In addition, there are three ways to organize the data in a text or binary file object. A file object can be organized in normal mode, stream mode, or database mode.

In **normal mode,** a file object contains a **table of contents** that lists all multiply-referenced objects in the file. This is usually the most compact file object organization, but it requires random access to the file object data in order to resolve references. (It might not, therefore, be the best mode to use when transferring 3D data to a remote machine on a network.)

In **stream mode,** a file object does not contain a table of contents and any references to objects are simply copies of the objects themselves. This may result in a larger file than normal mode, but it allows the file object to be processed sequentially, without random access.

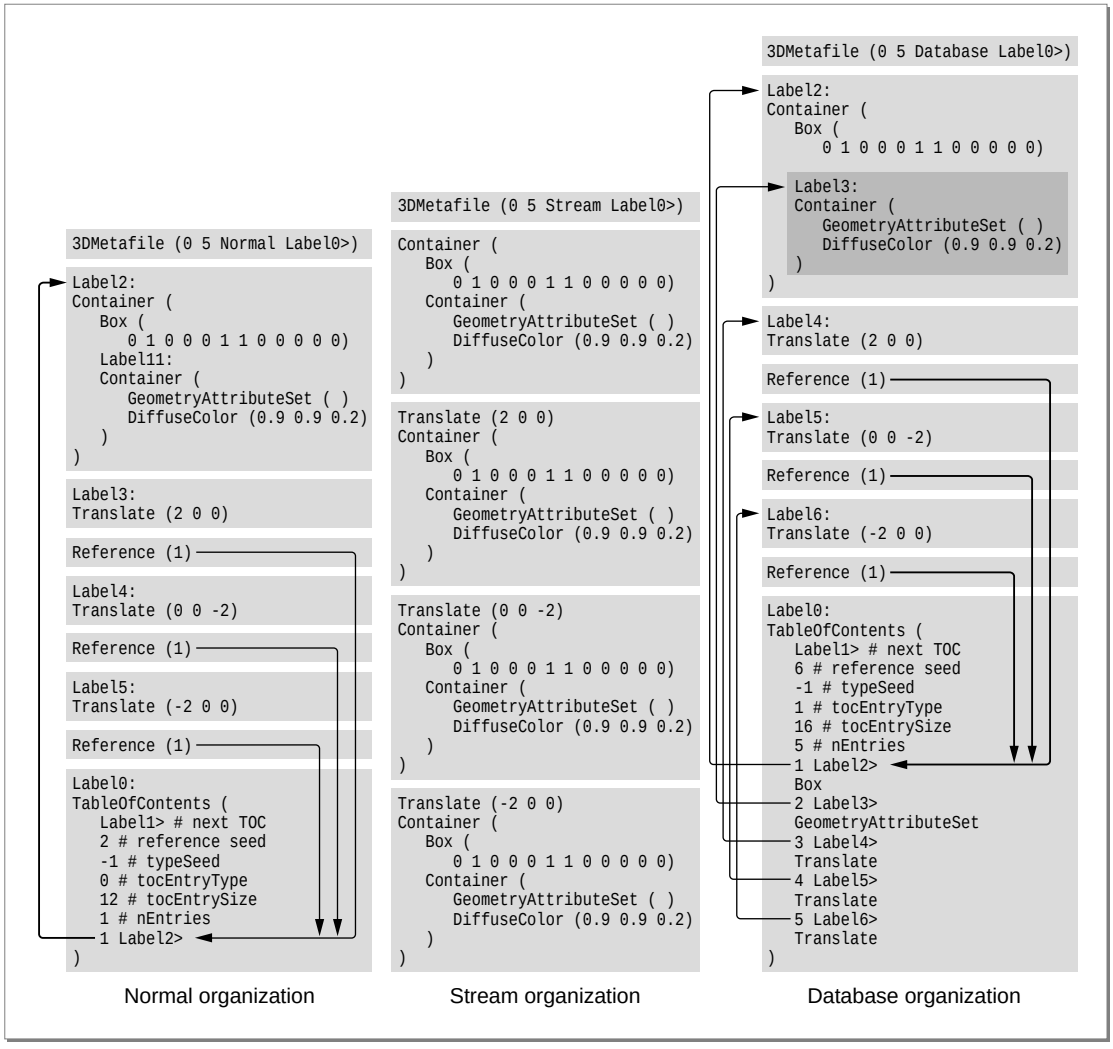
In **database mode,** a file object contains a table of contents that lists *every* object in the file, whether or not it is referenced within the file. This organization is useful if you want to determine what information a file object contains without having to read and process the entire file. This would be useful, for example, for creating a catalog of textures.

Figure 17-1 shows a sample text file object organized in each of these three ways. Once again, for complete information about the types of file objects and the ways of organizing them, see the *3D Metafile Reference*.

View Hints

To include in a metafile information about the lights, the renderer, the camera, and other view settings, you can by create and write a view hints object. A **view hints object** is an object in a metafile that gives hints about how to configure a scene. For instance, you can create a view hints object (by calling `Q3ViewHints_New`) and then record a view's current settings by calling functions like `Q3ViewHints_SetRenderer` and `Q3ViewHints_SetCamera`. Conversely, when you are reading objects from a metafile and you encounter a view hints object in the file, you can use the information in that object to configure a view object, thereby reconstructing the image as accurately as possible. Or, you can choose to ignore the information in a view hints object you find in a metafile. For information about using view hints objects see "Managing View Hints Objects" (page 1074).

Figure 17-1 Types of file objects



Using File Objects

You use file objects to read 3DMF data from or write 3DMF data to a storage object, which represents a physical storage device available on a computer. Before you can access 3DMF data in a piece of storage, however, you need to create a storage object to represent the physical storage device, create a file object, and attach the file object to the storage object.

Creating a File Object

To access the data in a piece of storage that conforms to the 3DMF standard (such as a file on disk or a block of memory on the Clipboard), you need to create a new storage object, create a new file object, and attach the file object to the storage object. Thereafter, you can open the file object and read the data in it or write data to it. Listing 17-1 illustrates how to create storage and file objects and attach them to one another.

The `MyGetInputFile` function defined in Listing 17-1 calls the application-defined routine `MyGetInputFileName` to get the name of the disk file to open. Then it calls `Q3FSSpecStorage_New` to create a new storage object associated with that disk file and `Q3File_New` to create a new file object. If both creation calls complete successfully, `MyGetInputFile` calls `Q3File_SetStorage` to attach the file object to the storage object.

Note
See the chapter "Storage Objects" for complete details on creating storage objects. ◆

Listing 17-1 Creating a new file object

```
TQ3FileObject MyGetInputFile (void)
{
    TQ3FileObject           myFileObj;
    TQ3StorageObject        myStorageObj;
    OSType                  myFileType;
    FSSpec                  myFSSpec;
```

```
    if (MyGetInputFileName(&myFSSpec) == kQ3False)
        return(NULL);

    /*Create new storage object and new file object.*/
    if(((myStorageObj = Q3FSSpecStorage_New(&myFSSpec)) == NULL)
        || ((myFileObj = Q3File_New()) == NULL))
    {
        if (myStorageObj)
            Q3Object_Dispose(myStorageObj);
        return(NULL);
    }

    /*Set the storage for the file object.*/
    Q3File_SetStorage(myFileObj, myStorageObj);
    Q3Object_Dispose(myStorageObj);

    return (myFileObj);
}
```

Notice that the call to `Q3File_SetStorage` is followed immediately by a call to `Q3Object_Dispose`. The call to `Q3File_SetStorage` increases the reference count of the storage object, and the call to `Q3Object_Dispose` simply decreases that count.

Reading Data from a File Object

The data in an 3DMF file is organized into discrete units called **metafile objects** (or, more briefly, and despite the risk of confusion with QuickDraw 3D objects, **objects**). You read data from an 3DMF file by reading each individual metafile object in it (by calling the `Q3File_ReadObject` function), until you reach the end of the file. Listing 17-2 illustrates how to read the metafile objects in an 3DMF file.

The `MyRead3DMFModel` function defined in Listing 17-2 opens a file object and sequentially reads each metafile object in the 3DMF file into a QuickDraw 3D object. `MyRead3DMFModel` determines the type of the QuickDraw 3D object read. If the object is a view hints object, `MyRead3DMFModel` returns that object in the `viewHints` parameter. If the object isn't a view object, it must be some other drawable QuickDraw 3D object. In that case, `MyRead3DMFModel` either returns that object in the `model` parameter (if there are no more objects in the 3DMF file) or adds it to a display group. When it executes successfully, `MyRead3DMFModel` returns both a 3D model and a view hints object to its caller.

Listing 17-2 Reading metafile objects

```
TQ3Status MyRead3DMFModel
                    (TQ3FileObject file, TQ3Object *model, TQ3Object *viewHints)
{
    TQ3Object           myGroup;
    TQ3Object           myObject;

    /*Initialize view hints and model to be returned.*/
    *viewHints = NULL;
    *model = NULL;
    myGroup = NULL;
    myObject = NULL;

    /*Open the file object and exit gracefully if unsuccessful.*/
    if (Q3File_OpenRead(file, NULL) != kQ3Success)
    {
        DoError("MyRead3DMFModel", "Reading failed %s", filename);
        return kQ3Failure;
    }

    while (Q3File_IsEndOfFile(file) == kQ3False)
    {
        myObject = NULL;
        /*Read a metafile object from the file object.*/
        myObject = Q3File_ReadObject(file);
        if (myObject == NULL)
            continue;

        /*Save a view hints object, and add any drawable objects to a group.*/
        if (Q3Object_IsType(myObject, kQ3SharedTypeViewHints))
        {
            if (*viewHints == NULL)
            {
                *viewHints = myObject;
                myObject = NULL;
            }
        }
        else if (Q3Object_IsDrawable(myObject))
        {
            if (myGroup)
```

```
            {
                Q3Group_AddObject(myGroup, myObject);
            }
            else if (*model == NULL)
            {
                *model = myObject;
                myObject = NULL;
            }
            else
            {
                myGroup = Q3DisplayGroup_New();
                Q3Group_AddObject(myGroup, *model);
                Q3Group_AddObject(myGroup, myObject);
                Q3Object_Dispose(*model);
                *model = myGroup;
            }
        }
        if (myObject != NULL)
            Q3Object_Dispose(myObject);
    }

    if (Q3Error_Get(NULL) != kQ3ErrorNone)
    {
        if (*model != NULL) {
            Q3Object_Dispose(*model);
            *model = NULL;
        }

        if (*viewHints != NULL) {
            Q3Object_Dispose(*viewHints);
            *viewHints = NULL;
        }
        return (kQ3Failure);
    }
 return kQ3Success;
}
```

Writing Data to a File Object

To write a model or other 3D data into a file conforming to the QuickDraw 3D Object Metafile format, you can use submit calls (such as `Q3Object_Submit`) with an open file object that is attached to a storage object. Depending on the complexity of the model and the amount of available memory, QuickDraw 3D might need to traverse the model more than once to write the data to the target physical storage device. Accordingly, you should perform all write operations within a **writing loop,** bracketed by calls to `Q3View_StartWriting` and `Q3View_EndWriting`. Listing 17-3 illustrates a simple writing loop.

Listing 17-3 Writing 3D data to a file object

```
Q3View_StartWriting(myView, myFileObj);
do {
    Q3Object_Submit(myModel, myView);
    Q3Polyline_Submit(&myAnimatedData, myView);
    Q3TriGrid_Submit(&myBumpExtrapolationGrid, myView);
} while (Q3View_EndWriting(myView) == kQ3ViewStatusRetraverse);
```

Metafile External References

Suppose E is a metafile that contains an object R that you wish to reference from another metafile M. E must have an entry to object R in its Table of Contents (TOC). There are two ways to achieve this: you can write E out in database mode or make sure that R is written out twice in E while E is in normal mode.

With such a metafile E, the first step in writing object R out as an external reference is to read R from E. You must open E using the `UnixPath` storage class. While E is still open for reading, call `Q3File_MarkAsExternalReference` on R and the `TQ3FileObject` associated with E. This marks R as an object that will always be written out as an external reference. This means that whenever a `Q3..._Submit` call is made on the object in a write loop, an external reference object is written out that specifies the location of the object in E.

The process of reading metafiles containing external references is transparent to the user, so long as no problems arise. If the location of the object is not given correctly by the UNIX pathname, then the read call on the external reference object will return `NULL`. Also, the file object containing the externally referenced object must not currently be open for reading or writing.

The external reference object contains two pieces of information: the name (or pathname) of the metafile that's being externally referenced (E in this example), and the reference ID of the object R.

Once a file containing external references has been created, calling `Q3File_GetExternalReferences` returns the names of the files that are externally referred to by M. If no files are externally referred to, the call returns `NULL`. If one or more files are externally referred to, `Q3File_GetExternalReferences` returns a group that contains one `Q3String` object for each external reference object in the metafile. The `Q3String` object contains the name (in general, the pathname) of the file in question. Because one `Q3String` object is produced per external reference, it is possible for the same name to appear in several `Q3String` objects.

Routines that let you access and manipulate external references in metafiles are described in "Custom File Object Routines," beginning on page 1086. For general information about metafiles, see the document *QuickDraw 3D 1.5 Metafile Reference.*

File Objects Reference

This section describes the constants, data structures, and routines that you can use to create and manage file objects.

Constants

This section describes the constants you can use to specify file modes for file objects.

File Mode Flags

QuickDraw 3D defines a set of **file mode flags** to specify a file object's current file mode. The file mode is returned to you when you call `Q3File_OpenRead`, `Q3Open_Write`, or `Q3File_GetMode`.

```
typedef enum TQ3FileModeMasks {
    kQ3FileModeNormal                       = 0,
    kQ3FileModeStream                       = 1 << 0,
```

```
    kQ3FileModeDatabase                     = 1 << 1,
    kQ3FileModeText                         = 1 << 2
} TQ3FileModeMasks;
```

Constant descriptions

`kQ3FileModeNormal`
: Set if the file object is organized in normal mode. A file object is in normal mode if it contains a table of contents that lists all referenced objects in the file object. Normal mode is the most compact metafile representation.

`kQ3FileModeStream`
: Set if the file object is organized in stream mode. A file is in stream mode if there are no internal references in the file. You can use stream mode for reading or writing unidirectional streams, but a file in stream mode is usually larger than a file in normal mode.

`kQ3FileModeDatabase`
: Set if the file object is organized in database mode. A file object is in database mode if the file object lists in its table of contents all shared objects contained in the file object, whether or not those objects are multiply referenced.

`kQ3FileModeText` Set if the file object is a text file. The file object is read as text, using tokens and behaviors appropriate for text file objects.

You can combine the `kQ3FileModeText` mask with any of the other masks, and you can combine the `kQ3FileModeStream` and `kQ3FileModeDatabase` masks in a single file mode.

Data Structures

This section describes the data structures provided by QuickDraw 3D for accessing the data in a text or binary unknown object.

Primitive Types

```
typedef unsigned char   TQ3Uns8;    /* 1 byte unsigned integer */
```

```
typedef signed char     TQ3Int8;    /* 1 byte signed integer */

typedef unsigned short  TQ3Uns16;   /* 2 byte unsigned integer */

typedef signed short    TQ3Int16;   /* 2 byte signed integer */

typedef unsigned long   TQ3Uns32;   /* 4 byte unsigned integer */

typedef signed long     TQ3Int32;   /* 4 byte signed integer */

typedef struct TQ3Uns64 {           /* for the Mac OS */
    unsigned long   hi;
    unsigned long   lo;
} TQ3Uns64;                         /* 8 byte unsigned integer */

typedef struct TQ3Uns64 {           /* for Windows */
    unsigned long   lo;
    unsigned long   hi;
} TQ3Uns64;                         /* 8 byte unsigned integer */

typedef struct TQ3Int64 {           /* for the Mac OS */
    signed long     hi;
    unsigned long   lo;
} TQ3Int64;                         /* 8 byte signed integer */

typedef struct TQ3Int64 {           /* for Windows */
    unsigned long   lo;
    signed long     hi;
} TQ3Int64;                         /* 8 byte signed integer */

typedef float   TQ3Float32;     /* 4 byte floating point number */

typedef double  TQ3Float64;     /* 8 byte floating point number */

typedef TQ3Uns32    TQ3Size;
```

Version and Mode

```
#define Q3FileVersion(majorVersion, minorVersion)
    (TQ3FileVersion) ((((TQ3Uns32) majorVersion & 0xFFFF) << 16)
    | ((TQ3Uns32) minorVersion & 0xFFFF))

typedef unsigned long           TQ3FileVersion;

#define kQ3FileVersionCurrent   Q3FileVersion(1,2)

typedef enum TQ3FileModeMasks {
    kQ3FileModeNormal       = 0,
    kQ3FileModeStream       = 1 << 0,
    kQ3FileModeDatabase     = 1 << 1,
    kQ3FileModeText         = 1 << 2
} TQ3FileModeMasks;
typedef unsigned long       TQ3FileMode;
```

Group Reading States

```
typedef enum TQ3FileReadGroupStateMasks{
    kQ3FileReadWholeGroup           = 0,
    kQ3FileReadObjectsInGroup       = 1 << 0,
    kQ3FileCurrentlyInsideGroup     = 1 << 1
} TQ3FileReadGroupStateMasks;
typedef unsigned long       TQ3FileReadGroupState;
```

Unknown Object Data Structures

QuickDraw 3D returns data about unknown text or binary data objects in an **unknown text data structure** or an **unknown binary data structure**. An unknown text data structure is defined by the `TQ3UnknownTextData` data type.

```
typedef struct TQ3UnknownTextData {
    char                        *objectName;        /*'\0' terminated*/
    char                        *contents;          /*'\0' terminated*/
} TQ3UnknownTextData;
```

Field descriptions

`objectName`	A pointer to the name of the unknown text object. This name is a C string terminated by the null character ('\0').
`contents`	A pointer to the contents of the unknown text object. This string is a C string terminated by the null character ('\0').

An unknown binary data structure is defined by the `TQ3UnknownBinaryData` data type.

```
typedef struct TQ3UnknownBinaryData {
    TQ3ObjectType                       objectType;
    unsigned long                       size;
    TQ3Endian                           byteOrder;
    char                                *contents;
} TQ3UnknownBinaryData;
```

Field descriptions

`objectType`	The type of the data in the unknown binary object.
`size`	The size, in bytes, of the data in the unknown binary object.
`byteOrder`	The order in which the bytes in a word are addressed. This field must contain `kQ3EndianBig` or `kQ3EndianLittle`.
`contents`	A pointer to a copy of the data of the unknown binary object.

File Objects Routines

This section describes routines you can use to create and manage file objects.

Creating File Objects

QuickDraw 3D provides a routine that you can use to create a file object.

Q3File_New

You can use the `Q3File_New` function to create a new file object.

```
TQ3FileObject Q3File_New (void);
```

DESCRIPTION

The `Q3File_New` function returns, as its function result, a new file object. If `Q3File_New` cannot create a new file object, it returns the value `NULL`.

ERRORS

`kQ3ErrorOutOfMemory`

Attaching File Objects to Storage Objects

To read data from or write data to a file object, you must first attach the file object to a storage object. QuickDraw 3D provides routines you can use to get and set the current storage object for a file object.

Q3File_GetStorage

You can use the `Q3File_GetStorage` function to get the current storage object for a file object.

```
TQ3Status Q3File_GetStorage (
                    TQ3FileObject file,
                    TQ3StorageObject *storage);
```

`file` A file object.

`storage` On exit, the storage object currently attached to the specified file object.

DESCRIPTION

The `Q3File_GetStorage` function returns, in the `storage` parameter, the storage object currently attached to the file object specified by the `file` parameter.

ERRORS

```
kQ3ErrorInvalidObject
kQ3ErrorNULLParameter
```

Q3File_SetStorage

You can use the `Q3File_SetStorage` function to set the storage object for a file object.

```
TQ3Status Q3File_SetStorage (
                    TQ3FileObject file,
                    TQ3StorageObject storage);
```

`file` A file object.

`storage` A storage object, or `NULL`.

DESCRIPTION

The `Q3File_SetStorage` function attaches the file object specified by the `file` parameter to the storage object specified by the `storage` parameter. The reference count of the storage object is incremented. You can pass the value `NULL` in the `storage` parameter to clear a file object's storage.

You cannot attach the same storage object to more than one file object.

ERRORS

```
kQ3ErrorFileAlreadyOpen
kQ3ErrorInvalidObject
kQ3ErrorStorageInUse
```

Accessing File Objects

QuickDraw 3D provides routines that you can use to open file objects, access information about them, and read and write their data.

Q3File_OpenRead

You can use the `Q3File_OpenRead` function to open a file object for reading.

```
TQ3Status Q3File_OpenRead (
                    TQ3FileObject file,
                    TQ3FileMode *mode);
```

`file` A file object.

`mode` On exit, a set of bit flags that specify the file mode of the specified file object. Set this field to `NULL` if you do not want a file mode to be returned.

DESCRIPTION

The `Q3File_OpenRead` function opens for reading the file object specified by the `file` parameter and returns, in the `mode` parameter, the file mode of the file object. See "File Mode Flags" (page 1029) for a description of the available file mode flags.

ERRORS

```
kQ3ErrorOSError
kQ3ErrorOutOfMemory
```

Q3File_OpenWrite

You can use the `Q3File_OpenWrite` function to open a file object for writing.

```
TQ3Status Q3File_OpenWrite (
                    TQ3FileObject file,
                    TQ3FileMode mode);
```

`file` A file object.

`mode` On exit, a set of bit flags that specify the file mode of the specified file object. Set this field to `NULL` if you do not want a file mode to be returned.

DESCRIPTION

The `Q3File_OpenWrite` function opens for writing the file object specified by the `file` parameter and returns the file mode of the file object in the `mode` parameter. See "File Mode Flags" (page 1029) for a description of the available file mode flags.

ERRORS

```
kQ3ErrorOSError
kQ3ErrorOutOfMemory
```

Q3File_IsOpen

You can use the `Q3File_IsOpen` function to determine whether a file object is open.

```
TQ3Status Q3File_IsOpen (TQ3FileObject file, TQ3Boolean *isOpen);
```

`file`	A file object.
`isOpen`	On exit, a Boolean value that indicates whether the specified file is open (`kQ3True`) or closed (`kQ3False`).

DESCRIPTION

The `Q3File_IsOpen` function returns, in the `isOpen` parameter, a Boolean value that indicates whether the file object specified by the `file` parameter is open (`kQ3True`) or closed (`kQ3False`).

ERRORS

```
kQ3ErrorFileNotOpen
kQ3ErrorInvalidObjectParameter
kQ3ErrorNULLParameter
```

Q3File_Close

You can use the `Q3File_Close` function to close a file object.

```
TQ3Status Q3File_Close (TQ3FileObject file);
```

`file` A file object.

DESCRIPTION

The `Q3File_Close` function closes the file object specified by the `file` parameter. `Q3File_Close` flushes any caches associated with the file and releases that memory for other uses. You should close a file object only when all operations on the file have completed successfully and you no longer need to keep the file object open.

ERRORS

```
kQ3ErrorFileInUse
kQ3ErrorInvalidObjectParameter
kQ3ErrorOSError
```

Q3File_Cancel

You can use the `Q3File_Cancel` function to cancel a file object.

```
TQ3Status Q3File_Cancel (TQ3FileObject file);
```

`file` A file object.

DESCRIPTION

The `Q3File_Cancel` function removes from memory any data associated with the file object specified by the `file` parameter and disposes of the file object itself. You should call `Q3File_Cancel` when some fatal error occurs in your application or simply when you're finished using a file object. Once the file object has been canceled, you can no longer read data from it or write data to it. In all likelihood, the file object is corrupt after you call the `Q3File_Cancel` function.

ERRORS

```
kQ3ErrorInvalidObjectParameter
kQ3ErrorOSError
```

Q3File_GetMode

You can use the `Q3File_GetMode` function to determine an open file object's current file mode.

```
TQ3Status Q3File_GetMode (
                    TQ3FileObject file,
                    TQ3FileMode *mode);
```

`file` A file object. This file object must be open.

`mode` On exit, the current file mode of the specified file object.

DESCRIPTION

The `Q3File_GetMode` function returns, in the `mode` parameter, a set of flags that encodes the current file mode of the file object specified by the `file` parameter. See "File Mode Flags" (page 1029) for a complete description of the available file mode flags.

ERRORS

```
kQ3ErrorFileNotOpen
kQ3ErrorInvalidObjectParameter
kQ3ErrorNULLParameter
```

Q3File_GetVersion

You can use the `Q3File_GetVersion` function to get the version of an open file object.

```
TQ3Status Q3File_GetVersion (
                    TQ3FileObject file,
                    TQ3FileVersion *version);
```

`file` A file object.

`version` On entry, a pointer to a file version. On exit, the current version of the specified file object.

DESCRIPTION

The `Q3File_GetVersion` function returns, through the `version` parameter, the current version of the file object specified by the `file` parameter.

ERRORS

```
kQ3ErrorFileNotOpen
kQ3ErrorInvalidObjectParameter
kQ3ErrorNULLParameter
```

Accessing Objects Directly

QuickDraw 3D provides low-level routines that you can use to find and manipulate objects in a file by reading sequentially through all the objects in it.

Q3File_GetNextObjectType

You can use the `Q3File_GetNextObjectType` function to get the type of the next object in a file.

```
TQ3ObjectType Q3File_GetNextObjectType (TQ3FileObject file);
```

`file` A file object.

DESCRIPTION

The `Q3File_GetNextObjectType` function returns, as its function result, the type of the next object in the file object specified by the `file` parameter. Depending

on the type of that object, you can then call `Q3File_ReadObject` to read it or `Q3File_SkipObject` to skip it.

If an error occurs, `Q3File_GetNextObjectType` returns the value `kQ3ObjectTypeInvalid`.

Q3File_IsNextObjectOfType

You can use the `Q3File_IsNextObjectOfType` function to determine whether the next object in a file is of a certain type.

```
TQ3Boolean Q3File_IsNextObjectOfType (
                    TQ3FileObject file,
                    TQ3ObjectType ofType);
```

`file` A file object.

`ofType` An object type.

DESCRIPTION

The `Q3File_IsNextObjectOfType` function returns, as its function result, a Boolean value that indicates whether the next object in the file object specified by the `file` parameter is of the type specified by the `ofType` parameter (`kQ3True`) or not (`kQ3False`).

Q3File_ReadObject

You can use the `Q3File_ReadObject` function to read the next object in a file.

```
TQ3Object Q3File_ReadObject (TQ3FileObject file);
```

`file` A file object.

DESCRIPTION

The `Q3File_ReadObject` function returns, as its function result, the next object in the file specified by the `file` parameter. If an error occurs, `Q3File_ReadObject` returns the value `NULL`.

Q3File_SkipObject

You can use the `Q3File_SkipObject` function to skip over an object in a file.

```
TQ3Status Q3File_SkipObject (TQ3FileObject file);
```

`file` A file object.

DESCRIPTION

The `Q3File_SkipObject` function skips the next object in the file object specified by the `file` parameter. Note that `Q3File_SkipObject` skips the next object whether or not you have already called `Q3File_GetNextObjectType` to get information about that object's type.

Q3File_IsEndOfFile

You can use the `Q3File_IsEndOfFile` function to determine whether the file position of a file object is at the end of the file.

```
TQ3Boolean Q3File_IsEndOfFile (TQ3FileObject file);
```

`file` A file object.

DESCRIPTION

The `Q3File_IsEndOfFile` function returns, as its function result, a Boolean value that indicates whether the current file position of the file object specified by the `file` parameter is at the end of the file (`kQ3True`) or not (`kQ3False`).

ERRORS

```
kQ3ErrorFileNotOpen
kQ3ErrorInvalidObjectParameter
kQ3ErrorNULLParameter
```

Setting Idle Methods

QuickDraw 3D provides a function that you can use to set a file object's idle method. QuickDraw 3D executes your idle method occasionally during lengthy file operations. See "Application-Defined Routines" (page 1095) for information on writing an idle method.

Q3File_SetIdleMethod

You can use the `Q3File_SetIdleMethod` function to set a file object's idle method.

```
TQ3Status Q3File_SetIdleMethod (
                    TQ3FileObject file,
                    TQ3FileIdleMethod idle,
                    const void *idleData);
```

`file` A file object.

`idle` A pointer to an idle method. See page 1096 for information on idle methods.

`idlerData` A pointer to an application-defined block of data. This pointer is passed to the idler callback routine when it is executed.

DESCRIPTION

The `Q3File_SetIdleMethod` function sets the idle method of the file object specified by the `file` parameter to the function specified by the `idle` parameter. The `idlerData` parameter is passed to your idle method whenever it is executed.

Reading and Writing File Subobjects

QuickDraw 3D provides functions that you can use to read QuickDraw 3D objects that are subobjects of custom objects. In general, you should call these

functions only within your custom read data method. For additional routines you can use, see "Custom File Object Routines," beginning on page 1086.

Q3File_IsEndOfData

You can use the `Q3File_IsEndOfData` function to determine whether there is more data for your application to read.

```
TQ3Boolean Q3File_IsEndOfData (TQ3FileObject file);
```

`file` A file object.

DESCRIPTION

The `Q3File_IsEndOfData` function returns, as its function result, a Boolean value that indicates whether there is more data to be read from the file object specified by the `file` parameter (`kQ3True`) or not (`kQ3False`).

SPECIAL CONSIDERATIONS

You should call this function only within a custom read data method.

Q3File_IsEndOfContainer

You can use the `Q3File_IsEndOfContainer` function to determine whether there are more subobjects of a custom object for your application to read.

```
TQ3Boolean Q3File_IsEndOfContainer (
                    TQ3FileObject file,
                    TQ3Object rootObject);
```

`file` A file object.

`rootObject` A root object in the specified file object.

DESCRIPTION

The `Q3File_IsEndOfContainer` function returns, as its function result, a Boolean value that indicates whether more subobjects remain to be read from a custom object specified by the `rootObject` parameter in the file object specified by the `file` parameter (`kQ3True`) or not (`kQ3False`).

SPECIAL CONSIDERATIONS

You should call this function only within a custom read data method.

Reading and Writing File Data

QuickDraw 3D provides routines that you can use to access custom data in a file object. In all cases, the reading or writing occurs at the current file position, and the file position is advanced if the read or write operation completes successfully.

IMPORTANT

You should call the `_Read` functions only in a custom read data method (of type `kQ3MethodTypeObjectReadData`), and you should call the `_Write` functions only in a custom write method (of type `kQ3MethodTypeObjectWrite`). ▲

These functions can read and write data in either text or binary files.

Q3Uns8_Read

You can use the `Q3Uns8_Read` function to read an unsigned 8-byte value from a file object.

```
TQ3Status Q3Uns8_Read (TQ3Uns8 *data, TQ3FileObject file);
```

`data`	On entry, a pointer to a block of memory large enough to hold an unsigned 8-byte value.
`file`	A file object.

DESCRIPTION

The `Q3Uns8_Read` function returns, in the block of memory pointed to by the `data` parameter, the unsigned 8-byte value read from the current position in the file object specified by the `file` parameter.

Q3Uns8_Write

You can use the `Q3Uns8_Write` function to write an unsigned 8-byte value to a file object.

```
TQ3Status Q3Uns8_Write (const TQ3Uns8 data, TQ3FileObject file);
```

`data` A pointer to an unsigned 8-byte value.

`file` A file object.

DESCRIPTION

The `Q3Uns8_Write` function writes the unsigned 8-byte value pointed to by the `data` parameter to the file object specified by the `file` parameter.

Q3Int8_Read

You can use the `Q3Int8_Read` function to read an 8-byte integer value from a file object.

```
TQ3Status Q3Int8_Read (TQ3Int8 *data, TQ3FileObject file);
```

`data` On entry, a pointer to a block of memory large enough to hold an 8-byte integer value.

`file` A file object.

DESCRIPTION

The `Q3Int8_Read` function returns, in the block of memory pointed to by the `data` parameter, the signed 8-byte integer value read from the current position in the file object specified by the `file` parameter.

Q3Int8_Write

You can use the `Q3Int8_Write` function to write an 8-byte integer value to a file object.

```
TQ3Status Q3Int8_Write (const TQ3Int8 data, TQ3FileObject file);
```

`data` A pointer to an 8-byte integer value.

`file` A file object.

DESCRIPTION

The `Q3Int8_Write` function writes the signed 8-byte integer value pointed to by the `data` parameter to the file object specified by the `file` parameter.

Q3Uns16_Read

You can use the `Q3Uns16_Read` function to read an unsigned 16-byte value from a file object.

```
TQ3Status Q3Uns16_Read (TQ3Uns16 *data, TQ3FileObject file);
```

`data` On entry, a pointer to a block of memory large enough to hold an unsigned 16-byte value.

`file` A file object.

DESCRIPTION

The `Q3Uns16_Read` function returns, in the block of memory pointed to by the `data` parameter, the unsigned 16-byte value read from the current position in the file object specified by the `file` parameter.

Q3Uns16_Write

You can use the `Q3Uns16_Write` function to write an unsigned 16-byte value to a file object.

```
TQ3Status Q3Uns16_Write (const TQ3Uns16 data, TQ3FileObject file);
```

`data` A pointer to an unsigned 16-byte value.

`file` A file object.

DESCRIPTION

The `Q3Uns16_Write` function writes the unsigned 16-byte value pointed to by the `data` parameter to the file object specified by the `file` parameter.

Q3Int16_Read

You can use the `Q3Int16_Read` function to read a 16-byte integer value from a file object.

```
TQ3Status Q3Int16_Read (TQ3Int16 *data, TQ3FileObject file);
```

`data` On entry, a pointer to a block of memory large enough to hold a 16-byte integer value.

`file` A file object.

DESCRIPTION

The `Q3Int16_Read` function returns, in the block of memory pointed to by the `data` parameter, the signed 16-byte integer value read from the current position in the file object specified by the `file` parameter.

Q3Int16_Write

You can use the `Q3Int16_Write` function to write a 16-byte integer value to a file object.

```
TQ3Status Q3Int16_Write (const TQ3Int16 data, TQ3FileObject file);
```

`data` A pointer to a 16-byte integer value.

`file` A file object.

DESCRIPTION

The `Q3Int16_Write` function writes the signed 16-byte integer value pointed to by the `data` parameter to the file object specified by the `file` parameter.

Q3Uns32_Read

You can use the `Q3Uns32_Read` function to read an unsigned 32-byte value from a file object.

```
TQ3Status Q3Uns32_Read (TQ3Uns32 *data, TQ3FileObject file);
```

`data` On entry, a pointer to a block of memory large enough to hold an unsigned 32-byte value.

`file` A file object.

DESCRIPTION

The `Q3Uns32_Read` function returns, in the block of memory pointed to by the `data` parameter, the unsigned 32-byte value read from the current position in the file object specified by the `file` parameter.

Q3Uns32_Write

You can use the `Q3Uns32_Write` function to write an unsigned 32-byte value to a file object.

```
TQ3Status Q3Uns32_Write (const TQ3Uns32 data, TQ3FileObject file);
```

`data` A pointer to an unsigned 32-byte value.

`file` A file object.

DESCRIPTION

The `Q3Uns32_Write` function writes the unsigned 32-byte value pointed to by the `data` parameter to the file object specified by the `file` parameter.

Q3Int32_Read

You can use the `Q3Int32_Read` function to read a signed 32-byte value from a file object.

```
TQ3Status Q3Int32_Read (TQ3Int32 *data, TQ3FileObject file);
```

`data` On entry, a pointer to a block of memory large enough to hold a signed 32-byte value.

`file` A file object.

DESCRIPTION

The `Q3Int32_Read` function returns, in the block of memory pointed to by the `data` parameter, the signed 32-byte value read from the current position in the file object specified by the `file` parameter.

Q3Int32_Write

You can use the `Q3Int32_Write` function to write a signed 32-byte value to a file object.

```
TQ3Status Q3Int32_Write (const TQ3Int32 data, TQ3FileObject file);
```

`data` A pointer to a signed 32-byte value.

`file` A file object.

DESCRIPTION

The `Q3Int32_Write` function writes the signed 32-byte value pointed to by the `data` parameter to the file object specified by the `file` parameter.

Q3Uns64_Read

You can use the `Q3Uns64_Read` function to read an unsigned 64-byte value from a file object.

```
TQ3Status Q3Uns64_Read (TQ3Uns64 *data, TQ3FileObject file);
```

`data` On entry, a pointer to a block of memory large enough to hold an unsigned 64-byte value.

`file` A file object.

DESCRIPTION

The `Q3Uns64_Read` function returns, in the block of memory pointed to by the `data` parameter, the unsigned 64-byte value read from the current position in the file object specified by the `file` parameter.

Q3Uns64_Write

You can use the `Q3Uns64_Write` function to write an unsigned 64-byte value to a file object.

```
TQ3Status Q3Uns64_Write (const TQ3Uns64 data, TQ3FileObject file);
```

`data` A pointer to an unsigned 64-byte value.

`file` A file object.

DESCRIPTION

The `Q3Uns64_Write` function writes the unsigned 64-byte value pointed to by the `data` parameter to the file object specified by the `file` parameter.

Q3Int64_Read

You can use the `Q3Int64_Read` function to read a signed 64-byte value from a file object.

```
TQ3Status Q3Int64_Read (TQ3Int64 *data, TQ3FileObject file);
```

`data` On entry, a pointer to a block of memory large enough to hold a signed 64-byte value.

`file` A file object.

DESCRIPTION

The `Q3Int64_Read` function returns, in the block of memory pointed to by the `data` parameter, the signed 64-byte value read from the current position in the file object specified by the `file` parameter.

Q3Int64_Write

You can use the `Q3Int64_Write` function to write a signed 64-byte value to a file object.

```
TQ3Status Q3Int64_Write (const TQ3Int64 data, TQ3FileObject file);
```

`data`	A pointer to a signed 64-byte value.
`file`	A file object.

DESCRIPTION

The `Q3Int64_Write` function writes the signed 64-byte value pointed to by the `data` parameter to the file object specified by the `file` parameter.

Q3Float32_Read

You can use the `Q3Float32_Read` function to read a floating-point 32-byte value from a file object.

```
TQ3Status Q3Float32_Read (TQ3Float32 *data, TQ3FileObject file);
```

`data`	On entry, a pointer to a block of memory large enough to hold a floating-point 32-byte value.
`file`	A file object.

DESCRIPTION

The `Q3Float32_Read` function returns, in the block of memory pointed to by the `data` parameter, the floating-point 32-byte value read from the current position in the file object specified by the `file` parameter.

Q3Float32_Write

You can use the `Q3Float32_Write` function to write a floating-point 32-byte value to a file object.

```
TQ3Status Q3Float32_Write (
                    const TQ3Float32 data,
                    TQ3FileObject file);
```

`data` A pointer to a floating-point 32-byte value.

`file` A file object.

DESCRIPTION

The `Q3Float32_Write` function writes the floating-point 32-byte value pointed to by the `data` parameter to the file object specified by the `file` parameter.

Q3Float64_Read

You can use the `Q3Float64_Read` function to read a floating-point 64-byte value from a file object.

```
TQ3Status Q3Float64_Read (TQ3Float64 *data, TQ3FileObject file);
```

`data` On entry, a pointer to a block of memory large enough to hold a floating-point 64-byte value.

`file` A file object.

DESCRIPTION

The `Q3Float64_Read` function returns, in the block of memory pointed to by the `data` parameter, the floating-point 64-byte value read from the current position in the file object specified by the `file` parameter.

Q3Float64_Write

You can use the `Q3Float64_Write` function to write a floating-point 64-byte value to a file object.

```
TQ3Status Q3Float64_Write (
                    const TQ3Float64 data,
                    TQ3FileObject file);
```

`data` A pointer to a floating-point 64-byte value.

`file` A file object.

DESCRIPTION

The `Q3Float64_Write` function writes the floating-point 64-byte value pointed to by the `data` parameter to the file object specified by the `file` parameter.

Q3Size_Pad

You can use the `Q3Size_Pad` function to determine the number of bytes occupied by a longword-aligned block.

```
TQ3Size Q3Size_Pad (TQ3Size size);
```

`size` The size, in bytes, of an object or structure.

DESCRIPTION

The `Q3Size_Pad` function returns, as its function result, the number of bytes it would take to contain a longword-aligned block whose size, before alignment, is specified by the `size` parameter.

Q3String_Read

You can use the `Q3String_Read` function to read a string from a file object.

```
TQ3Status Q3String_Read (
                    char *data,
                    unsigned long *length,
                    TQ3FileObject file);
```

`data` On entry, a pointer to a buffer whose length is of size `kQ3StringMaximumLength`, or `NULL`. On exit, a pointer to the string read from the specified file object. If this parameter is set to `NULL` on entry, no string is read, but its length is returned in the `length` parameter.

`length` On exit, the number of characters actually copied into the specified buffer. If `data` is set to `NULL` on entry, this parameter returns the length of the string.

`file` A file object.

DESCRIPTION

The `Q3String_Read` function returns, in the `data` parameter, a pointer to the next string in the file object specified by the `file` parameter. The string data is 7-bit ASCII, with standard escape sequences for any special characters in the string. The `Q3String_Read` function also returns, in the `length` parameter, the length of the string.

Q3String_Write

You can use the `Q3String_Write` function to write a string to a file object.

```
TQ3Status Q3String_Write (const char *data, TQ3FileObject file);
```

`data` A pointer to a string.

`file` A file object.

DESCRIPTION

The `Q3String_Write` function writes the string data pointed to by the `data` parameter to the file object specified by the `file` parameter. The number of bytes written to the file object is equal to `Q3Size_Pad(strlen(data)+1)`.

Q3NewLine_Write

You can use the `Q3NewLine_Write` function to write a newline character to a text metafile.

```
TQ3Status Q3NewLine_Write (TQ3FileObject file);
```

`file` A file object.

DESCRIPTION

The `Q3NewLine_Write` function writes a newline character to the text file object specified by the `file` parameter. It writes nothing if the file is binary.

Q3RawData_Read

You can use the `Q3RawData_Read` function to read raw data from a file object.

```
TQ3Status Q3RawData_Read (
                    unsigned char *data,
                    unsigned long size,
                    TQ3FileObject file);
```

`data`	On entry, a pointer to a buffer whose length is of the specified size. On exit, a pointer to the raw data read from the specified file object.
`size`	On entry, the number of bytes of raw data to be read from the specified file object into the specified buffer. On exit, the number of bytes actually copied into the specified buffer.
`file`	A file object.

DESCRIPTION

The `Q3RawData_Read` function returns, in the `data` parameter, a pointer to the next `size` bytes of raw data in the file object specified by the `file` parameter.

Q3RawData_Write

You can use the `Q3RawData_Write` function to write raw data to a file object.

```
TQ3Status Q3RawData_Write (
                    const unsigned char *data,
                    unsigned long size,
                    TQ3FileObject file);
```

`data`	On entry, a pointer to a buffer of raw data whose length is of the specified size.
`size`	On entry, the number of bytes of raw data to be read from the specified buffer and written to the specified file object. On exit, the number of bytes actually written to the file object.
`file`	A file object.

DESCRIPTION

The `Q3RawData_Write` function writes the raw data pointed to by the `data` parameter to the file object specified by the `file` parameter. The number of bytes written to the file object is equal to `Q3Size_Pad(size)`. If the number of bytes written to the file object is greater than `size`, `Q3RawData_Write` pads the data to the nearest 4-byte boundary with 0's.

In text files, raw data is output in hexadecimal form.

Q3Point2D_Read

You can use the `Q3Point2D_Read` function to read a two-dimensional point from a file object.

```
TQ3Status Q3Point2D_Read (
                    TQ3Point2D *point2D,
                    TQ3FileObject file);
```

`point2D` On entry, a pointer to a block of memory large enough to hold a two-dimensional point.

`file` A file object.

DESCRIPTION

The `Q3Point2D_Read` function returns, in the block of memory pointed to by the `point2D` parameter, the two-dimensional point read from the current position in the file object specified by the `file` parameter.

Q3Point2D_Write

You can use the `Q3Point2D_Write` function to write a two-dimensional point to a file object.

```
TQ3Status Q3Point2D_Write (
                    const TQ3Point2D *point2D,
                    TQ3FileObject file);
```

`point2D` A pointer to a two-dimensional point.

`file` A file object.

DESCRIPTION

The `Q3Point2D_Write` function writes the two-dimensional point pointed to by the `point2D` parameter to the file object specified by the `file` parameter.

Q3Point3D_Read

You can use the `Q3Point3D_Read` function to read a three-dimensional point from a file object.

```
TQ3Status Q3Point3D_Read (
                    TQ3Point3D *point3D,
                    TQ3FileObject file);
```

`point3D` On entry, a pointer to a block of memory large enough to hold a three-dimensional point.

`file` A file object.

DESCRIPTION

The `Q3Point3D_Read` function returns, in the block of memory pointed to by the `point3D` parameter, the three-dimensional point read from the current position in the file object specified by the `file` parameter.

Q3Point3D_Write

You can use the `Q3Point3D_Write` function to write a three-dimensional point to a file object.

```
TQ3Status Q3Point3D_Write (
                    const TQ3Point3D *point3D,
                    TQ3FileObject file);
```

`point3D` A pointer to a three-dimensional point.

`file` A file object.

DESCRIPTION

The `Q3Point3D_Write` function writes the three-dimensional point pointed to by the `point3D` parameter to the file object specified by the `file` parameter.

Q3RationalPoint3D_Read

You can use the `Q3RationalPoint3D_Read` function to read a rational three-dimensional point from a file object.

```
TQ3Status Q3RationalPoint3D_Read (
                    TQ3RationalPoint3D *point3D,
                    TQ3FileObject file);
```

`point3D` On entry, a pointer to a block of memory large enough to hold a rational three-dimensional point.

`file` A file object.

DESCRIPTION

The `Q3RationalPoint3D_Read` function returns, in the block of memory pointed to by the `point3D` parameter, the rational three-dimensional point read from the current position in the file object specified by the `file` parameter.

Q3RationalPoint3D_Write

You can use the `Q3RationalPoint3D_Write` function to write a rational three-dimensional point to a file object.

```
TQ3Status Q3RationalPoint3D_Write (
                    const TQ3RationalPoint3D *point3D,
                    TQ3FileObject file);
```

`point3D` A pointer to a rational three-dimensional point.

`file` A file object.

DESCRIPTION

The `Q3RationalPoint3D_Write` function writes the rational three-dimensional point pointed to by the `point3D` parameter to the file object specified by the `file` parameter.

Q3RationalPoint4D_Read

You can use the `Q3RationalPoint4D_Read` function to read a rational four-dimensional point from a file object.

```
TQ3Status Q3RationalPoint4D_Read (
                    TQ3RationalPoint4D *point4D,
                    TQ3FileObject file);
```

`point4D` On entry, a pointer to a block of memory large enough to hold a rational four-dimensional point.

`file` A file object.

DESCRIPTION

The `Q3RationalPoint4D_Read` function returns, in the block of memory pointed to by the `point4D` parameter, the rational four-dimensional point read from the current position in the file object specified by the `file` parameter.

Q3RationalPoint4D_Write

You can use the `Q3RationalPoint4D_Write` function to write a rational four-dimensional point to a file object.

```
TQ3Status Q3RationalPoint4D_Write (
                    const TQ3RationalPoint4D *point4D,
                    TQ3FileObject file);
```

`point4D` A pointer to a rational four-dimensional point.

`file` A file object.

DESCRIPTION

The `Q3RationalPoint4D_Write` function writes the rational four-dimensional point pointed to by the `point4D` parameter to the file object specified by the `file` parameter.

Q3Vector2D_Read

You can use the `Q3Vector2D_Read` function to read a two-dimensional vector from a file object.

```
TQ3Status Q3Vector2D_Read (
                    TQ3Vector2D *vector2D,
                    TQ3FileObject file);
```

`vector2D` On entry, a pointer to a block of memory large enough to hold a two-dimensional vector.

`file` A file object.

DESCRIPTION

The `Q3Vector2D_Read` function returns, in the block of memory pointed to by the `vector2D` parameter, the two-dimensional vector read from the current position in the file object specified by the `file` parameter.

Q3Vector2D_Write

You can use the `Q3Vector2D_Write` function to write a two-dimensional vector to a file object.

```
TQ3Status Q3Vector2D_Write (
                    const TQ3Vector2D *vector2D,
                    TQ3FileObject file);
```

`vector2D` A pointer to a two-dimensional vector.

`file` A file object.

DESCRIPTION

The `Q3Vector2D_Write` function writes the two-dimensional vector pointed to by the `vector2D` parameter to the file object specified by the `file` parameter.

Q3Vector3D_Read

You can use the `Q3Vector3D_Read` function to read a three-dimensional vector from a file object.

```
TQ3Status Q3Vector3D_Read (
                    TQ3Vector3D *vector3D,
                    TQ3FileObject file);
```

`vector3D` On entry, a pointer to a block of memory large enough to hold a three-dimensional vector.

`file` A file object.

DESCRIPTION

The `Q3Vector3D_Read` function returns, in the block of memory pointed to by the `vector3D` parameter, the three-dimensional vector read from the current position in the file object specified by the `file` parameter.

Q3Vector3D_Write

You can use the `Q3Vector3D_Write` function to write a three-dimensional vector to a file object.

```
TQ3Status Q3Vector3D_Write (
                    const TQ3Vector3D *vector3D,
                    TQ3FileObject file);
```

`vector3D` A pointer to a three-dimensional vector.

`file` A file object.

DESCRIPTION

The `Q3Vector3D_Write` function writes the three-dimensional vector pointed to by the `vector3D` parameter to the file object specified by the `file` parameter.

Q3Matrix4x4_Read

You can use the `Q3Matrix4x4_Read` function to read a 4-by-4 matrix from a file object.

```
TQ3Status Q3Matrix4x4_Read (
                    TQ3Matrix4x4 *matrix4x4,
                    TQ3FileObject file);
```

`matrix4x4` On entry, a pointer to a block of memory large enough to hold a 4-by-4 matrix.

`file` A file object.

DESCRIPTION

The `Q3Matrix4x4_Read` function returns, in the block of memory pointed to by the `matrix4x4` parameter, the 4-by-4 matrix read from the current position in the file object specified by the `file` parameter.

Q3Matrix4x4_Write

You can use the `Q3Matrix4x4_Write` function to write a 4-by-4 matrix to a file object.

```
TQ3Status Q3Matrix4x4_Write (
                    const TQ3Matrix4x4 *matrix4x4,
                    TQ3FileObject file);
```

`matrix4x4` A pointer to a 4-by-4 matrix.

`file` A file object.

DESCRIPTION

The `Q3Matrix4x4_Write` function writes the 4-by-4 matrix pointed to by the `matrix4x4` parameter to the file object specified by the `file` parameter.

Q3Tangent2D_Read

You can use the `Q3Tangent2D_Read` function to read a two-dimensional tangent from a file object.

```
TQ3Status Q3Tangent2D_Read (
                    TQ3Tangent2D *tangent2D,
                    TQ3FileObject file);
```

`tangent2D` On entry, a pointer to a block of memory large enough to hold a two-dimensional tangent.

`file` A file object.

DESCRIPTION

The `Q3Tangent2D_Read` function returns, in the block of memory pointed to by the `tangent2D` parameter, the two-dimensional tangent read from the current position in the file object specified by the `file` parameter.

Q3Tangent2D_Write

You can use the `Q3Tangent2D_Write` function to write a two-dimensional tangent to a file object.

```
TQ3Status Q3Tangent2D_Write (
                    const TQ3Tangent2D *tangent2D,
                    TQ3FileObject file);
```

`tangent2D` A pointer to a two-dimensional tangent.

`file` A file object.

DESCRIPTION

The `Q3Tangent2D_Write` function writes the two-dimensional tangent pointed to by the `tangent2D` parameter to the file object specified by the `file` parameter.

Q3Tangent3D_Read

You can use the `Q3Tangent3D_Read` function to read a three-dimensional tangent from a file object.

```
TQ3Status Q3Tangent3D_Read (
                    TQ3Tangent3D *tangent3D,
                    TQ3FileObject file);
```

`tangent3D` On entry, a pointer to a block of memory large enough to hold a three-dimensional tangent.

`file` A file object.

DESCRIPTION

The `Q3Tangent3D_Read` function returns, in the block of memory pointed to by the `tangent3D` parameter, the three-dimensional tangent read from the current position in the file object specified by the `file` parameter.

Q3Tangent3D_Write

You can use the `Q3Tangent3D_Write` function to write a three-dimensional tangent to a file object.

```
TQ3Status Q3Tangent3D_Write (
                    const TQ3Tangent3D *tangent3D,
                    TQ3FileObject file);
```

`tangent3D` A pointer to a three-dimensional tangent.

`file` A file object.

DESCRIPTION

The `Q3Tangent3D_Write` function writes the three-dimensional tangent pointed to by the `tangent3D` parameter to the file object specified by the `file` parameter.

Q3Comment_Write

You can use the `Q3Comment_Write` function to write a comment to a file object.

```
TQ3Status Q3Comment_Write (
                    char *comment,
                    TQ3FileObject file);
```

`comment` A pointer to a null-terminated C string.

`file` A file object.

DESCRIPTION

The `Q3Comment_Write` function writes the string of characters pointed to by the `comment` parameter to the file object specified by the `file` parameter. QuickDraw 3D currently supports writing comments to text files only; if you call `Q3Comment_Write` to write a comment to a binary file, QuickDraw 3D ignores the call. In addition, you cannot currently use QuickDraw 3D to read comments from a file.

Managing Unknown Objects

QuickDraw 3D creates an unknown object when it encounters an unrecognized type of object while reading a metafile. Your application might know how to handle objects of that type, so QuickDraw 3D provides routines that you can use to get the type and contents of an unknown object.

Note
You cannot explicitly create an unknown object. ◆

Q3Unknown_GetType

You can use the `Q3Unknown_GetType` function to get the type of an unknown object.

```
TQ3ObjectType Q3Unknown_GetType (TQ3UnknownObject unknownObject);
```

`unknownObject`
An unknown object.

DESCRIPTION

The `Q3Unknown_GetType` function returns, as its function result, the type of the unknown object specified by the `unknownObject` parameter. If successful, `Q3Unknown_GetType` returns one of these constants:

```
kQ3UnknownTypeBinary
kQ3UnknownTypeText
```

If the type cannot be determined or is invalid, `Q3Unknown_GetType` returns the value `kQ3ObjectTypeInvalid`.

Q3Unknown_GetDirtyState

You can use the `Q3Unknown_GetDirtyState` function to get the current dirty state of an unknown object.

```
TQ3Status Q3Unknown_GetDirtyState (
                    TQ3UnknownObject unknownObject,
                    TQ3Boolean *isDirty);
```

`unknownObject`
An unknown object.

`isDirty` On exit, a Boolean value that indicates whether the specified unknown object is dirty (`kQ3True`) or not (`kQ3False`).

DESCRIPTION

The `Q3Unknown_GetDirtyState` function returns, in the `isDirty` parameter, the current dirty state of the unknown object specified by the `unknownObject` parameter. The **dirty state** of an unknown object is a Boolean value that indicates whether an unknown object is preserved in its original state (`kQ3False`) or should be updated when written back to the file object from which it was originally read (`kQ3True`).

An unknown object is marked as dirty when it's first read into memory. You can mark the object as not dirty (by calling `Q3Unknown_SetDirtyState`) if you know that no state or contextual information has changed in the object. The application that generated the unknown data is responsible for either discarding any dirty data or attempting to preserve it.

Q3Unknown_SetDirtyState

You can use the `Q3Unknown_SetDirtyState` function to set the dirty state of an unknown object.

```
TQ3Status Q3Unknown_SetDirtyState (
                    TQ3UnknownObject unknownObject,
                    TQ3Boolean isDirty);
```

`unknownObject`
An unknown object.

`isDirty` A Boolean value that indicates whether the specified unknown object is dirty (`kQ3True`) or not (`kQ3False`).

DESCRIPTION

The `Q3Unknown_SetDirtyState` function sets the dirty state of the unknown object specified by the `unknownObject` parameter to the Boolean value passed in the `isDirty` parameter.

Q3UnknownText_GetData

You can use the `Q3UnknownText_GetData` function to get the data of an unknown text object.

```
TQ3Status Q3UnknownText_GetData (
                    TQ3UnknownObject unknownObject,
                    TQ3UnknownTextData *unknownTextData);
```

`unknownObject`
An unknown text object.

`unknownTextData`
A pointer to an unknown text data structure.

DESCRIPTION

The `Q3UnknownText_GetData` function returns, in the `objectName` and `contents` fields of the unknown text data structure pointed to by the `unknownTextData` parameter, pointers to the name and contents of an unknown text object (that is, an unknown object of type `kQ3UnknownTypeText`) specified by the `unknownObject` parameter. The `contents` field of the unknown text data structure points to the data stored in the text metafile, excluding any excess white space and any delimiter characters (that is, outermost parentheses).

Your application is responsible for allocating the memory occupied by the `unknownTextData` parameter. `Q3UnknownText_GetData` allocates memory to hold the name and contents pointed to by the fields of that structure. You must make certain to call `Q3UnknownText_EmptyData` to release the memory allocated by `Q3UnknownText_GetData` when you are finished using the data.

Q3UnknownText_EmptyData

You can use the `Q3UnknownText_EmptyData` function to dispose of the memory allocated by a previous call to `Q3UnknownText_GetData`.

```
TQ3Status Q3UnknownText_EmptyData (
                    TQ3UnknownTextData *unknownTextData);
```

`unknownTextData`
A pointer to an unknown text data structure that was filled in by a previous call to `Q3UnknownText_GetData`.

DESCRIPTION

The `Q3UnknownText_EmptyData` function deallocates the memory pointed to by the fields of the `unknownTextData` parameter. If successful, `Q3UnknownText_EmptyData` sets those fields to the value `NULL`.

Q3UnknownBinary_GetData

You can use the `Q3UnknownBinary_GetData` function to get the data of an unknown binary object.

```
TQ3Status Q3UnknownBinary_GetData (
                    TQ3UnknownObject unknownObject,
                    TQ3UnknownBinaryData *unknownBinaryData);
```

`unknownObject`
An unknown binary object.

`unknownBinaryData`
A pointer to an unknown binary data structure.

DESCRIPTION

The `Q3UnknownBinary_GetData` function returns, in the `contents` field of the unknown binary data structure pointed to by the `unknownBinaryData` parameter, a pointer to a copy of the contents of the unknown binary object (that is, an unknown object of type `kQ3UnknownTypeBinary`) specified by the `unknownObject` parameter. `Q3UnknownBinary_GetData` also returns, in the `objectType` and `size` fields of the unknown binary data structure, the type of the unknown binary object and the size, in bytes, of the data pointed to by the `contents` field.

Your application is responsible for allocating the memory occupied by the `unknownBinaryData` parameter. `Q3UnknownBinary_GetData` allocates memory to hold the data pointed to by the `contents` field of that structure. You must make

certain to call `Q3UnknownBinary_EmptyData` to release the memory allocated by `Q3UnknownBinary_GetData` when you are finished using the data.

Q3UnknownBinary_EmptyData

You can use the `Q3UnknownBinary_EmptyData` function to dispose of the memory allocated by a previous call to `Q3UnknownBinary_GetData`.

```
TQ3Status Q3UnknownBinary_EmptyData (
                    TQ3UnknownBinaryData *unknownBinaryData);
```

`unknownBinaryData`
A pointer to an unknown binary data structure that was filled in by a previous call to `Q3UnknownBinary_GetData`.

DESCRIPTION

The `Q3UnknownBinary_EmptyData` function deallocates the memory pointed to by the `contents` field of the `unknownBinaryData` parameter. If successful, `Q3UnknownBinary_EmptyData` sets that field to the value `NULL`. It also sets the `objectType` and `size` fields to default values.

Q3UnknownBinary_GetTypeString

You can use the `Q3UnknownBinary_GetTypeString` function to get the type string of an unknown binary object.

```
TQ3Status Q3UnknownBinary_GetTypeString (
                    TQ3UnknownObject unknownObject,
                    char **typeString);
```

`unknownObject`
An unknown binary object.

`typeString`
A handle to the type string of an unknown binary data structure.

DESCRIPTION

The `Q3UnknownBinary_GetTypeString` function returns a handle to the type string of an unknown binary object.

Your application is responsible for allocating the memory occupied by the `typeString` parameter. You must call `Q3UnknownBinary_EmptyTypeString` to release the memory allocated by `Q3UnknownBinary_GetTypeString` when you are finished using the data.

Q3UnknownBinary_EmptyTypeString

You can use the `Q3UnknownBinary_EmptyTypeString` function to dispose of the memory allocated by a previous call to `Q3UnknownBinary_GetTypeString`.

```
TQ3Status Q3UnknownBinary_EmptyTypeString (
                    char **typeString);
```

`typeString` A handle to the type string of an unknown binary data structure.

DESCRIPTION

The `Q3UnknownBinary_EmptyTypeString` function deallocates the memory used by a previous call to `Q3UnknownBinary_GetTypeString`.

Managing View Hints Objects

QuickDraw 3D provides routines that you can use to create and manage view hints objects. A view hints object is an object in a metafile that gives hints about how to render a scene. You can use that information to configure a view object, or you can choose to ignore it.

A view hints object contains specific information, derived from a view object and stored in a 3DMF file, that is separate from the group model submitted with a view. The view hints object is created from an existing view object using the `Q3ViewHints_New` call and should be written out at the beginning of the 3DMF file, followed by the group model for a scene. When an application reads a 3DMF file it should check for view hints and set up the view with the view hints settings if it wishes to preserve a scene's appearance between applications.

View hints may include instructions about the renderer, camera, lights, and view attributes, plus other information in the draw context such as the window's dimensions, mask state, mask bitmap, and clear image color. The version 1.5.1 QuickDraw 3D Viewer source code uses the following view hints information and applies it to the model's view if a view hints object is found while reading a 3DMF file:

- light group
- camera
- renderer
- window dimensions
- clear image color.

Q3ViewHints_New

You can use the `Q3ViewHints_New` function to create a new view hints object.

```
TQ3ViewHintsObject Q3ViewHints_New (TQ3ViewObject view);
```

`view` A view.

DESCRIPTION

The `Q3ViewHints_New` function returns, as its function result, a new view hints object that incorporates the view configuration information of the view object specified by the `view` parameter.

Q3ViewHints_GetRenderer

You can use the `Q3ViewHints_GetRenderer` function to get the renderer associated with a view hints object.

```
TQ3Status Q3ViewHints_GetRenderer (
                    TQ3ViewHintsObject viewHints,
                    TQ3RendererObject *renderer);
```

viewHints	A view hints object.
renderer	On exit, the renderer currently associated with the specified view hints object.

DESCRIPTION

The Q3ViewHints_GetRenderer function returns, in the renderer parameter, the renderer currently associated with the view hints object specified by the viewHints parameter. The reference count of that renderer is incremented.

Q3ViewHints_SetRenderer

You can use the Q3ViewHints_SetRenderer function to set the renderer associated with a view hints object.

```
TQ3Status Q3ViewHints_SetRenderer (
                    TQ3ViewHintsObject viewHints,
                    TQ3RendererObject renderer);
```

viewHints	A view hints object.
renderer	A renderer object.

DESCRIPTION

The Q3ViewHints_SetRenderer function attaches the renderer specified by the renderer parameter to the view hints object specified by the viewHints parameter. The reference count of the specified renderer is incremented. In addition, if some other renderer was already attached to the specified view hints object, the reference count of that renderer is decremented.

Q3ViewHints_GetCamera

You can use the Q3ViewHints_GetCamera function to get the camera associated with a view hints object.

```
TQ3Status Q3ViewHints_GetCamera (
                    TQ3ViewHintsObject viewHints,
                    TQ3CameraObject *camera);
```

`viewHints` A view hints object.

`camera` On exit, the camera object currently associated with the specified view hints object.

DESCRIPTION

The `Q3ViewHints_GetCamera` function returns, in the `camera` parameter, the camera currently associated with the view hints object specified by the `viewHints` parameter. The reference count of that camera is incremented.

Q3ViewHints_SetCamera

You can use the `Q3ViewHints_SetCamera` function to set the camera associated with a view hints object.

```
TQ3Status Q3ViewHints_SetCamera (
                    TQ3ViewHintsObject viewHints,
                    TQ3CameraObject camera);
```

`viewHints` A view hints object.

`camera` A camera object.

DESCRIPTION

The `Q3ViewHints_SetCamera` function attaches the camera specified by the `camera` parameter to the view hints object specified by the `viewHints` parameter. The reference count of the specified camera is incremented. In addition, if some other camera was already attached to the specified view hints object, the reference count of that camera is decremented.

Q3ViewHints_GetLightGroup

You can use the `Q3ViewHints_GetLightGroup` function to get the light group associated with a view hints object.

```
TQ3Status Q3ViewHints_GetLightGroup (
                    TQ3ViewHintsObject viewHints,
                    TQ3GroupObject *lightGroup);
```

`viewHints` A view hints object.

`lightGroup` On exit, the light group currently associated with the specified view hints object.

DESCRIPTION

The `Q3ViewHints_GetLightGroup` function returns, in the `lightGroup` parameter, the light group currently associated with the view hints object specified by the `viewHints` parameter. The reference count of that light group is incremented.

Q3ViewHints_SetLightGroup

You can use the `Q3ViewHints_SetLightGroup` function to set the light group associated with a view hints object.

```
TQ3Status Q3ViewHints_SetLightGroup (
                    TQ3ViewHintsObject viewHints,
                    TQ3GroupObject lightGroup);
```

`viewHints` A view hints object.

`lightGroup` A light group.

DESCRIPTION

The `Q3ViewHints_SetLightGroup` function attaches the light group specified by the `lightGroup` parameter to the view hints object specified by the `viewHints` parameter. The reference count of the specified light group is incremented. In

addition, if some other light group was already attached to the specified view hints object, the reference count of that light group is decremented.

Q3ViewHints_GetAttributeSet

You can use the `Q3ViewHints_GetAttributeSet` function to get the current attribute set associated with a view hints object.

```
TQ3Status Q3ViewHints_GetAttributeSet (
                    TQ3ViewHintsObject viewHints,
                    TQ3AttributeSet *attributeSet);
```

`viewHints` A view hints object.

`attributeSet` On exit, the attribute set currently associated with the specified view hints object.

DESCRIPTION

The `Q3ViewHints_GetAttributeSet` function returns, in the `attributeSet` parameter, the current attribute set of the view hints object specified by the `viewHints` parameter. The reference count of the attribute set is incremented.

Q3ViewHints_SetAttributeSet

You can use the `Q3ViewHints_SetAttributeSet` function to set the attribute set associated with a view hints object.

```
TQ3Status Q3ViewHints_SetAttributeSet (
                    TQ3ViewHintsObject viewHints,
                    TQ3AttributeSet attributeSet);
```

`viewHints` A view hints object.

`attributeSet` An attribute set.

DESCRIPTION

The `Q3ViewHints_SetAttributeSet` function attaches the attribute set specified by the `attributeSet` parameter to the view hints object specified by the `viewHints` parameter. The reference count of the specified attribute set is incremented. In addition, if some other attribute set was already attached to the specified view hints object, the reference count of that attribute set is decremented.

Q3ViewHints_GetDimensionsState

You can use the `Q3ViewHints_GetDimensionsState` function to get the dimension state associated with a view hints object.

```
TQ3Status Q3ViewHints_GetDimensionsState (
                    TQ3ViewHintsObject viewHints,
                    TQ3Boolean *isValid);
```

`viewHints` A view hints object.

`isValid` On exit, the current dimension state of the specified view hints object.

DESCRIPTION

The `Q3ViewHints_GetDimensionsState` function returns, in the `isValid` parameter, a Boolean value that indicates whether the dimensions in the view hints object specified by the `viewHints` parameter are to be used (`kQ3True`) or not (`kQ3False`).

Q3ViewHints_SetDimensionsState

You can use the `Q3ViewHints_SetDimensionsState` function to set the dimension state associated with a view hints object.

```
TQ3Status Q3ViewHints_SetDimensionsState (
                    TQ3ViewHintsObject viewHints,
                    TQ3Boolean isValid);
```

viewHints	A view hints object.
isValid	A dimension state.

DESCRIPTION

The Q3ViewHints_SetDimensionsState function sets the dimension state of the view hints object specified by the viewHints parameter to the value passed in the isValid parameter.

Q3ViewHints_GetDimensions

You can use the Q3ViewHints_GetDimensions function to get the dimensions associated with a view hints object.

```
TQ3Status Q3ViewHints_GetDimensions (
                    TQ3ViewHintsObject viewHints,
                    unsigned long *width,
                    unsigned long *height);
```

viewHints	A view hints object.
width	On exit, the width of the specified view hints object.
height	On exit, the height of the specified view hints object.

DESCRIPTION

The Q3ViewHints_GetDimensions function returns, in the width and height parameters, the current width and height associated with the view hints object specified by the viewHints parameter.

Q3ViewHints_SetDimensions

You can use the Q3ViewHints_SetDimensions function to set the dimensions associated with a view hints object.

```
TQ3Status Q3ViewHints_SetDimensions (
                    TQ3ViewHintsObject viewHints,
                    unsigned long width,
                    unsigned long height);
```

`viewHints` A view hints object.

`width` The desired width of the view hints object.

`height` The desired height of the view hints object.

DESCRIPTION

The `Q3ViewHints_SetDimensions` function sets the width and height of the view hints object specified by the `viewHints` parameter to the values passed in the `width` and `height` parameters.

Q3ViewHints_GetMaskState

You can use the `Q3ViewHints_GetMaskState` function to get the mask state associated with a view hints object.

```
TQ3Status Q3ViewHints_GetMaskState (
                    TQ3ViewHintsObject viewHints,
                    TQ3Boolean *isValid);
```

`viewHints` A view hints object.

`isValid` On exit, the current mask state of the specified view hints object.

DESCRIPTION

The `Q3ViewHints_GetMaskState` function returns, in the `isValid` parameter, a Boolean value that determines whether the mask associated with the view hints object specified by the `viewHints` parameter is to be used (`kQ3True`) or not (`kQ3False`).

Q3ViewHints_SetMaskState

You can use the `Q3ViewHints_SetMaskState` function to set the mask state associated with a view hints object.

```
TQ3Status Q3ViewHints_SetMaskState (
                    TQ3ViewHintsObject viewHints,
                    TQ3Boolean isValid);
```

`viewHints` A view hints object.

`isValid` The desired mask state of the specified view hints object.

DESCRIPTION

The `Q3ViewHints_SetMaskState` function sets the mask state of the view hints object specified by the `viewHints` parameter to the value specified in the `isValid` parameter. Set `isValid` to `kQ3True` if you want the mask enabled and to `kQ3False` otherwise.

Q3ViewHints_GetMask

You can use the `Q3ViewHints_GetMask` function to get the mask associated with a view hints object.

```
TQ3Status Q3ViewHints_GetMask (
                    TQ3ViewHintsObject viewHints,
                    TQ3Bitmap *mask);
```

`viewHints` A view hints object.

`mask` On exit, the mask of the specified view hints object.

DESCRIPTION

The `Q3ViewHints_GetMask` function returns, in the `mask` parameter, the current mask for the view hints object specified by the `viewHints` parameter. The mask is a bitmap whose bits determine whether or not corresponding pixels in the drawing destination are drawn or are masked out. `Q3ViewHints_GetMask`

allocates memory internally for the returned bitmap; when you're done using the bitmap, you should call the `Q3Bitmap_Empty` function to dispose of that memory.

Q3ViewHints_SetMask

You can use the `Q3ViewHints_SetMask` function to set the mask associated with a view hints object.

```
TQ3Status Q3ViewHints_SetMask (
                    TQ3ViewHintsObject viewHints,
                    const TQ3Bitmap *mask);
```

`viewHints` A view hints object.

`mask` The desired mask of the specified view hints object.

DESCRIPTION

The `Q3ViewHints_SetMask` function sets the mask of the view hints object specified by the `viewHints` parameter to the bitmap specified in the `mask` parameter. `Q3ViewHints_SetMask` copies the bitmap to internal QuickDraw 3D memory, so you can dispose of the specified bitmap after calling `Q3ViewHints_SetMask`.

Q3ViewHints_GetClearImageMethod

You can use the `Q3ViewHints_GetClearImageMethod` function to get the image clearing method associated with a view hints object.

```
TQ3Status Q3ViewHints_GetClearImageMethod (
                    TQ3ViewHintsObject viewHints,
                    TQ3DrawContextClearImageMethod *clearMethod);
```

`viewHints` A view hints object.

`clearMethod` On exit, the current image clearing method of the specified view hints object. See "Draw Context Data Structure" (page 843) for the values that can be returned in this parameter.

DESCRIPTION

The `Q3ViewHints_GetClearImageMethod` function returns, in the `clearMethod` parameter, a constant that indicates the current image clearing method for the view hints object specified by the `viewHints` parameter.

Q3ViewHints_SetClearImageMethod

You can use the `Q3ViewHints_SetClearImageMethod` function to set the image clearing method associated with a view hints object.

```
TQ3Status Q3ViewHints_SetClearImageMethod (
                    TQ3ViewHintsObject viewHints,
                    TQ3DrawContextClearImageMethod clearMethod);
```

`viewHints` A view hints object.

`clearMethod` The desired image clearing method of the specified view hints object. See "Draw Context Data Structure" (page 843) for the values that can be passed in this parameter.

DESCRIPTION

The `Q3ViewHints_SetClearImageMethod` function sets the image clearing method of the view hints object specified by the `viewHints` parameter to the value specified in the `clearMethod` parameter.

Q3ViewHints_GetClearImageColor

You can use the `Q3ViewHints_GetClearImageColor` function to get the image clearing color associated with a view hints object.

```
TQ3Status Q3ViewHints_GetClearImageColor (
                    TQ3ViewHintsObject viewHints,
                    TQ3ColorARGB *color);
```

`viewHints` A view hints object.

`color` On exit, the current image clearing color of the specified view hints object.

DESCRIPTION

The `Q3ViewHints_GetClearImageColor` function returns, in the `color` parameter, a constant that indicates the current image clearing color for the view hints object specified by the `viewHints` parameter.

Q3ViewHints_SetClearImageColor

You can use the `Q3ViewHints_SetClearImageColor` function to set the image clearing color associated with a view hints object.

```
TQ3Status Q3ViewHints_SetClearImageColor (
                    TQ3ViewHintsObject viewHints,
                    const TQ3ColorARGB *color);
```

`viewHints` A view hints object.

`color` The desired image clearing color of the specified view hints object.

DESCRIPTION

The `Q3ViewHints_SetClearImageColor` function sets the image clearing color of the view hints object specified by the `viewHints` parameter to the value specified in the `color` parameter.

Custom File Object Routines

This section describes routines you can use to manage custom file objects.

Marking and Getting External References

QuickDraw 3D supplies routines you can use to manage external references from one metafile to another.

Q3File_MarkAsExternalReference

You can use the `Q3File_MarkAsExternalReference` function to mark an object in a metafile as being shared with another metafile.

```
TQ3Status  Q3File_MarkAsExternalReference(
                    TQ3FileObject file,
                    TQ3SharedObject sharedObject);
```

`file` A metafile object.

`sharedObject` An object in the metafile that is shared with another metafile.

DESCRIPTION

The `Q3File_MarkAsExternalReference` function marks the object `sharedObject` in the metafile object `file` as an object that will always be written out as an external reference. Whenever a `submit` call is made on the object in write loop, an external reference object will be written out that specifies the location of the object in `file`.

Q3File_GetExternalReferences

You can use the `Q3File_GetExternalReferences` function to obtain the names of files externally referred to by a metafile.

```
TQ3GroupObject  Q3File_GetExternalReferences(
                    TQ3FileObject file);
```

`file` A metafile object.

DESCRIPTION

The `Q3File_GetExternalReferences` function returns the names of the files that are externally referred to by a metafile that contains external references. It returns a group that contains one `Q3String` object for each external reference object in the metafile. The `Q3String` object contains the name (in general, the pathname) of the file. Since one `Q3String` object is produced for each external reference, it is possible for the same name to appear in several `Q3String` objects. If no files are externally referred to, the call returns `NULL`.

Group Reading Modes

QuickDraw 3D provides routines that let you control how group objects are read.

Q3File_SetReadInGroup

You can use the `Q3File_SetReadInGroup` function to set the mode by which objects in a group are read.

```
TQ3Status  Q3File_SetReadInGroup(
                    TQ3FileObject file,
                    TQ3FileReadGroupState readGroupState);
```

`file` A metafile object.

`readGroupState`
The mode of reading objects in a group; see "Group Reading States" (page 1032).

DESCRIPTION

The `Q3File_SetReadInGroup` function sets the group reading mode for the metafile object `file` to one of the two following values, based on the value of `readGroupState`. The default value for `readGroupState` is `kQ3FileReadWholeGroup`.

`kQ3FileReadWholeGroup`
A group is read as a single object; a single call to `Q3File_ReadObject` reads the group and everything in it.

`kQ3FileReadObjectsInGroup`
Each object inside the group is read individually. In this case, the first call to `Q3File_ReadObject` reads the group itself and returns an empty group. Each subsequent `Q3File_ReadObject` call reads one more object. The last `Q3File_ReadObject` call returns an `EndGroup` object; this signals that the end of the group has been read.

So long as the `TQ3FileReadGroupState` has most recently been set to `kQ3FileReadObjectsInGroup`, every group that's encountered will be read this way.

You can set `kQ3FileReadObjectsInGroup`, read some groups, then set the state back to `kQ3FileReadWholeGroup`. When this has been done, then every new group will be read in as with `kQ3FileReadWholeGroup`, but groups currently being read will finish as with `kQ3FileReadObjectsInGroup`. Because groups can be nested, it is possible to be inside of many groups, all of which would be completed as with `kQ3FileReadObjectsInGroup`.

Q3File_GetReadInGroup

You can use the `Q3File_GetReadInGroup` function to set the mode by which objects in a group are read.

```
TQ3Status Q3File_GetReadInGroup(
                    TQ3FileObject file,
                    TQ3FileReadGroupState *readGroupState);
```

`file` A metafile object.

`readGroupState`
A mask for the mode of reading objects in a group; see "Group Reading States" (page 1032).

DESCRIPTION

The `Q3File_SetReadInGroup` function gets a mask for the group reading mode of the metafile object `file` and returns it in `readGroupState`. In addition to either of

the values described in "Q3File_SetReadInGroup" (`kQ3FileReadWholeGroup` or `kQ3FileReadObjectsInGroup`), `readGroupState` can have this value:

`kQ3FileCurrentlyInsideGroup`
: This value is OR-combined into the mask if the reading process is currently inside a group. This value is compatible with the next group being read with either `kQ3FileReadWholeGroup` or `kQ3FileReadObjectsInGroup`.

Writing to Custom File Objects

Writing to custom file objects is done in two stages: the traversal stage, where the data to be written is set up, and the actual writing stage.

Traversal is done by the custom object's `TQ3XObjectTraverseMethod` or `TQ3XObjectTraverseDataMethod`:

```
typedef TQ3Status ( QD3D_CALLBACK
                    *TQ3XObjectTraverseMethod)(
                    TQ3Object object,
                    void *data,
                    TQ3ViewObject view);
```

Writing is done by the custom object's `TQ3XObjectWriteMethod`:

```
typedefTQ3Status(QD3D_CALLBACK *TQ3XObjectWriteMethod)(
                    const void *object,
                    TQ3FileObject file);
```

The first part of the custom object's `TQ3XObjectTraverseMethod` traverses the root object. A metafile object always has a root object, which may or may not have one or more subobjects. The root object consists of all data that is not itself a QD3D object. All data in the form of QD3D objects must appear in the metafile as subobjects. For example, in a box the geometrical data (origin, orientation, and axes) makes up the root object. But the attribute sets (both face attribute sets and box attribute sets) are themselves QD3D objects, so they must be subobjects. If an object has subobjects, then the root and the subobjects are all contained in a container. If there are no subobjects, then no container is necessary.

The custom object's `TQ3XObjectTraverseMethod` computes the size of the root object and then calls `Q3XView_SubmitWriteData` once to traverse the root. After

that, it can submit the subobjects, if any. It does this by making the public API call `Q3Object_Submit` on each subobject. As a shortcut, it can call the `Q3XView_SubmitSubObjectData` function.

If you need data from the view that's passed to the `Q3View_StartWriting` call that initiates the write loop, you must obtain it during your traverse routine (which is passed this view as argument). You cannot obtain data from the view during your write routine, since it does not take a view as argument and there is no other way to access the view from within it.

Note that your `TQ3XObjectTraverseMethod` can check some condition and, based on the result, decide not to write a particular part of the memory accessible from the `data` parameter in `Q3XView_SubmitWriteData` as part of the root object. It does this by not adding bytes to the `size` parameter. This decision must be mirrored in your `TQ3XObjectWriteMethod` method, where the corresponding `Q3..._Write` calls must be bypassed. The `TQ3XObjectTraverseMethod` can decide not to make any particular `Q3Object_Submit` call on a subobject, but this doesn't require any mirroring in the `TQ3XObjectWriteMethod` because `Q3Object_Submit` in the traverse method is all that's needed for writing a subobject. There is one special case: your traverse method can decide to write nothing at all by simply returning `kQ3Success` immediately. In that case the write method will never get called, so it doesn't need to do any checking of conditions.

Your `TQ3XObjectWriteMethod` consists of making any of the following calls (and no others) to write out the data of its root object. These calls are described in "Reading and Writing File Data," beginning on page 1045.

```
Q3Uns8_Write
Q3Uns16_Write
Q3Uns32_Write
Q3Int32_Write
Q3Uns64_Write
Q3Float32_Write
Q3Float64_Write
Q3Point2D_Write
Q3Point3D_Write
Q3RationalPoint3D_Write
Q3RationalPoint4D_Write
Q3Vector2D_Write
Q3Vector3D_Write
Q3Matrix4x4_Write
Q3Tangent2D_Write
Q3Tangent3D_Write
```

```
Q3NewLine_Write
Q3String_Write
Q3Size_Pad
Q3RawData_Write
Q3Comment_Write
```

The `TQ3XObjectWriteMethod` does nothing with subobjects; their roots are written by their own `TQ3XObjectWriteMethod`.

Q3XView_SubmitWriteData

You can use the `Q3XView_SubmitWriteData` function to write data to a custom file object.

```
TQ3Status Q3XView_SubmitWriteData(
                          TQ3ViewObject view,
                          TQ3Size size,
                          void *data,
                          TQ3XDataDeleteMethod deleteData);
```

`view`	A view.
`size`	The size of the data actually written.
`data`	A pointer to memory containing the data to be written.
`deleteData`	A `TQ3XDataDeleteMethod` method.

DESCRIPTION

The `Q3XView_SubmitWriteData` function writes the data pointed to by `data`, of size `size`, to the `view` view object. The `deleteData` parameter designates a method that disposes of memory allocations upon completion.

It is important that the `size` parameter matches the size of the data actually written. If this is not so, `Q3XView_SubmitWriteData` will fail.

The `data` pointer is later passed to your `TQ3XObjectWriteMethod`. Typically, it will point to a data structure, and your write routine will contain various calls from the family `Q3Uns8_Write`, `Q3Uns32_Write`, etc (see “Reading and Writing File

Data," beginning on page 1045), which will write to various fields in that data structure.

The `deleteData` parameter designates a `TQ3XDataDeleteMethod`. It is passed a pointer to your data structure, and it will delete whatever needs to be deleted (dispose of QD3D objects, deallocate memory, and so on). The delete method will be called upon exit of your write method whether or not your write method succeeded.

Q3XView_SubmitSubObjectData

You can use the `Q3XView_SubmitSubObjectData` function to write data to a custom file object more efficiently than you can with `Q3XView_SubmitWriteData`.

```
TQ3Status Q3XView_SubmitSubObjectData(
                    TQ3ViewObject view,
                    TQ3XObjectClass objectClass,
                    TQ3Size size,
                    void *data,
                    TQ3XDataDeleteMethod deleteData);
```

`view`	A view.
`size`	The size of the data actually written.
`objectClass`	An object class.
`data`	A pointer to memory containing the data to be written.
`deleteData`	A `TQ3XDataDeleteMethod` method.

DESCRIPTION

The `Q3XView_SubmitSubObjectData` function is a shortcut alternative to the `Q3XView_SubmitWriteData` function. It writes the data pointed to by `data`, of size `size`, to the `view` view object. The `deleteData` parameter designates a method that disposes of memory allocations upon completion.

You can use the `Q3XView_SubmitSubObjectData` function in the following situation. Suppose that your custom object C1 has another object of class S1 as a subobject in its metafile, and the only purpose for the existence of class S1 is to enable the writing and

reading of these metafile subobjects. On the writing side, you don't need to create the object. You can just to do what's needed in a traverse method (pass its size and a pointer to the data), using `Q3XView_SubmitSubObjectData`. (`Q3XView_SubmitSubObjectData` also takes a `TQ3XObjectClass` as a parameter; this parameter is implicit in `Q3XView_SubmitWriteData`, where it is assumed to be the class of the root.) The traversal routine for C1 can call `Q3XView_SubmitSubObjectData`, where the data is C1's data structure, instead of having to first create an object of class S1 and then call `Q3Object_Submit` on that object. The write method for class S1 remains as it is with `Q3XView_SubmitWriteData`.

Edit Tracking

The EditTracking feature is designed to allow users to keep track of whether an object that contains custom elements or attributes (including unknown elements or attributes) has been edited.

Here's an example of applying this feature. ObjA is created and written out by application AppA, which registers custom attribute type CusAttA. ObjA contains some attributes of type CusAttA. Assume that ObjA is a geometry and attributes of type CusAttA are per-vertex attributes that depend on the overall geometry of the object. Suppose now that ObjA is read by AppB, which does not register attribute type CusAttA. This means that when AppB reads in ObjA, every attribute of type CusAttA is read in as an unknown object. Suppose that AppB edits the geometry of ObjA, resulting in object ObjB. Since AppB doesn't know about attributes of type CusAttA, it cannot edit them so that they conform properly to the edited geometry. This means that in ObjB, geometry and attributes of type CusAttA may be incompatible. To track this possibility, when AppB writes ObjB into a metafile the EditTracking feature will add a special subobject of type `edited` to ObjB.

Suppose now that the user is back working in AppA and reads in ObjB. AppA recognizes attributes of type CusAttA, so they will be read in as known and will have their effect during rendering. During reading AppA also reads in the `edited` subobject. This automatically marks object ObjB in such a way that the call `Q3Shared_GetEditTrackingState` made on ObjB returns 1, signifying that the object contains a custom attribute (or element) which was edited after having been read in. In this case, there is a risk of incorrect rendering. If this condition is not satisfied, `Q3Shared_GetEditTrackingState` returns 0.

Q3Shared_GetEditTrackingState

You can use the `Q3Shared_GetEditTrackingState` function to determine if an object contains a custom attribute or element that was edited after having been read in.

```
unsigned long Q3Shared_GetEditTrackingState(
                    TQ3SharedObject sharedObject);
```

`sharedObject` An object shared by two or more metafiles.

DESCRIPTION

The `Q3Shared_GetEditTrackingState` function returns 1 if object `sharedObject` contains a custom attribute or element that was edited after having been read in; otherwise it returns 0.

Q3Shared_ClearEditTracking

You can use the `Q3Shared_ClearEditTracking` function to clear the marker read by `Q3Shared_GetEditTrackingState`.

```
TQ3Status Q3Shared_ClearEditTracking(
                    TQ3SharedObject sharedObject);
```

`sharedObject` An object shared by two or more metafiles.

DESCRIPTION

You can call the `Q3Shared_ClearEditTracking` function on an object if you wish to guarantee that an `edited` subobject will not be written to the metafile or that an object that has been read in will return 0 from `Q3Shared_GetEditTrackingState`.

Application-Defined Routines

This section describes the I/O methods you can implement to handle a custom object type. Your custom methods are reported to QuickDraw 3D by your object

metahandler. This section also describes how to write a file idler callback routine.

Note
For information about defining an object metahandler and about the basic methods for handling custom objects, see the chapter "QuickDraw 3D Objects." ◆

These I/O methods define how QuickDraw 3D handles your custom objects when reading them from or writing them to a metafile. Each distinct object in a metafile consists of a root object that determines the object's type and default data. Some types of objects can have child objects attached to them, which add information to the parent object or override the parent's default data. A parent object and its child (or children) are encapsulated in a container, the first object in which is always the parent object.

To read a custom object from a file, you need to define a read data method for the custom object. To write a custom object to a file, you need to define two I/O methods for the custom object: a traversal method and a write method.

TQ3FileIdleMethod

You can define an idle method to receive occasional callbacks to your application during lengthy file operations.

```
typedef TQ3Status (*TQ3FileIdleMethod) (
                    TQ3FileObject file,
                    const void *idlerData);
```

`file` A file object.

`idlerData` A pointer to an application-defined block of data.

DESCRIPTION

Your `TQ3FileIdleMethod` function is called occasionally during lengthy file operations. You can use an idle method to provide a method for the user to cancel the lengthy operation (for example, by clicking a button or pressing a key sequence such as Command-period).

If your idle method returns `kQ3Success`, QuickDraw 3D continues its current operation. If your idle method returns `kQ3Failure`, QuickDraw 3D cancels its current operation and returns `kQ3ViewStatusCancelled` the next time you call `Q3View_EndWriting`.

There is currently no way to indicate how often you want your idle method to be called. You can read the time maintained by the Operating System if you need to determine the amount of time that has elapsed since your idle method was last called.

You must not call any QuickDraw 3D routines inside your idle method.

File System Errors, Warnings, and Notices

The following is a list of file system errors, warnings, and notices. A list of general QuickDraw 3D errors is given in "QuickDraw 3D Errors, Warnings, and Notices" (page 87).

```
kQ3ErrorNoStorageSetForFile
kQ3ErrorEndOfFile
kQ3ErrorFileCancelled
kQ3ErrorInvalidMetafile
kQ3ErrorInvalidMetafilePrimitive
kQ3ErrorInvalidMetafileLabel
kQ3ErrorInvalidMetafileObject
kQ3ErrorInvalidMetafileSubObject
kQ3ErrorInvalidSubObjectForObject
kQ3ErrorUnresolvableReference
kQ3ErrorUnknownObject
kQ3ErrorFileAlreadyOpen
kQ3ErrorFileNotOpen
kQ3ErrorFileIsOpen
kQ3ErrorBeginWriteAlreadyCalled
kQ3ErrorBeginWriteNotCalled
kQ3ErrorEndWriteNotCalled
kQ3ErrorReadStateInactive
kQ3ErrorStateUnavailable
kQ3ErrorWriteStateInactive
kQ3ErrorSizeNotLongAligned
kQ3ErrorFileModeRestriction
```

```
kQ3ErrorInvalidHexString
kQ3ErrorWroteMoreThanSize
kQ3ErrorWroteLessThanSize
kQ3ErrorReadLessThanSize
kQ3ErrorReadMoreThanSize
kQ3ErrorNoBeginGroup
kQ3ErrorSizeMismatch
kQ3ErrorStringExceedsMaximumLength
kQ3ErrorValueExceedsMaximumSize
kQ3ErrorNonUniqueLabel
kQ3ErrorEndOfContainer
kQ3ErrorUnmatchedEndGroup
kQ3ErrorFileVersionExists
kQ3ErrorBadStringType
kQ3WarningInvalidSubObjectForObject
kQ3WarningInvalidHexString
kQ3WarningUnknownObject
kQ3WarningInvalidMetafileObject
kQ3WarningUnmatchedBeginGroup
kQ3WarningUnmatchedEndGroup
kQ3WarningInvalidTableOfContents
kQ3WarningUnresolvableReference
kQ3WarningNoAttachMethod
kQ3WarningInconsistentData
kQ3WarningReadLessThanSize
kQ3WarningFilePointerResolutionFailed
kQ3WarningFilePointerRedefined
kQ3WarningStringExceedsMaximumLength
kQ3NoticeFileAliasWasChanged
kQ3NoticeFileCancelled
```

CHAPTER 18

Pointing Device Manager

This chapter describes the QuickDraw 3D Pointing Device Manager, a set of functions that you can use to manage three-dimensional pointing devices. By using this manager, you ensure that your application's users can interact with the three-dimensional objects modeled in your windows in a simple and natural manner, using the input devices that are available on their computers.

To use this chapter, you should already be familiar with creating and manipulating views, as described in the chapter "View Objects." If you are developing a 3D pointing device (which allows the user to control locations in three dimensions), you need to read the information on trackers and controllers in this chapter, as well as the information on writing device drivers in the book *Inside Macintosh: Devices*.

This chapter begins by describing controllers and trackers. Then it provides some sample code illustrating how to use the routines in the QuickDraw 3D Pointing Device Manager. The chapter ends with a complete reference for this manager.

About the Pointing Device Manager

The **QuickDraw 3D Pointing Device Manager** is a set of functions that you can use to manage three-dimensional pointing devices. The QuickDraw 3D Pointing Device Manager is the 3D analogue of some of the managers contained in the Macintosh Toolbox, which you can use to create and handle two-dimensional aspects of your application's user interface (such as windows, controls, and menus). The key benefit in both cases (that is, two- and three-dimensional) is the same: by using the routines supplied by Apple Computer, Inc., you can guarantee that your application looks and acts just like any other applications that use those routines.

This consistency of the user interface among different applications helps users learn to use your application; it also helps them focus on the distinctive features of your application, because they are not distracted by unnecessary differences between your application and other 3D applications they may have used.

IMPORTANT
In general, you should use the user interface routines contained in the QuickDraw 3D Pointing Device Manager for your three-dimensional user interface elements unless you have a compelling reason to adopt some other user interface paradigms. ▲

The QuickDraw 3D Pointing Device Manager contains several kinds of routines, including routines you can use to

- determine what kinds of pointing devices are available on a particular computer
- configure one or more of those devices to control items in a 3D model (such as the position of an object or a camera)

The following sections describe these tasks and the routines you can use to perform them.

Controllers

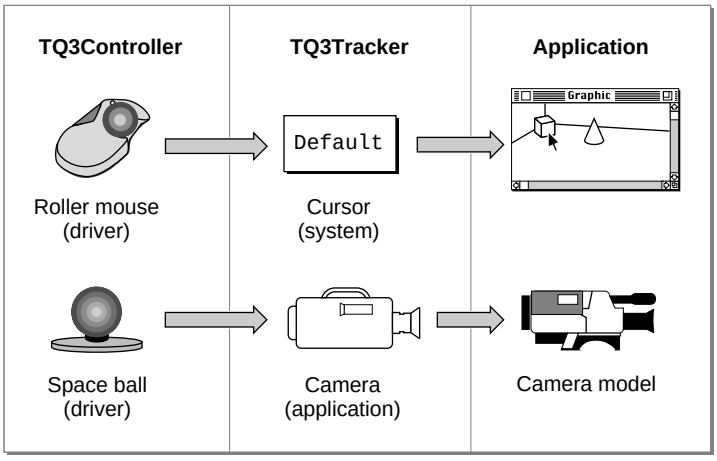
In order for a user to interact successfully with the objects in a three-dimensional model, it's necessary for the computer to provide some means of manipulating positions along three independent axes. Most existing computer systems support only two-dimensional input devices, such as mouse pointers or graphics tablets. QuickDraw 3D provides a standard interface between applications and devices that allows users to work with any available 3D pointing devices. In addition, the QuickDraw 3D Pointing Device Manager provides routines that you can use to determine what kinds of 3D pointing devices are available and to assign certain of them to specific uses in your application.

A **3D pointing device** is any physical device capable of controlling movements or specifying positions in three-dimensional space. QuickDraw 3D represents 3D pointing devices as **controller objects** (or, more briefly, **controllers**). A user can attach more than one 3D pointing device to a computer. Accordingly, QuickDraw 3D can support more than one controller at a time. When several 3D pointing devices are present, they can all contribute to the movement of a

single user interface element (such as the position of the selected object), or they can control different elements. For example, a particular 3D pointing device can be dedicated to controlling a view's camera, and another 3D pointing device can drive the position of the selected object.

The position and orientation of a single element in your application's user interface are represented by a **tracker object** (or, more briefly, a **tracker**). For instance, the position and orientation of a selected object are represented by a tracker, as is any other interface element you've assigned to some controller. Each controller can affect only one tracker, but a tracker can be affected by one or more controllers. Figure 18-1 illustrates a possible arrangement of devices, controllers, and trackers.

Figure 18-1 A sample configuration of input devices, controllers, and trackers



The controller object associated with a particular 3D pointing device is usually created by a device driver, the software that communicates with the device using whatever low-level protocols are appropriate for the device. The device can be connected to the computer through a serial port, via ADB connections, through an expansion card, or by other means. The device driver receives data from the device and passes it to the associated controller. As already indicated, a controller is associated with exactly one tracker. Changes in the position or orientation of the pointing device thereby result in changes in the position or orientation of the associated tracker.

IMPORTANT
By default, a controller contributes to the position of the system's cursor. You can, if you wish, reassign a particular controller to control the position or orientation of some other user interface element. ▲

All controllers are capable of controlling positions, and some controllers are capable of controlling orientations as well. Pointing devices contain one or more buttons; the associated controller must be capable of reading button states (up or down) from the pointing device and reporting those states to the tracker. Currently, QuickDraw 3D supports up to 32 buttons on a 3D pointing device. More generally, a pointing device may support additional input and output modes as well. For example, it's possible to construct a 3D pointing device that contains a number of dials and alphanumeric displays labeling those dials. The device's controller must then be able to communicate information about dials and labels between the device and an application using that device.

Any piece of information, beyond the standard position, orientation, and buttons, that the user sends to the application by means of an input device is called a **controller value.** Any piece of information sent from the application to the input device is called a **controller channel.** A dial position, for example, is a controller value, whereas an alphanumeric label generated by the application is a controller channel.

In general, your application does not need to communicate with controllers directly. As already indicated, controllers are almost always created by their associated device drivers, which read data from the devices and pass it to the associated controller. Moreover, a controller is by default connected to the cursor. Your application needs to access a controller only to assign it to some interface element other than the cursor or to read controller data other than position, orientation, and button states. To get information about other controller values, for instance, you need to call routines that query the controller directly.

QuickDraw 3D maintains a list of all the controllers that are available on a computer. A controller is identified by its signature, which is a string that uniquely identifies the manufacturer and model of the controller. You can search for a controller by signature by calling QuickDraw 3D Pointing Device Manager routines. Once a controller is added to the list of available controllers, it cannot be removed from it, but it can be made inactive. If for some reason a device becomes unavailable, the device driver should mark the controller as inactive. The device might later become available, in which case the driver can

reactivate the controller. You should always check that a controller is active before directly accessing a controller from the list of controllers.

Note
Because controllers may be shared by multiple applications, you cannot dispose of a controller. Instead, you can decommission the controller by calling `Q3Controller_Decommission`. Decommissioning a controller makes it inoperative for any application. ◆

Controllers are referenced by the `TQ3ControllerRef` type:

```
typedef void *TQ3ControllerRef;
```

Controller States

When your application is inactive, some other application might use a particular pointing device your application was using. That other application might also reset some of the controller channels. As a result, you need to keep track of the current controller state across the times your application is inactive. A **controller state object** (or, more briefly, a **controller state**) consists mainly of the current channels and other settings of a controller. When your application is about to be inactivated, you should call the function `Q3ControllerState_SaveAndReset` to save the current controller state. Then, when your application is reactivated, you should call `Q3ControllerState_Restore` to restore the proper controller state.

Trackers

A tracker is a kind of QuickDraw 3D object that controls the position, orientation, and button state of a specific element in your application's user interface. QuickDraw 3D always provides a tracker that controls the location and orientation of the system cursor. You can create additional trackers and attach them to other visible elements of your application's user interface. As suggested earlier, you can attach a 3D pointing device to a view's camera and then let users control the camera's position and orientation using the device. If the device has one or more buttons, you could let users turn the lights on and off using those buttons.

Note
This is not necessarily a good human interface for turning lights on and off; it is intended only for illustrative purposes. ◆

All the controllers currently reporting data to a particular tracker, whether absolute or relative, jointly contribute to the button states of the tracker. The button state of a tracker button of a particular index is the logical OR of the button states of all controller buttons of that index.

You can determine that a tracker has moved in one or both of two ways. You can poll for a **tracker serial number,** which changes every time the coordinates of the tracker are updated by a controller. Or, you can install a **tracker notify function** that is called whenever the coordinates of a tracker change by more than a specified amount (the **tracker thresholds**). Your tracker notify function can respond itself to the change, or it can just wake up your application. These two techniques can also be combined.

Using the QuickDraw 3D Pointing Device Manager

This section shows how to use some of the routines in the QuickDraw 3D Pointing Device Manager. In particular, it shows how to reassign a 3D pointing device to control a camera's position.

Controlling a Camera Position With a Pointing Device

By default, a 3D pointing device contributes to the position and orientation of the cursor. You can, however, reassign a particular pointing device so that it controls some other element in a user interface view, such as the position and orientation of the view's camera. To do this, you must first find the pointing device. Then you need to disconnect the device from the cursor and connect it to the desired user interface element.

Suppose that the pointing box you want to reassign is a knob box, which consists of a set of 12 knobs and associated alphanumeric displays. Six of the knobs control the standard position and orientation values, and the remaining 6 knobs are device-specific. Listing 18-1 shows first how to search for the knob box.

Listing 18-1 Searching for a particular 3D pointing device

```
TQ3ControllerRef        gBoxController              = NULL;
TQ3TrackerObject        gBoxTracker                 = NULL;
unsigned long           gBoxSerialNumber            = 0;

void MyFindKnobBox (void)
{
    TQ3ControllerRef        controller;
    char                    mySig[256];         /*controller signature*/
    char                    *boxSig =
                                "Knob Systems, Inc.::Knob Box Grandé";
    TQ3Boolean              isActive;

    /*Find the box controller.*/
    for (Q3Controller_Next(NULL, &controller); controller != NULL;
            Q3Controller_Next(controller, &controller)) {
        Q3Controller_GetSignature(controller, mySig, 256);
        Q3Controller_GetActivation(controller, &isActive);

        if (isActive && strncmp(mySig, boxSig, strlen(boxSig))

== 0)
            gBoxController = controller;
    }

    /*If we found a knob box, remember it.*/
    if (gBoxController != NULL) {
        gBoxTracker = Q3Tracker_New(MyBoxNotifyFunc);
        if (gBoxTracker != NULL) {
            Q3Tracker_SetNotifyThresholds(gBoxTracker, 0.05, 0.05);
        }
        Q3Controller_SetTracker(gBoxController, gBoxTracker);
    }
}
```

Once you've found a knob box, you must connect it to the camera, but only for as long as your application's window is active. When the window is inactive, the box should revert to its previous function. Listing 18-2 defines two functions you should call when your application becomes active or inactive.

Listing 18-2 Activating and deactivating a pointing device

```
void MyOnActivation (void)
{
    /*Any knob box data goes to your tracker.*/
    if (gBoxController != NULL)
        Q3Controller_SetTracker(gBoxController, gBoxTracker);
}

void MyOnDeactivation (void)
{
    /*Any knob box data goes to the default tracker.*/
    if (gBoxController != NULL)
        Q3Controller_SetTracker(gBoxController, NULL);
}
```

As long as the knob box is attached to a view's camera, your application receives notification of changes in the knob box through the notify function `MyBoxNotifyFunc`, defined in Listing 18-3. `MyBoxNotifyFunc` may be called at interrupt time. On Macintosh computers, you should wake up your process so that it can poll the tracker. This ensures that the application will recover control from the `WaitNextEvent` function.

Listing 18-3 Receiving notification of changes in a pointing device

```
TQ3Status MyBoxNotifyFunc (TQ3TrackerObject tracker,
                                    TQ3ControllerRef controller)
{
    MyOSWakeUpMyProcess();      /*wake up app; poll for data later*/
    return(kQ3Success);
}
```

The `MyPollKnobBox` function defined in Listing 18-4 shows how to poll for data from the device. Your application's idle procedure should call `MyPollKnobBox`.

Listing 18-4 Polling for data from a pointing device

```
void MyPollKnobBox (void)
{
    TQ3Boolean              changed;
    TQ3Point3D              position;
    TQ3Vector3D             delta;

    /*Get the current knob positions.*/
    changed = kQ3False;
    if (gBoxTracker != NULL) {
        Q3Tracker_GetPosition(gBoxTracker, &position, &delta,
                                        &changed, &gBoxSerialNumber);
    }

    /*Move camera and redraw if positions are new.*/
    if (changed) {
        MyComputeCameraFromKnobBox(&position, &orientation);
        MyRedrawScene();
    }
}
```

QuickDraw 3D Pointing Device Manager Reference

This section describes the QuickDraw 3D data structures and routines that you can use to manage controllers and controller states, trackers, and cursors.

Data Structures

This section describes the data structure that you use to create a new controller object. In general, only device drivers need to create controller objects.

Controller Data Structure

You use a **controller data structure** to specify information when creating a new controller object. A controller data structure is defined by the `TQ3ControllerData` data type.

```
typedef struct TQ3ControllerData {
    char                                *signature;
    unsigned long                       valueCount;
    unsigned long                       channelCount;
    TQ3ChannelGetMethod                 channelGetMethod;
    TQ3ChannelSetMethod                 channelSetMethod;
} TQ3ControllerData;
```

Field descriptions

`signature` The controller's signature. A signature is a null-terminated C string that uniquely identifies the manufacturer and model of a controller device. You are responsible for defining your controller's signature.

`valueCount` The number of values supported by the controller.

`channelCount` The number of channels supported by the controller. If the value in this field is greater than 0, you may define optional routines that get and set those channels.

`channelGetMethod` A pointer to a controller's channel-getting method. See page 1140 for information on this method. This field is valid only if the value in the `channelCount` field is greater than 0. You may, however, pass `NULL` in this field if the controller cannot report the current channels.

`channelSetMethod` A pointer to a controller's channel-setting method. See page 1141 for information on this method. This field is valid only if the value in the `channelCount` field is greater than 0. You may, however, pass `NULL` in this field if the controller cannot set the channels.

QuickDraw 3D Pointing Device Manager Routines

This section describes routines you can use to manage various aspects of your application's user interface or to create and manage controllers and trackers.

Creating and Managing Controllers

QuickDraw 3D provides routines that you can use to create and manipulate controller objects.

Note
Some of these functions are intended for use only by controller device drivers. You should not call those functions from within applications. ◆

Q3Controller_New

You can use the `Q3Controller_New` function to create a new controller.

```
TQ3ControllerRef Q3Controller_New (
                    const TQ3ControllerData *controllerData);
```

`controllerData`
A pointer to a controller data structure.

DESCRIPTION

The `Q3Controller_New` function returns, as its function result, a reference to a new controller object having the characteristics specified by the `controllerData` parameter. The new controller object is initially made active and is associated with the system cursor's tracker. You can call `Q3Controller_SetTracker` to associate the controller with some other tracker. The serial number of the new controller object is set to 1. If `Q3Controller_New` cannot create a new controller, it returns `NULL`.

You cannot delete a controller, but you can make it no longer operational. See the description of `Q3Controller_Decommission` (page 1111) for details.

SPECIAL CONSIDERATIONS

In general, you need to use this function only if you are writing a device driver for a controller.

SEE ALSO

See “Controller Data Structure” (page 1108) for a description of the fields of the controller data structure.

Q3Controller_GetListChanged

You can use the `Q3Controller_GetListChanged` function to determine whether the list of available controllers has changed.

```
TQ3Status Q3Controller_GetListChanged (
                    TQ3Boolean *listChanged,
                    unsigned long *serialNumber);
```

`listChanged` On exit, a Boolean value that indicates whether the list of available controllers has changed (`kQ3True`) or not (`kQ3False`).

`serialNumber` On entry, a serial number of the list of available controllers. On exit, the current serial number of that list.

DESCRIPTION

The `Q3Controller_GetListChanged` function returns, in the `listChanged` parameter, a Boolean value that indicates whether the list of available controllers has changed since the time the serial number passed in the `serialNumber` parameter was generated. If the list has changed, the new serial number is returned in the `serialNumber` parameter; otherwise, the `serialNumber` parameter is unchanged.

Q3Controller_Next

You can use the `Q3Controller_Next` function to read through the list of available controllers.

```
TQ3Status Q3Controller_Next (
                    TQ3ControllerRef controllerRef,
                    TQ3ControllerRef *nextControllerRef);
```

`controllerRef` A reference to a controller, or `NULL`.

`nextControllerRef`
On exit, a reference to the controller that immediately follows the specified controller. If the value in the `controllerRef` parameter is `NULL`, this parameter returns a reference to the first controller.

DESCRIPTION

The `Q3Controller_Next` function returns, in the `nextControllerRef` parameter, a reference to the controller that immediately follows the controller specified by the `controllerRef` parameter. To get the first controller in the list, pass the value `NULL` in the `controllerRef` parameter. If the controller specified by the `controllerRef` parameter is the last controller in the list, `nextControllerRef` is set to `NULL`.

Q3Controller_Decommission

You can use the `Q3Controller_Decommission` function to make a controller inactive.

```
TQ3Status Q3Controller_Decommission (TQ3ControllerRef controllerRef);
```

`controllerRef` A reference to a controller.

DESCRIPTION

The `Q3Controller_Decommission` function makes the controller specified by the `controllerRef` parameter inactive. Any remaining references to a controller that has been decommissioned are still valid, but the controller is no longer operational. (In other words, when the specified controller is referred to by an application or process other than the one that created it, reasonable default values are returned, not `kQ3Failure`.) Decommissioning a controller might cause the notify function of the tracker currently associated with the specified controller to be called.

SPECIAL CONSIDERATIONS

The `Q3Controller_Decommission` function should be called only by the application or process that created the specified controller.

Q3Controller_GetActivation

You can use the `Q3Controller_GetActivation` function to get the activation state of a controller.

```
TQ3Status Q3Controller_GetActivation (
                    TQ3ControllerRef controllerRef,
                    TQ3Boolean *active);
```

`controllerRef` A reference to a controller.

`active` On exit, a Boolean value that indicates whether the specified controller is active (`kQ3True`) or inactive (`kQ3False`).

DESCRIPTION

The `Q3Controller_GetActivation` function returns, in the `active` parameter, a Boolean value that indicates whether the controller specified by the `controllerRef` parameter is currently active or inactive.

Q3Controller_SetActivation

You can use the `Q3Controller_SetActivation` function to set the activation state of a controller.

```
TQ3Status Q3Controller_SetActivation (
                    TQ3ControllerRef controllerRef,
                    TQ3Boolean active);
```

`controllerRef` A reference to a controller.

`active` A Boolean value that indicates whether the specified controller is to be made active (`kQ3True`) or inactive (`kQ3False`).

DESCRIPTION

The `Q3Controller_SetActivation` function sets the activation state of the controller specified by the `controllerRef` parameter to the value specified in the `active` parameter. If the activation state of a controller is changed, the serial number of the list of available controllers is incremented. A controller should be inactive if it is temporarily off-line.

The notify function of the tracker currently associated with the specified controller might be called when `Q3Controller_SetActivation` is called.

SPECIAL CONSIDERATIONS

In general, you need to use this function only if you are writing a device driver for a controller.

Q3Controller_GetSignature

You can use the `Q3Controller_GetSignature` function to get the signature of a controller.

```
TQ3Status Q3Controller_GetSignature (
                    TQ3ControllerRef controllerRef,
                    char *signature,
                    unsigned long numChars);
```

`controllerRef` A reference to a controller.

`signature` On entry, a pointer to a buffer that is to be filled with the signature of the specified controller.

`numChars` On entry, the size of the buffer pointed to by the `signature` parameter.

DESCRIPTION

The `Q3Controller_GetSignature` function returns, through the `signature` parameter, the signature of the controller specified by the `controllerRef` parameter. You are responsible for allocating a buffer whose address is passed in the `signature` parameter and whose size is passed in the `numChars` parameter.

If the signature is larger than the specified size, the signature is truncated to fit in the buffer.

Q3Controller_GetChannel

You can use the `Q3Controller_GetChannel` function to get a controller channel.

```
TQ3Status Q3Controller_GetChannel (
                    TQ3ControllerRef controllerRef,
                    unsigned long channel,
                    void *data,
                    unsigned long *dataSize);
```

`controllerRef` A reference to a controller.

`channel` An index into the list of channels associated with the specified controller. This value is always greater than or equal to 0 and less than the channel count specified at the time `Q3Controller_New` was called.

`data` On exit, a pointer to the current value of the specified controller channel. The data type of the returned channel is controller-specific.

`dataSize` On entry, the number of bytes in the specified buffer. On exit, the number of bytes actually written to that buffer.

DESCRIPTION

The `Q3Controller_GetChannel` function returns, through the `data` parameter, the current controller channel specified by the `controllerRef` and `channel` parameters. You are responsible for allocating memory for the data buffer and passing the size of that buffer in the `dataSize` parameter. `Q3Controller_GetChannel` returns, in the `dataSize` parameter, the number of bytes written to the data buffer.

Q3Controller_SetChannel

You can use the `Q3Controller_SetChannel` function to set a controller channel.

```
TQ3Status Q3Controller_SetChannel (
                    TQ3ControllerRef controllerRef,
                    unsigned long channel,
                    const void *data,
                    unsigned long dataSize);
```

`controllerRef` A reference to a controller.

`channel` An index into the list of channels associated with the specified controller. This value is always greater than or equal to 0 and less than the channel count specified at the time `Q3Controller_New` was called.

`data` On entry, a pointer to a buffer that contains the desired value of the specified controller channel. The data type of the channel is controller-specific. If this field contains the value `NULL`, the specified channel is reset to a default or inactive value.

`dataSize` On entry, the number of bytes of data in the specified buffer.

DESCRIPTION

The `Q3Controller_SetChannel` function sets the controller channel specified by the `controllerRef` and `channel` parameters to the data whose address is passed in the `data` parameter. The `dataSize` parameter specifies the number of bytes in the `data` buffer.

Q3Controller_GetValueCount

You can use the `Q3Controller_GetValueCount` function to get the number of values of a controller.

```
TQ3Status Q3Controller_GetValueCount (
                    TQ3ControllerRef controllerRef,
                    unsigned long *valueCount);
```

`controllerRef` A reference to a controller.

`valueCount` On exit, the number of values supported by the specified controller.

DESCRIPTION

The `Q3Controller_GetValueCount` function returns, in the `valueCount` parameter, the number of values supported by the controller specified by the `controllerRef` parameter.

Q3Controller_SetTracker

You can use the `Q3Controller_SetTracker` function to set the tracker associated with a controller.

```
TQ3Status Q3Controller_SetTracker (
                    TQ3ControllerRef controllerRef,
                    TQ3TrackerObject tracker);
```

`controllerRef` A reference to a controller.

`tracker` A tracker object.

DESCRIPTION

The `Q3Controller_SetTracker` function associates the tracker specified by the `tracker` parameter with the controller specified by the `controllerRef` parameter. If the value of the `tracker` parameter is `NULL`, the controller is attached to the system cursor tracker. Changing a controller's tracker might cause the notify functions of both the previous tracker and the new tracker to be called.

Q3Controller_HasTracker

You can use the `Q3Controller_HasTracker` function to determine whether a controller is currently associated with a tracker.

```
TQ3Status Q3Controller_HasTracker (
                    TQ3ControllerRef controllerRef,
                    TQ3Boolean *hasTracker);
```

`controllerRef` A reference to a controller.

`hasTracker` On exit, a Boolean value that indicates whether the specified controller is currently associated with an active tracker (`kQ3True`) or not (`kQ3False`).

DESCRIPTION

The `Q3Controller_HasTracker` function returns, in the `hasTracker` parameter, a Boolean value that indicates whether the controller specified by the `controllerRef` parameter is active and is currently associated with an active tracker.

SPECIAL CONSIDERATIONS

In general, you need to use this function only if you are writing a device driver for a controller.

Q3Controller_Track2DCursor

You can use the `Q3Controller_Track2DCursor` function to determine whether a controller is currently affecting the two-dimensional system cursor.

```
TQ3Status Q3Controller_Track2DCursor (
                    TQ3ControllerRef controllerRef,
                    TQ3Boolean *track2DCursor);
```

`controllerRef` A reference to a controller.

`track2DCursor` On exit, a Boolean value that indicates whether the specified controller is currently affecting the two-dimensional system cursor (`kQ3True`) or not (`kQ3False`).

DESCRIPTION

The `Q3Controller_Track2DCursor` function returns, in the `track2DCursor` parameter, a Boolean value that indicates whether the controller specified by the `controllerRef` parameter is currently affecting the two-dimensional system cursor but the *z* axis values and orientation of the system cursor tracker are being ignored. If the specified controller is not attached to the system cursor tracker or if that controller is inactive, `track2DCursor` is set to `kQ3False`.

SPECIAL CONSIDERATIONS

In general, you need to use this function only if you are writing a device driver for a controller.

Q3Controller_Track3DCursor

You can use the `Q3Controller_Track3DCursor` function to determine whether a controller is currently affecting the depth information also being used with the system cursor.

```
TQ3Status Q3Controller_Track3DCursor (
                    TQ3ControllerRef controllerRef,
                    TQ3Boolean *track3DCursor);
```

`controllerRef` A reference to a controller.

`track3DCursor` On exit, a Boolean value that indicates whether the specified controller is currently affecting the system cursor and the depth is being used (`kQ3True`) or not (`kQ3False`).

DESCRIPTION

The `Q3Controller_Track3DCursor` function returns, in the `track3DCursor` parameter, a Boolean value that indicates whether the controller specified by the `controllerRef` parameter is currently affecting the two-dimensional system cursor and the *z* axis values and orientation of the system cursor tracker are not being ignored. If the specified controller is not attached to the system cursor tracker or if that controller is inactive, `track3DCursor` is set to `kQ3False`.

SPECIAL CONSIDERATIONS

In general, you need to use this function only if you are writing a device driver for a controller.

Q3Controller_GetButtons

You can use the `Q3Controller_GetButtons` function to get the button state of a controller.

```
TQ3Status Q3Controller_GetButtons (
                    TQ3ControllerRef controllerRef,
                    unsigned long *buttons);
```

`controllerRef` A reference to a controller.

`buttons` On exit, the current button state value of the specified controller.

DESCRIPTION

The `Q3Controller_GetButtons` function returns, in the `buttons` parameter, the current button state value of the controller specified by the `controllerRef` parameter.

Q3Controller_SetButtons

You can use the `Q3Controller_SetButtons` function to set the button state of a controller.

```
TQ3Status Q3Controller_SetButtons (
                    TQ3ControllerRef controllerRef,
                    unsigned long buttons);
```

`controllerRef` A reference to a controller.

`buttons` A button state value.

DESCRIPTION

The `Q3Controller_SetButtons` function sets the button state of the controller specified by the `controllerRef` parameter to the button state value passed in the `buttons` parameter. If the specified controller is inactive, `Q3Controller_SetButtons` has no effect. Changing a controller's button state might cause the notify function of the tracker currently associated with that controller to be called.

Q3Controller_GetTrackerPosition

You can use the `Q3Controller_GetTrackerPosition` function to get the position of a controller's tracker.

```
TQ3Status Q3Controller_GetTrackerPosition (
                    TQ3ControllerRef controllerRef,
                    TQ3Point3D *position);
```

`controllerRef` A reference to a controller.

`position` On exit, the current position of the tracker associated with the specified controller.

DESCRIPTION

The `Q3Controller_GetTrackerPosition` function returns, in the `position` parameter, the current position of the tracker associated with the controller specified by the `controllerRef` parameter. If no tracker is currently associated with that controller, `Q3Controller_GetTrackerPosition` returns the position of the system cursor's tracker. `Q3Controller_GetTrackerPosition` has no effect if the controller is inactive.

Q3Controller_SetTrackerPosition

You can use the `Q3Controller_SetTrackerPosition` function to set the position of a controller's tracker.

```
TQ3Status Q3Controller_SetTrackerPosition (
                    TQ3ControllerRef controllerRef,
                    const TQ3Point3D *position);
```

`controllerRef` A reference to a controller.

`position` The desired position of the tracker associated with the specified controller.

DESCRIPTION

The `Q3Controller_SetTrackerPosition` function changes the position of the tracker currently associated with the controller specified by the `controllerRef` parameter to the position specified in the `position` parameter. If no tracker is currently associated with that controller, `Q3Controller_SetTrackerPosition` changes the position of the system cursor's tracker. `Q3Controller_SetTrackerPosition` has no effect if the controller is inactive.

Note
Calling `Q3Controller_SetTrackerPosition` might cause the notify function of the controller's tracker to be called. ◆

Q3Controller_MoveTrackerPosition

You can use the `Q3Controller_MoveTrackerPosition` function to move a controller's tracker relative to its current position.

```
TQ3Status Q3Controller_MoveTrackerPosition (
                    TQ3ControllerRef controllerRef,
                    const TQ3Vector3D *delta);
```

`controllerRef` A reference to a controller.

`delta` A three-dimensional vector specifying a relative change in the position of the tracker associated with the specified controller.

DESCRIPTION

The `Q3Controller_MoveTrackerPosition` function changes the position of the tracker currently associated with the controller specified by the `controllerRef` parameter by the relative amount specified in the `delta` parameter. If no tracker is currently associated with that controller, `Q3Controller_MoveTrackerPosition` changes the position of the system cursor's tracker relative to its current position. `Q3Controller_MoveTrackerPosition` has no effect if the controller is inactive.

Note
Calling `Q3Controller_MoveTrackerPosition` might cause the notify function of the controller's tracker to be called. ◆

Q3Controller_GetTrackerOrientation

You can use the `Q3Controller_GetTrackerOrientation` function to get the current orientation of a controller's tracker.

```
TQ3Status Q3Controller_GetTrackerOrientation (
                    TQ3ControllerRef controllerRef,
                    TQ3Quaternion *orientation);
```

`controllerRef` A reference to a controller.

`orientation` On exit, the current orientation of the tracker associated with the specified controller.

DESCRIPTION

The `Q3Controller_GetTrackerOrientation` function returns, in the `orientation` parameter, the current orientation of the tracker associated with the controller specified by the `controllerRef` parameter. If no tracker is currently associated with that controller, `Q3Controller_GetTrackerOrientation` returns the orientation of the system cursor's tracker. `Q3Controller_GetTrackerOrientation` has no effect if the controller is inactive.

Q3Controller_SetTrackerOrientation

You can use the `Q3Controller_SetTrackerOrientation` function to set the orientation of a controller's tracker.

```
TQ3Status Q3Controller_SetTrackerOrientation (
                    TQ3ControllerRef controllerRef,
                    const TQ3Quaternion *orientation);
```

`controllerRef` A reference to a controller.

`orientation` The desired orientation of the tracker associated with the specified controller.

DESCRIPTION

The `Q3Controller_SetTrackerOrientation` function changes the orientation of the tracker currently associated with the controller specified by the `controllerRef` parameter to the orientation specified in the `orientation` parameter. If no tracker is currently associated with that controller, `Q3Controller_SetTrackerOrientation` changes the orientation of the system cursor's tracker. `Q3Controller_SetTrackerOrientation` has no effect if the controller is inactive.

Note
Calling `Q3Controller_SetTrackerOrientation` might cause the notify function of the controller's tracker to be called. ◆

Q3Controller_MoveTrackerOrientation

You can use the `Q3Controller_MoveTrackerOrientation` function to reorient a controller's tracker relative to its current orientation.

```
TQ3Status Q3Controller_MoveTrackerOrientation (
                    TQ3ControllerRef controllerRef,
                    const TQ3Quaternion *delta);
```

`controllerRef` A reference to a controller.

`delta` The desired relative change in the orientation of the tracker associated with the specified controller.

DESCRIPTION

The `Q3Controller_MoveTrackerOrientation` function changes the orientation of the tracker currently associated with the controller specified by the `controllerRef` parameter by the relative amount specified in the `delta` parameter. If no tracker is currently associated with that controller, `Q3Controller_MoveTrackerOrientation` changes the orientation of the system cursor's tracker relative to its current orientation. `Q3Controller_MoveTrackerOrientation` has no effect if the controller is inactive.

Note
Calling `Q3Controller_MoveTrackerOrientation` might cause the notify function of the controller's tracker to be called. ◆

Q3Controller_GetValues

You can use the `Q3Controller_GetValues` function to get the list of values of a controller.

```
TQ3Status Q3Controller_GetValues (
                    TQ3ControllerRef controllerRef,
                    unsigned long valueCount,
                    float *values,
                    TQ3Boolean *changed,
                    unsigned long *serialNumber);
```

`controllerRef` A reference to a controller.

`valueCount` The number of elements in the array pointed to by the `values` parameter.

`values` On entry, a pointer to an array of controller values. The size of the array is determined by the number of elements in the array (as specified by the `valueCount` parameter) and the size of a controller value (which is controller-dependent).

`changed` On exit, a Boolean value that indicates whether the specified array of values was changed (`kQ3True`) or not (`kQ3False`).

`serialNumber` On entry, a controller serial number, or `NULL`.

DESCRIPTION

The `Q3Controller_GetValues` function returns, in the `values` parameter, a pointer to an array that contains the current values for the controller specified in the `controllerRef` parameter. The `valueCount` parameter specifies the number of elements in the array (which you must already have allocated). `Q3Controller_GetValues` might fill in fewer elements if the controller does not support the specified number of values.

If the value of the `serialNumber` parameter is `NULL`, `Q3Controller_GetValues` fills in the `values` array and returns the value `kQ3True` in the `changed` parameter. Otherwise, the value specified in the `serialNumber` parameter is compared with the controller's current serial number. If the two serial numbers are identical, `Q3Controller_GetValues` leaves the `values` array and the `serialNumber` parameter unchanged and returns the value `kQ3False` in the `changed` parameter. If the two serial number differ, `Q3Controller_GetValues` fills in the `values` array, updates the `serialNumber` parameter, and returns the value `kQ3True` in the `changed` parameter.

If the specified controller is inactive, the `values` array and the `changed` parameter are unchanged.

Q3Controller_SetValues

You can use the `Q3Controller_SetValues` function to set the list of values of a controller.

```
TQ3Status Q3Controller_SetValues (
                    TQ3ControllerRef controllerRef,
                    const float *values,
                    unsigned long valueCount);
```

`controllerRef` A reference to a controller.

`values` A pointer to an array of controller values. The size of the array is determined by the number of elements in the array (as specified by the `valueCount` parameter) and the size of a controller value (which is controller-dependent).

`valueCount` The number of elements in the array pointed to by the `values` parameter.

DESCRIPTION

The `Q3Controller_SetValues` function copies the data specified in the `values` parameter into the value list of the controller specified by the `controllerRef` parameter. `Q3Controller_SetValues` copies the number of elements specified by the `valueCount` parameter.

SPECIAL CONSIDERATIONS

In general, you need to use this function only if you are writing a device driver for a controller.

Managing Controller States

QuickDraw 3D provides routines that you can use to save and restore the states of all the channels associated with a controller. You should save the controller states when your application becomes inactive and restore them when it becomes active once again.

Q3ControllerState_New

You can use the `Q3ControllerState_New` function to create a new controller state object.

```
TQ3ControllerStateObject Q3ControllerState_New (
                    TQ3ControllerRef controllerRef);
```

`controllerRef` A reference to a controller.

DESCRIPTION

The `Q3ControllerState_New` function returns, as its function result, a reference to a new controller state object for the controller specified by the `controllerRef` parameter. You need to call `Q3ControllerState_SaveAndReset` to actually fill in the new controller state object with the current channels. If `Q3ControllerState_New` cannot create a new controller state object, it returns `NULL`.

Q3ControllerState_SaveAndReset

You can use the `Q3ControllerState_SaveAndReset` function to save the current state of a controller.

```
TQ3Status Q3ControllerState_SaveAndReset (
                    TQ3ControllerStateObject controllerStateObject);
```

`controllerStateObject`
A controller state object.

DESCRIPTION

The `Q3ControllerState_SaveAndReset` function saves the current state of the controller that is associated with the controller state object specified by the `controllerStateObject` parameter. `Q3ControllerState_SaveAndReset` also resets those channels to their inactive states. You should call `Q3ControllerState_SaveAndReset` to save a controller's channels when your application becomes inactive.

Q3ControllerState_Restore

You can use the `Q3ControllerState_Restore` function to restore a saved set of controller state values.

```
TQ3Status Q3ControllerState_Restore (
                    TQ3ControllerStateObject controllerStateObject);
```

`controllerStateObject`
A controller state object.

DESCRIPTION

The `Q3ControllerState_Restore` function sets the channels of the controller associated with the controller state object specified by the `controllerStateObject` parameter to the channels saved in that state object.

Creating and Managing Trackers

QuickDraw 3D provides routines that you can use to create and manipulate tracker objects.

Q3Tracker_New

You can use the `Q3Tracker_New` function to create a new tracker.

```
TQ3TrackerObject Q3Tracker_New (TQ3TrackerNotifyFunc notifyFunc);
```

`notifyFunc` A pointer to a tracker notify function. See page 1143 for information on writing a tracker notify function.

DESCRIPTION

The `Q3Tracker_New` function returns, as its function result, a reference to a new tracker object. The `notifyFunc` parameter specifies the tracker's notify function, which is called whenever the position or orientation of the tracker changes. If you want to poll for such changes instead of being notified, set `notifyFunc` to `NULL`. The new tracker is active and has both its position threshold and its orientation threshold set to 0. If `Q3Tracker_New` cannot create a new tracker, it returns `NULL`.

Q3Tracker_GetNotifyThresholds

You can use the `Q3Tracker_GetNotifyThresholds` function to get the current notify thresholds of a tracker.

```
TQ3Status Q3Tracker_GetNotifyThresholds (
                    TQ3TrackerObject trackerObject,
                    float *positionThresh,
                    float *orientationThresh);
```

`trackerObject` A tracker object.

`positionThresh`
On exit, the current position threshold of the specified tracker.

`orientationThresh`
On exit, the current orientation threshold (in radians) of the specified tracker.

DESCRIPTION

The `Q3Tracker_GetNotifyThresholds` function returns, in the `positionThresh` and `orientationThresh` parameters, the current position and orientation thresholds of the tracker specified by the `trackerObject` parameter. These thresholds determine whether or not a change in position or orientation is large enough to cause QuickDraw 3D to call the tracker's notify function. Both thresholds for a new tracker are set to 0.

Q3Tracker_SetNotifyThresholds

You can use the `Q3Tracker_SetNotifyThresholds` function to set the notify thresholds of a tracker.

```
TQ3Status Q3Tracker_SetNotifyThresholds (
                    TQ3TrackerObject trackerObject,
                    float positionThresh,
                    float orientationThresh);
```

`trackerObject` A tracker object.

`positionThresh`
The desired position threshold of the specified tracker.

`orientationThresh`
The desired orientation threshold (in radians) of the specified tracker.

DESCRIPTION

The `Q3Tracker_SetNotifyThresholds` function sets the position and orientation thresholds of the tracker specified by the `trackerObject` parameter to the values in the `positionThresh` and `orientationThresh` parameters.

Q3Tracker_GetActivation

You can use the `Q3Tracker_GetActivation` function to get the activation state of a tracker.

```
TQ3Status Q3Tracker_GetActivation (
                    TQ3TrackerObject trackerObject,
                    TQ3Boolean *active);
```

`trackerObject` A tracker object.

`active` On exit, a Boolean value that indicates whether the specified tracker is active (`kQ3True`) or inactive (`kQ3False`).

DESCRIPTION

The `Q3Tracker_GetActivation` function returns, in the `active` parameter, a Boolean value that indicates whether the tracker specified by the `trackerObject` parameter is currently active or inactive.

Q3Tracker_SetActivation

You can use the `Q3Tracker_SetActivation` function to set the activation state of a tracker.

```
TQ3Status Q3Tracker_SetActivation (
                    TQ3TrackerObject trackerObject,
                    TQ3Boolean active);
```

`trackerObject` A tracker object.

`active` A Boolean value that indicates whether the specified tracker is to be made active (`kQ3True`) or inactive (`kQ3False`).

DESCRIPTION

The `Q3Tracker_SetActivation` function sets the activation state of the tracker specified by the `trackerObject` parameter to the value specified in the `active` parameter. If the activation state of a tracker is changed, the serial number of the tracker is incremented.

Q3Tracker_GetEventCoordinates

You can use the `Q3Tracker_GetEventCoordinates` function to get the settings (coordinates) of a tracker that were recorded at a particular moment (typically, the time of a button click) by a previous call to `Q3Tracker_SetEventCoordinates`.

```
TQ3Status Q3Tracker_GetEventCoordinates (
                    TQ3TrackerObject trackerObject,
                    unsigned long timeStamp,
                    unsigned long *buttons,
                    TQ3Point3D *position,
                    TQ3Quaternion *orientation);
```

`trackerObject` A tracker object.

`timeStamp` A time stamp.

`buttons` On exit, the button state value of the specified tracker at the specified time.

`position` On exit, the position of the specified tracker at the specified time. If the tracker is absolute, this parameter contains the absolute coordinates of the tracker. If the tracker is relative, this parameter contains the change in position since the last call to `Q3Tracker_GetEventCoordinates`.

`orientation` On exit, the orientation of the specified tracker at the specified time.

DESCRIPTION

The `Q3Tracker_GetEventCoordinates` function returns, in the `buttons`, `position`, and `orientation` parameters, the button state value, position, and orientation of the tracker specified by the `trackerObject` parameter, at the time specified by the `timeStamp` parameter. You can set any of the `buttons`, `position`, and `orientation` parameters to `NULL` to prevent `Q3Tracker_GetEventCoordinates` from returning a value in that parameter.

`Q3Tracker_GetEventCoordinates` selects the set of event coordinates whose time stamp is closest to the value specified in the `timeStamp` parameter. Any event coordinate sets that are older are discarded from the tracker's ring buffer. If the ring buffer is empty, `Q3Tracker_GetEventCoordinates` returns `kQ3Failure`.

Q3Tracker_SetEventCoordinates

You can use the `Q3Tracker_SetEventCoordinates` function to record the settings (coordinates) of a tracker at a particular time.

```
TQ3Status Q3Tracker_SetEventCoordinates (
                    TQ3TrackerObject trackerObject,
                    unsigned long timeStamp,
                    unsigned long buttons,
                    const TQ3Point3D *position,
                    const TQ3Quaternion *orientation);
```

`trackerObject` A tracker object.

`timeStamp` A time stamp.

`buttons` The button state value of the specified tracker, or `NULL`.

`position` The position of the specified tracker, or `NULL`.

`orientation` The orientation (in radians) of the specified tracker, or `NULL`.

DESCRIPTION

The `Q3Tracker_SetEventCoordinates` function places into the ring buffer of event coordinates for the tracker specified by the `trackerObject` parameter the values specified in the `buttons`, `position`, and `orientation` parameters. The event coordinates are marked with the time stamp specified by the `timeStamp` parameter. If the tracker's ring buffer is full, the oldest item in the buffer is discarded.

Note
A tracker's ring buffer can contain up to 10 items. Time stamps of items in the buffer increase from oldest to newest. ◆

Q3Tracker_GetButtons

You can use the `Q3Tracker_GetButtons` function to get the button state of a tracker.

```
TQ3Status Q3Tracker_GetButtons (
                    TQ3TrackerObject trackerObject,
                    unsigned long *buttons);
```

`trackerObject` A tracker object.

`buttons` On exit, the current button state value of the specified tracker.

DESCRIPTION

The `Q3Tracker_GetButtons` function returns, in the `buttons` parameter, the current button state of the tracker specified by the `trackerObject` parameter.

Q3Tracker_ChangeButtons

You can use the `Q3Tracker_ChangeButtons` function to change the button state of a tracker.

```
TQ3Status Q3Tracker_ChangeButtons (
                    TQ3TrackerObject trackerObject,
                    TQ3ControllerRef controllerRef,
                    unsigned long buttons,
                    unsigned long buttonMask);
```

`trackerObject` A tracker object.

`controllerRef` A reference to a controller.

`buttons` The desired button state value of the specified tracker.

`buttonMask` A button mask.

DESCRIPTION

The `Q3Tracker_ChangeButtons` function sets the button state of the tracker specified by the `trackerObject` parameter to the value specified in the `buttons` parameter. The `buttonMask` parameter specifies a button mask for the tracker. A bit in the mask should be set if the corresponding button has changed since the last call to `Q3Tracker_ChangeButtons`.

The notify function of the specified tracker object may be called when the `Q3Tracker_ChangeButtons` function is executed. If, however, the tracker is inactive when `Q3Tracker_ChangeButtons` is called, the tracker's activation count for the buttons is updated but the notify function is not called.

Note
The `controllerRef` parameter is used only by the tracker's notify function. ◆

Q3Tracker_GetPosition

You can use the `Q3Tracker_GetPosition` function to get the position of a tracker.

```
TQ3Status Q3Tracker_GetPosition (
                    TQ3TrackerObject trackerObject,
                    TQ3Point3D *position,
```

```
                    TQ3Vector3D *delta,
                    TQ3Boolean *changed,
                    unsigned long *serialNumber);
```

`trackerObject`	A tracker object.
`position`	On exit, the current position of the specified tracker.
`delta`	On exit, the change in position since the last call to `Q3Tracker_GetPosition`.
`changed`	On exit, a Boolean value that indicates whether the `position` or `delta` parameter was changed (`kQ3True`) or not (`kQ3False`).
`serialNumber`	On entry, a tracker serial number, or `NULL`. On output, the current tracker serial number.

DESCRIPTION

The `Q3Tracker_GetPosition` function returns, in the `position` parameter, the current position of the tracker specified by the `trackerObject` parameter. In addition, it can return, in the `delta` parameter, the relative change in position since the previous call to `Q3Tracker_GetPosition`.

On entry, if the value of `delta` is `NULL`, the relative contribution is combined into the reported position. If the value of `delta` is not `NULL`, then `delta` is set to the relative motion that has been accumulated since the previous call to `Q3Tracker_GetPosition`. In either case, the position accumulator is set to (0, 0, 0) by this function.

If the value of the `serialNumber` parameter is `NULL`, `Q3Tracker_GetPosition` fills in the `position` and `delta` parameters and returns the value `kQ3True` in the `changed` parameter. Otherwise, the value specified in the `serialNumber` parameter is compared with the tracker's current serial number. If the two serial numbers are identical, `Q3Tracker_GetPosition` leaves the two coordinate parameters and the `serialNumber` parameter unchanged and returns the value `kQ3False` in the `changed` parameter. If the two serial number differ, `Q3Tracker_GetPosition` fills in the two coordinate parameters, updates the `serialNumber` parameter, and returns the value `kQ3True` in the `changed` parameter.

If the specified tracker is inactive, then the `position` parameter is set to the point (0, 0, 0), the `delta` parameter is set to (0, 0, 0) if it is non-`NULL`, and the `changed` parameter is set to `kQ3False` if it is non-`NULL`.

Q3Tracker_SetPosition

You can use the `Q3Tracker_SetPosition` function to set the position of a tracker.

```
TQ3Status Q3Tracker_SetPosition (
                    TQ3TrackerObject trackerObject,
                    TQ3ControllerRef controllerRef,
                    const TQ3Point3D *position);
```

`trackerObject` A tracker object.

`controllerRef` A reference to a controller.

`position` The desired position of the specified tracker.

DESCRIPTION

The `Q3Tracker_SetPosition` function sets the position of the tracker specified by the `trackerObject` and `controllerRef` parameters to the value specified in the `position` parameter. If the specified tracker is inactive, `Q3Tracker_SetPosition` has no effect.

Note
Calling `Q3Tracker_SetPosition` might cause the notify function of the tracker to be called. ◆

Q3Tracker_MovePosition

You can use the `Q3Tracker_MovePosition` function to move the position of a tracker relative to its current position.

```
TQ3Status Q3Tracker_MovePosition (
                    TQ3TrackerObject trackerObject,
                    TQ3ControllerRef controllerRef,
                    const TQ3Vector3D *delta);
```

`trackerObject` A tracker object.

`controllerRef` A reference to a controller.

`delta` The desired change in position of the specified tracker.

DESCRIPTION

The `Q3Tracker_MovePosition` function adds the value specified by the `delta` parameter to the position of the tracker specified by the `trackerObject` and `controllerRef` parameters. If the specified tracker is inactive, `Q3Tracker_MovePosition` has no effect.

Note
Calling `Q3Tracker_MovePosition` might cause the notify function of the tracker to be called. ◆

Q3Tracker_GetOrientation

You can use the `Q3Tracker_GetOrientation` function to get the current orientation of a tracker.

```
TQ3Status Q3Tracker_GetOrientation (
                    TQ3TrackerObject trackerObject,
                    TQ3Quaternion *orientation,
                    TQ3Quaternion *delta,
                    TQ3Boolean *changed,
                    unsigned long *serialNumber);
```

`trackerObject` A tracker object.

`orientation` On exit, the current orientation of the specified tracker.

`delta` On exit, the change in orientation since the last call to `Q3Tracker_GetOrientation`.

`changed` On exit, a Boolean value that indicates whether the `orientation` or `delta` parameters was changed (`kQ3True`) or not (`kQ3False`).

`serialNumber` On entry, a tracker serial number, or `NULL`. On output, the current tracker serial number.

DESCRIPTION

The `Q3Tracker_GetOrientation` function returns, in the `orientation` parameter, the current orientation of the tracker specified by the `trackerObject` parameter. In addition, it may return, in the `delta` parameter, the relative change in orientation since the previous call to `Q3Tracker_GetOrientation`.

On entry, if the value of `delta` is `NULL`, the relative contribution is combined into the reported orientation. If the value of `delta` is not `NULL`, then `delta` is set to the relative motion that has been accumulated since the previous call to `Q3Tracker_GetOrientation`. In either case, the orientation accumulator is set to identity by this function.

If the value of the `serialNumber` parameter is `NULL`, `Q3Tracker_GetOrientation` fills in the `orientation` and `delta` parameters and returns the value `kQ3True` in the `changed` parameter. Otherwise, the value specified in the `serialNumber` parameter is compared with the tracker's current serial number. If the two serial numbers are identical, `Q3Tracker_GetOrientation` leaves the two coordinate parameters and the `serialNumber` parameter unchanged and returns the value `kQ3False` in the `changed` parameter. If the two serial number differ, `Q3Tracker_GetOrientation` fills in the two coordinate parameters, updates the `serialNumber` parameter, and returns the value `kQ3True` in the `changed` parameter.

If the specified tracker is inactive, then the `orientation` parameter is set to identity, the `delta` parameter is set to identity if it is non-`NULL`, and the `changed` parameter is set to `kQ3False` if it is non-`NULL`.

Q3Tracker_SetOrientation

You can use the `Q3Tracker_SetOrientation` function to set the orientation of a tracker.

```
TQ3Status Q3Tracker_SetOrientation (
                    TQ3TrackerObject trackerObject,
                    TQ3ControllerRef controllerRef,
                    const TQ3Quaternion *orientation);
```

`trackerObject` A tracker object.

`controllerRef` A reference to a controller.

`orientation` The desired orientation (in radians) of the specified tracker, or `NULL`.

DESCRIPTION

The `Q3Tracker_SetOrientation` function sets the orientation of the tracker specified by the `trackerObject` and `controllerRef` parameters to the value specified in the `orientation` parameter. If the specified tracker is inactive, `Q3Tracker_SetOrientation` has no effect.

Note
Calling `Q3Tracker_SetOrientation` might cause the notify function of the tracker to be called. ◆

Q3Tracker_MoveOrientation

You can use the `Q3Tracker_MoveOrientation` function to set the orientation of a tracker relative to its current orientation.

```
TQ3Status Q3Tracker_MoveOrientation (
                    TQ3TrackerObject trackerObject,
                    TQ3ControllerRef controllerRef,
                    const TQ3Quaternion *delta);
```

`trackerObject` A tracker object.

`controllerRef` A reference to a controller.

`delta` The desired change in orientation of the specified tracker.

DESCRIPTION

The `Q3Tracker_MoveOrientation` function adds the value specified by the `delta` parameter to the orientation of the tracker specified by the `trackerObject` and `controllerRef` parameters. If the specified tracker is inactive, `Q3Tracker_MoveOrientation` has no effect.

Note
Calling `Q3Tracker_MoveOrientation` might cause the notify function of the tracker to be called. ◆

Application-Defined Routines

This section describes the routines you might need to define when using the routines in the QuickDraw 3D Pointing Device Manager.

TQ3ChannelGetMethod

You can define a function that QuickDraw 3D calls to get a channel of a controller.

```
typedef TQ3Status (*TQ3ChannelGetMethod) (
                    TQ3ControllerRef controllerRef,
                    unsigned long channel,
                    void *data,
                    unsigned long *dataSize);
```

`controllerRef` A reference to a controller.

`channel` An index into the list of channels associated with the specified controller. This value is always greater than or equal to 0 and less than the channel count specified at the time `Q3Controller_New` was called.

`data` On entry, a pointer to a buffer. You should put the current value of the specified controller channel into this buffer.

`dataSize` On exit, the number of bytes of data written to the specified buffer.

DESCRIPTION

Your `TQ3ChannelGetMethod` function should return, in the buffer pointed to by the `data` parameter, the current value of the controller channel specified by the `controllerRef` and `channel` parameters. Your function should also return, in the `dataSize` parameter, the size of that data. QuickDraw 3D allocates memory for

the data buffer before it calls your function and deallocates the memory after your function has returned. The maximum number of bytes that the data buffer can hold is defined by a constant:

```
#define kQ3ControllerSetChannelMaxDataSize      256
```

SPECIAL CONSIDERATIONS

You need to define a channel-getting method only if you are writing a device driver for a controller. You can, however, call `Q3Controller_GetChannel` at any time to invoke a controller's channel-getting method.

RESULT CODES

Your channel-getting method should return `kQ3Success` if it is able to return the requested information and `kQ3Failure` otherwise.

SEE ALSO

See the description of `Q3Controller_GetChannel` on page 1114 for information on getting a controller's channels.

TQ3ChannelSetMethod

You can define a function that QuickDraw 3D calls to set a channel of a controller.

```
typedef TQ3Status (*TQ3ChannelSetMethod) (
                    TQ3ControllerRef controllerRef,
                    unsigned long channel,
                    const void *data,
                    unsigned long dataSize);
```

`controllerRef` A reference to a controller.

`channel`	An index into the list of channels associated with the specified controller. This value is always greater than or equal to 0 and less than the channel count specified at the time `Q3Controller_New` was called.
`data`	On entry, a pointer to a buffer that contains the desired value of the specified controller channel. If this field contains the value `NULL`, you should reset the specified channel to a default or inactive value.
`dataSize`	On entry, the number of bytes of data in the specified buffer.

DESCRIPTION

Your `TQ3ChannelSetMethod` function should set the controller channel specified by the `controllerRef` and `channel` parameters to the value specified by the `data` parameter. The `dataSize` parameter specifies the number of bytes in the data buffer. QuickDraw 3D allocates memory for the data buffer before it calls your function and deallocates the memory after your function has returned. The maximum number of bytes that the data buffer can hold is defined by a constant:

```
#define kQ3ControllerSetChannelMaxDataSize                256
```

SPECIAL CONSIDERATIONS

You need to define a channel-setting method only if you are writing a device driver for a controller. You can, however, call `Q3Controller_SetChannel` at any time to invoke a controller's channel-setting method.

RESULT CODES

Your channel-setting method should return `kQ3Success` if it is able to set the specified channel to the specified value and `kQ3Failure` otherwise.

SEE ALSO

See the description of `Q3Controller_SetChannel` on page 1115 for information on setting a controller's channels.

TQ3TrackerNotifyFunc

You can define a tracker notify function that QuickDraw 3D calls when a controller associated with a tracker has new data.

```
typedef TQ3Status (*TQ3TrackerNotifyFunc) (
                    TQ3TrackerObject trackerObject,
                    TQ3ControllerRef controllerRef);
```

`trackerObject` A tracker object.

`controllerRef` A reference to a controller.

DESCRIPTION

Your `TQ3TrackerNotifyFunc` function is called whenever any controller associated with a tracker has new data to be processed and the data meets or exceeds the current position and orientation thresholds for the tracker. The affected controller and tracker are passed in the `controllerRef` and `trackerObject` parameters. Your tracker notify function might, for example, schedule your application to awaken and redraw the scene.

SPECIAL CONSIDERATIONS

Your tracker notify function might be called at interrupt time, but it is never called reentrantly.

RESULT CODES

Your tracker notify function should return `kQ3Success` if it is successful and `kQ3Failure` otherwise.

SEE ALSO

See the description of `Q3Tracker_New` page 1128 for information on setting the notify function of a tracker.

Cursor Tracker Routines

QuickDraw 3D provides six cursor tracker routines that operate in the same ways as other tracker routines:

```
TQ3Status Q3CursorTracker_PrepareTracking (void);

TQ3Status Q3CursorTracker_SetTrackDeltas(
                    TQ3Boolean    trackDeltas);

TQ3Status Q3CursorTracker_GetAndClearDeltas(
                    float           *depth,
                    TQ3Quaternion   *orientation,
                    TQ3Boolean      *hasOrientation,
                    TQ3Boolean      *changed,
                    unsigned long   *serialNumber);

TQ3Status Q3CursorTracker_SetNotifyFunc(
                    TQ3CursorTrackerNotifyFunc    notifyFunc);

TQ3Status Q3CursorTracker_GetNotifyFunc(
                    TQ3CursorTrackerNotifyFunc    *notifyFunc);

typedef void (*TQ3CursorTrackerNotifyFunc) (void);
```

Pointing Device Errors

Pointing device routines may return the following errors. A list of general QuickDraw 3D errors is given in “QuickDraw 3D Errors, Warnings, and Notices” (page 87).

```
kQ3ErrorController
kQ3ErrorTracker
```

CHAPTER 19

Error Manager

This chapter describes the Error Manager, the part of QuickDraw 3D that you can use to handle any errors or other exceptional conditions that occur during the execution of QuickDraw 3D routines. Lists of QuickDraw 3D errors, warnings, and notices in specific areas are given at the end of each chapter.

About the Error Manager

QuickDraw 3D defines several levels of exceptional conditions that can occur during the execution of QuickDraw 3D routines. An exceptional condition can be an error, a warning, or a notice, depending on the severity of the exceptional condition.

- An **error** is a nonrecoverable condition that causes the currently executing QuickDraw 3D routine to fail. A **fatal error** is an error whose effects persist even after the call that caused it has ended. Once a fatal error has occurred, all future calls to QuickDraw 3D routines are likely to fail. After a nonfatal error, future calls should be limited to the error determination routines described in this chapter. Although some QuickDraw 3D routine calls made after a nonfatal error might succeed, system operation will be unpredictable and may lead to a system crash or loss of data.
- A **warning** is a condition that, although less severe than an error, might cause an error if your application continues execution without handling the warning.
- A **notice** is a condition that is less severe than a warning and will likely not cause problems. In general, notices indicate inefficiencies or other small problems in using QuickDraw 3D.

QuickDraw 3D notifies your application of errors, warnings, and notices by executing application-defined callback routines you have previously registered

with the Error Manager. Once a callback routine is registered, QuickDraw 3D calls it whenever the appropriate condition occurs.

IMPORTANT
Notices are generated only by debugging versions of the QuickDraw 3D shared library. ▲

You register a callback routine by passing its address to the `Q3Error_Register`, `Q3Warning_Register`, or `Q3Notice_Register` function, depending on whether the callback routine is to handle errors, warnings, or notices. If you do not register a callback routine for errors, the Error Manager calls an internal error handler that attempts to handle the exception. The manner in which the exception handler handles that error can vary, depending on the operating system. For example, on the Macintosh Operating System, the internal exception handler of the debugging version calls the `DebugStr` function.

Using the Error Manager

For each level of exceptional condition (that is, for errors, warnings, and notices), QuickDraw 3D keeps track of the first and the most recent exceptional conditions that have occurred since the last time an exceptional condition of that type was posted. For example, when the first error occurs, that error is posted both as the first and as the most recent error. Any subsequent error is posted as the most recent error to occur.

When you call a `_Get` function to retrieve an error, warning, or notice, the function returns, as its function result, the most recent error, warning, or notice. For example, when you call `Q3Error_Get`, it returns, as its function result, the most recent error. `Q3Error_Get` also returns, through its `firstError` parameter, the oldest unreported error that occurred during a QuickDraw 3D routine. You can set this parameter to `NULL` if you do not care about the oldest unreported error.

Note
The oldest unreported error, warning, or notice is sometimes called *sticky.* ◆

Once you've called the `Q3Error_Get` function to retrieve the most recent and the oldest unreported QuickDraw 3D errors, the Error Manager automatically

clears those error codes the next time you call a QuickDraw 3D function that is not part of the Error Manager.

If an error occurs in the operating system on which QuickDraw 3D is running, the Error Manager posts an error indicating which the operating system encountered the error. You can then call an appropriate function to retrieve the system-specific error. For instance, if an error occurs while reading or writing a file in the Macintosh Operating System, then the `Q3Error_Get` function returns the error `kQ3ErrorMacintoshError`. In that case, you can call the `Q3MacintoshError_Get` function to get the Macintosh-specific error code.

Error Manager Reference

This section describes the routines provided by the Error Manager. It also describes the callback routines you can define to handle QuickDraw 3D errors, warnings, and notices.

Error Manager Routines

This section describes the Error Manager routines you can use to handle errors, warnings, and notices.

Registering Error, Warning, and Notice Callback Routines

The Error Manager provides functions that you can use to register error, warning, and notice callback routines.

Q3Error_Register

You can use the `Q3Error_Register` function to register an application-defined error-handling routine.

```
TQ3Status Q3Error_Register (TQ3ErrorMethod errorPost, long reference);
```

`errorPost` A pointer to an application-defined error-handling routine.

`reference` A long integer for your application's own use.

DESCRIPTION

The `Q3Error_Register` function registers with the Error Manager the error-handling routine specified by the `errorPost` parameter. See page 1154 for information on defining an error-handling routine.

Q3Warning_Register

You can use the `Q3Warning_Register` function to register an application-defined warning-handling routine.

```
TQ3Status Q3Warning_Register (
                    TQ3WarningMethod warningPost,
                    long reference);
```

`warningPost` A pointer to an application-defined warning-handling routine.

`reference` A long integer for your application's own use.

DESCRIPTION

The `Q3Warning_Register` function registers with the Error Manager the warning-handling routine specified by the `warningPost` parameter. See page 1155 for information on defining a warning-handling routine.

Q3Notice_Register

You can use the `Q3Notice_Register` function to register an application-defined notice-handling routine.

```
TQ3Status Q3Notice_Register (TQ3NoticeMethod noticePost, long reference);
```

`noticePost` A pointer to an application-defined notice-handling routine.

`reference` A long integer for your application's own use.

DESCRIPTION

The `Q3Notice_Register` function registers with the Error Manager the notice-handling routine specified by the `noticePost` parameter. See page 1156 for information on defining a notice-handling routine.

Determining Whether an Error Is Fatal

The Error Manager provides a routine that you can use to determine whether an error is a fatal error.

Q3Error_IsFatalError

You can use the `Q3Error_IsFatalError` function to determine whether an error is fatal.

```
TQ3Boolean Q3Error_IsFatalError (TQ3Error error);
```

`error` A code that indicates the type of error that has occurred.

DESCRIPTION

The `Q3Error_IsFatalError` function returns, as its function result, a Boolean value that indicates whether the error value specified by the `error` parameter is a fatal error (`kQ3True`) or is not a fatal error (`kQ3False`). You can call `Q3Error_IsFatalError` from within an error-handling method or after having called `Q3Error_Get` to get an error directly. If `Q3Error_IsFatalError` returns `kQ3True`, you should not call any other QuickDraw 3D routines. QuickDraw 3D executes a long jump when it encounters a fatal error; your application should terminate.

Currently, QuickDraw 3D recognizes these errors as fatal:

```
kQ3ErrorInternalError
kQ3ErrorNoRecovery
```

Getting Errors, Warnings, and Notices Directly

The Error Manager provides routines that you can use to retrieve an error, warning, or notice directly.

IMPORTANT
You should use these routines only if you have not already registered an error-, warning-, or notice-handling callback routine. ▲

These routines return the following types, the values of which are enumerated in the header file `QD3DErrors.h`:

```
TQ3Error
TQ3Warning
TQ3Notice
```

Q3Error_Get

You can use the `Q3Error_Get` function to get the most recent and the oldest unreported errors from a QuickDraw 3D routine.

```
TQ3Error Q3Error_Get (TQ3Error *firstError);
```

`firstError` On exit, the first unreported error from a QuickDraw 3D routine. Set this parameter to `NULL` if you do not want the first unreported error to be returned to you.

DESCRIPTION

The `Q3Error_Get` function returns, as its function result, the code of the most recent error that occurred after one or more previous calls to any QuickDraw 3D routines. `Q3Error_Get` causes QuickDraw 3D to clear that error code when you next call any QuickDraw 3D routine other than `Q3Error_Get` itself. `Q3Error_Get` also returns, in the `firstError` parameter, the oldest unreported error that occurred during a QuickDraw 3D routine.

Q3Warning_Get

You can use the `Q3Warning_Get` function to get the most recent and the oldest unreported warnings from a QuickDraw 3D routine.

```
TQ3Warning Q3Warning_Get (TQ3Warning *firstWarning);
```

`firstWarning` On exit, the first unreported warning from a QuickDraw 3D routine. Set this parameter to `NULL` if you do not want the first unreported warning to be returned to you.

DESCRIPTION

The `Q3Warning_Get` function returns, as its function result, the code of the most recent warning that occurred after one or more previous calls to any QuickDraw 3D routines. `Q3Warning_Get` causes QuickDraw 3D to clear that warning code when you next call any QuickDraw 3D routine other than `Q3Warning_Get` itself. `Q3Warning_Get` also returns, in the `firstWarning` parameter, the last unreported warning that occurred during a QuickDraw 3D routine.

Q3Notice_Get

You can use the `Q3Notice_Get` function to get the most recent and the oldest unreported notice from a QuickDraw 3D routine.

```
TQ3Notice Q3Notice_Get (TQ3Notice *firstNotice);
```

`firstNotice` On exit, the first unreported notice from a QuickDraw 3D routine. Set this parameter to `NULL` if you do not want the first unreported notice to be returned to you.

DESCRIPTION

The `Q3Notice_Get` function returns, as its function result, the code of the most recent notice that occurred after one or more previous calls to any QuickDraw 3D routines. `Q3Notice_Get` causes QuickDraw 3D to clear that notice code when you next call any QuickDraw 3D routine other than `Q3Notice_Get`

itself. `Q3Notice_Get` also returns, in the `firstNotice` parameter, the last unreported notice that occurred during a QuickDraw 3D routine.

Notices are returned only by the debugging version of the QuickDraw 3D shared library.

Getting Operating System Errors

The Error Manager provides routines that you can use to retrieve errors that are specific to a particular operating system. In general, these errors are posted by the underlying operating system in response to errors encountered when accessing a file, a resource, or a window system.

Q3MacintoshError_Get

You can use the `Q3MacintoshError_Get` function to get the most recent and the oldest unreported error generated by the Macintosh Operating System.

```
OSErr Q3MacintoshError_Get (OSErr *firstMacErr);
```

`firstMacErr` On exit, the first unreported error from a Macintosh system software routine.

DESCRIPTION

The `Q3MacintoshError_Get` function returns, as its function result, the most recent error generated by the Macintosh system software. `Q3MacintoshError_Get` also returns, in the `firstMacErr` parameter, the first unreported error that occurred during a Macintosh system software routine.

Error-Reporting For Extensions

Extensions may need to report errors, warnings, and notices. These reports are handled by application-supplied handlers, or by the default QuickDraw 3D handlers if the application doesn't provide alternatives.

Q3XError_Post

You can use the `Q3XError_Post` function to post a QuickDraw 3D error from an extension.

```
void Q3XError_Post (TQ3Error error);
```

`error` A code that indicates the type of error that has occurred.

DESCRIPTION

The `Q3XError_Post` function posts the error code passed in the `error` parameter.

Q3XWarning_Post

You can use the `Q3XWarning_Post` function to post a QuickDraw 3D warning from an extension.

```
void Q3XWarning_Post (TQ3Warning warning);
```

`warning` A code that indicates the type of warning.

DESCRIPTION

The `Q3XWarning_Post` function posts the warning code passed in the `warning` parameter.. The warning code you pass into this routine must already be defined in `QD3DErrors.h`.

Q3XNotice_Post

You can use the `Q3XNotice_Post` function to post a QuickDraw 3D notice from an extension.

```
void Q3XNotice_Post (TQ3Notice notice);
```

`notice` A code that indicates the type of notice.

DESCRIPTION

The `Q3XNotice_Post` function posts the notice code passed in the `notice` parameter.. The notice code you pass into this routine must already be defined in `QD3DErrors.h`.

Q3XMacintoshError_Post

You can use the `Q3XMacintoshError_Post` function to post the QuickDraw 3D error `kQ3ErrorMacintoshError` or the Mac OS error `macOSErr`.

```
void Q3XMacintoshError_Post (OSErr macOSErr);
```

`macOSErr` A Mac OS or QuickDraw 3D error code.

DESCRIPTION

The `Q3XMacintoshError_Post` function posts the error code passed in the `macOSErr` parameter. You can retrieve this code by using `Q3MacintoshError_Get`, described on page 1152.

Application-Defined Routines

This section describes the callback routines you can define if you want your application to be automatically informed whenever an error, warning, or notice occurs during the execution of QuickDraw 3D routines.

TQ3ErrorMethod

You can define an error-handling function to handle errors that occur during the execution of QuickDraw 3D routines.

```
typedef void (*TQ3ErrorMethod) (
                    TQ3Error firstError,
                    TQ3Error lastError,
                    long reference);
```

`firstError` A code that indicates the first error that occurred since the last time your error-handling function was called.

`lastError` A code that indicates the most recent error that occurred.

`reference` A long integer for your application's own use.

DESCRIPTION

Your `TQ3ErrorMethod` function is called whenever a QuickDraw 3D routine generates an error (fatal or otherwise) during its execution that QuickDraw 3D cannot handle internally. Your error-handling function should handle the error conditions indicated by the `firstError` and `lastError` parameters. If necessary, you can long jump out of your error method.

Your function must not call any QuickDraw 3D routines other than `Q3Error_IsFatalError` (which you can call to determine if the error was fatal). The `reference` parameter contains the long integer that you passed to `Q3Error_Register` when you registered your error handler. You can, for example, use that long integer to point to any data required by your error handler.

TQ3WarningMethod

You can define a function to handle warnings that occur during the execution of QuickDraw 3D routines.

```
typedef void (*TQ3WarningMethod) (
                    TQ3Warning firstWarning,
                    TQ3Warning lastWarning,
                    long reference);
```

`firstWarning` A code that indicates the first warning that occurred since the last time your warning-handling function was called.

`lastWarning` A code that indicates the most recent warning that occurred.

`reference` A long integer for your application's own use.

DESCRIPTION

Your `TQ3WarningMethod` function is called whenever a QuickDraw 3D routine generates a warning during its execution that QuickDraw 3D cannot handle internally. Your warning-handling function should handle the warning conditions indicated by the `firstWarning` and `lastWarning` parameters. Your function must not call any QuickDraw 3D routines. The `reference` parameter contains the long integer that you passed to `Q3Warning_Register` when you registered your warning handler. You can, for example, use that long integer to point to any data required by your warning handler.

TQ3NoticeMethod

You can define a function to handle notices that occur during the execution of QuickDraw 3D routines.

```
typedef void (*TQ3NoticeMethod) (
                    TQ3Notice firstNotice,
                    TQ3Notice lastNotice,
                    long reference);
```

`firstNotice` A code that indicates the first notice that occurred since the last time your notice-handling function was called.

`lastNotice` A code that indicates the most recent notice that occurred.

`reference` A long integer for your application's own use.

DESCRIPTION

Your `TQ3NoticeMethod` function is called whenever a QuickDraw 3D routine generates a notice during its execution that QuickDraw 3D cannot handle internally. Your notice-handling function should handle the notice conditions indicated by the `firstNotice` and `lastNotice` parameters. Your function must not call any QuickDraw 3D routines. The `reference` parameter contains the long integer that you passed to `Q3Notice_Register` when you registered your notice

handler. You can, for example, use that long integer to point to any data required by your notice handler.

CHAPTER 19

Error Manager

CHAPTER 20

Mathematical Utilities

This chapter describes a large number of mathematical utility functions provided by QuickDraw 3D that you can use to perform mathematical operations on points, vectors, matrices, and quaternions. It also describes the trigonometric and other standard mathematical routines that QuickDraw 3D provides.

To use this chapter, you should already be familiar with the basic definitions of points, vectors, matrices, and quaternions that are in the chapter "Geometric Objects."

About the Mathematical Utilities

QuickDraw 3D provides a large number of utility functions for operating on basic mathematical objects such as points, vectors, matrices, and quaternions. You can use these utilities to

- set the components of points and vectors
- convert dimensions of points and vectors
- subtract points from points
- calculate distances between points
- determine point-relative ratios
- add and subtract points and vectors
- scale vectors
- determine the lengths of vectors
- normalize vectors

- add and subtract vectors
- determine vector cross products and dot products
- transform points and vectors
- negate vectors
- convert points from Cartesian form to polar or spherical form
- determine affine combinations of points
- manipulate matrices
- set up transformation matrices
- calculate trigonometric ratios
- manipulate quaternions

Many of these functions might be implemented as C language macros. As a result, you should avoid such operations as applying the auto-increment operator (++) to function parameters.

QuickDraw 3D also supplies functions that you can use to manage bounding boxes and spheres for any kind of QuickDraw 3D object.

QuickDraw 3D Mathematical Utilities Reference

This section describes the QuickDraw 3D utility routines that you can use to perform mathematical operations on points, vectors, matrices, and quaternions. It also describes the data structures and routines that you can use to manage bounding volumes.

Data Structures

This section describes the data structures you can use to define bounding volumes. QuickDraw 3D provides two kinds of bounding volumes:

- bounding boxes
- bounding spheres

Bounding Boxes

A bounding box is a rectangular box, aligned with the coordinate axes, that completely encloses an object. A bounding box is defined by the `TQ3BoundingBox` data type.

```
typedef struct TQ3BoundingBox {
    TQ3Point3D                      min;
    TQ3Point3D                      max;
    TQ3Boolean                      isEmpty;
} TQ3BoundingBox;
```

Field descriptions

`min`	The lower-left corner of the bounding box.
`max`	The upper-right corner of the bounding box.
`isEmpty`	A Boolean value that specifies whether the bounding box is empty (`kQ3True`) or not (`kQ3False`). If this field contains the value `kQ3True`, the other field of this structure are invalid.

Bounding Spheres

A bounding sphere is a sphere that completely encloses an object. A bounding sphere is defined by the `TQ3BoundingSphere` data type.

```
typedef struct TQ3BoundingSphere {
    TQ3Point3D                      origin;
    float                           radius;
    TQ3Boolean                      isEmpty;
} TQ3BoundingSphere;
```

Field descriptions

`origin`	The origin of the bounding sphere.
`radius`	The radius of the bounding sphere; all points making up the bounding sphere are this far away from the origin of the sphere.
`isEmpty`	A Boolean value that specifies whether the bounding sphere is empty (`kQ3True`) or not (`kQ3False`). If this field contains the value `kQ3True`, the other field of this structure are invalid.

QuickDraw 3D Mathematical Utilities

This section describes QuickDraw 3D's utility functions for operating on basic mathematical objects such as points, vectors, matrices, and quaternions. It also describes routines you can use to manage bounding volumes.

IMPORTANT

QuickDraw 3D mathematical utilities do not automatically normalize vectors. They expect vectors to be normalized by the calling code. ▲

Setting Points and Vectors

QuickDraw 3D supplies routines that you can use to set the components of a point or vector. You must already have allocated space for the point or vector before attempting to modify its contents.

Q3Point2D_Set

You can use the `Q3Point2D_Set` function to set the coordinates of a two-dimensional point.

```
TQ3Point2D *Q3Point2D_Set (
                    TQ3Point2D *point2D,
                    float x,
                    float y);
```

`point2D`	A two-dimensional point.
`x`	The *x* coordinate of the point.
`y`	The *y* coordinate of the point.

DESCRIPTION

The `Q3Point2D_Set` function returns, as its function result and in the `point2D` parameter, the two-dimensional point specified by the `x` and `y` parameters.

Q3Param2D_Set

You can use the `Q3Param2D_Set` function to set the components of a two-dimensional parametric point.

```
TQ3Param2D *Q3Param2D_Set (
                    TQ3Param2D *param2D,
                    float u,
                    float v);
```

`param2D` A parametric point.

`u` The *u* component of the parametric point.

`v` The *v* component of the parametric point.

DESCRIPTION

The `Q3Param2D_Set` function returns, as its function result and in the `param2D` parameter, the two-dimensional parametric point specified by the `u` and `v` parameters.

Q3Point3D_Set

You can use the `Q3Point3D_Set` function to set the coordinates of a three-dimensional point.

```
TQ3Point3D *Q3Point3D_Set (
                    TQ3Point3D *point3D,
                    float x,
                    float y,
                    float z);
```

`point3D` A three-dimensional point.

`x` The *x* coordinate of the point.

`y` The *y* coordinate of the point.

`z` The *z* coordinate of the point.

DESCRIPTION

The `Q3Point3D_Set` function returns, as its function result and in the `point3D` parameter, the three-dimensional point specified by the `x`, `y`, and `z` parameters.

Q3RationalPoint3D_Set

You can use the `Q3RationalPoint3D_Set` function to set the coordinates of a three-dimensional rational point.

```
TQ3RationalPoint3D *Q3RationalPoint3D_Set (
                    TQ3RationalPoint3D *point3D,
                    float x,
                    float y,
                    float w);
```

`point3D`	A three-dimensional point.
`x`	The *x* coordinate of the point.
`y`	The *y* coordinate of the point.
`w`	The *w* coordinate of the point.

DESCRIPTION

The `Q3RationalPoint3D_Set` function returns, as its function result and in the `point3D` parameter, the three-dimensional rational point specified by the `x`, `y`, and `w` parameters.

Q3RationalPoint4D_Set

You can use the `Q3RationalPoint4D_Set` function to set the coordinates of a four-dimensional rational point.

```
TQ3RationalPoint4D *Q3RationalPoint4D_Set (
                    TQ3RationalPoint4D *point4D,
                    float x,
```

```
                    float y,
                    float z,
                    float w);
```

`point4D`	A four-dimensional point.
`x`	The *x* coordinate of the point.
`y`	The *y* coordinate of the point.
`z`	The *z* coordinate of the point.
`w`	The *w* coordinate of the point.

DESCRIPTION

The `Q3RationalPoint4D_Set` function returns, as its function result and in the `point4D` parameter, the four-dimensional rational point specified by the `x`, `y`, `z`, and `w` parameters.

Q3PolarPoint_Set

You can use the `Q3PolarPoint_Set` function to set the components of a polar point.

```
TQ3PolarPoint *Q3PolarPoint_Set (
                    TQ3PolarPoint *polarPoint,
                    float r,
                    float theta);
```

`polarPoint`	A polar point.
`r`	The *r* component of the polar point.
`theta`	The θ component of the polar point.

DESCRIPTION

The `Q3PolarPoint_Set` function returns, as its function result and in the `polarPoint` parameter, the polar point specified by the `r` and `theta` parameters.

Q3SphericalPoint_Set

You can use the `Q3SphericalPoint_Set` function to set the components of a spherical point.

```
TQ3SphericalPoint *Q3SphericalPoint_Set (
                    TQ3SphericalPoint *sphericalPoint,
                    float rho,
                    float theta,
                    float phi);
```

`sphericalPoint`
A spherical point.

`rho` The ρ component of the spherical point.

`theta` The θ component of the spherical point.

`phi` The ϕ component of the spherical point.

DESCRIPTION

The `Q3SphericalPoint_Set` function returns, as its function result and in the `sphericalPoint` parameter, the spherical point specified by the `rho`, `theta`, and `phi` parameters.

Q3Vector2D_Set

You can use the `Q3Vector2D_Set` function to set the scalar components of a two-dimensional vector.

```
TQ3Vector2D *Q3Vector2D_Set (
                    TQ3Vector2D *vector2D,
                    float x,
                    float y);
```

`vector2D` A two-dimensional vector.

`x` The *x* scalar component of the vector.

`y` The *y* scalar component of the vector.

DESCRIPTION

The `Q3Vector2D_Set` function returns, as its function result and in the `vector2D` parameter, the two-dimensional vector whose scalar components are specified by the `x` and `y` parameters.

Q3Vector3D_Set

You can use the `Q3Vector3D_Set` function to set the scalar components of a three-dimensional vector.

```
TQ3Vector3D *Q3Vector3D_Set (
                    TQ3Vector3D *vector3D,
                    float x,
                    float y,
                    float z);
```

`vector3D`	A three-dimensional vector.
`x`	The *x* scalar component of the vector.
`y`	The *y* scalar component of the vector.
`z`	The *z* scalar component of the vector.

DESCRIPTION

The `Q3Vector3D_Set` function returns, as its function result and in the `vector3D` parameter, the three-dimensional vector whose scalar components are specified by the `x`, `y`, and `z` parameters.

Converting Dimensions of Points and Vectors

QuickDraw 3D provides routines that you can use to convert a point or vector of a given dimension to another dimension. When the given dimension is less than the result dimension, the last component is set to 1.0. When the given dimension is greater than the result dimension, each component in the result structure is set to its corresponding component in the given structure divided by the last component.

IMPORTANT
You must already have allocated space for the result structure before attempting to convert the dimension of a point or vector. ▲

Q3Point2D_To3D

You can use the `Q3Point2D_To3D` function to convert a two-dimensional point to a three-dimensional point.

```
TQ3Point3D *Q3Point2D_To3D (
                    const TQ3Point2D *point2D,
                    TQ3Point3D *result);
```

`point2D` A two-dimensional point.

`result` On exit, a three-dimensional point.

DESCRIPTION

The `Q3Point2D_To3D` function returns, as its function result and in the `result` parameter, the three-dimensional point that corresponds to the two-dimensional point `point2D`.

Q3Point3D_To4D

You can use the `Q3Point3D_To4D` function to convert a three-dimensional point to a four-dimensional point.

```
TQ3RationalPoint4D *Q3Point3D_To4D (
                    const TQ3Point3D *point3D,
                    TQ3RationalPoint4D *result);
```

`point3D` A three-dimensional point.

`result` On exit, a rational four-dimensional point.

DESCRIPTION

The `Q3Point3D_To4D` function returns, as its function result and in the `result` parameter, the rational four-dimensional point that corresponds to the three-dimensional point `point3D`.

Q3RationalPoint3D_To2D

You can use the `Q3RationalPoint3D_To2D` function to convert a three-dimensional rational point to a two-dimensional point.

```
TQ3Point2D *Q3RationalPoint3D_To2D (
                    const TQ3RationalPoint3D *point3D,
                    TQ3Point2D *result);
```

`point3D` A rational three-dimensional point.

`result` On exit, a two-dimensional point.

DESCRIPTION

The `Q3RationalPoint3D_To2D` function returns, as its function result and in the `result` parameter, the two-dimensional point that corresponds to the rational three-dimensional point `point3D`.

Q3RationalPoint4D_To3D

You can use the `Q3RationalPoint4D_To3D` function to convert a four-dimensional rational point to a three-dimensional point.

```
TQ3Point3D *Q3RationalPoint4D_To3D (
                    const TQ3RationalPoint4D *point4D,
                    TQ3Point3D *result);
```

`point4D` A rational four-dimensional point.

`result` On exit, a three-dimensional point.

DESCRIPTION

The `Q3RationalPoint4D_To3D` function returns, as its function result and in the `result` parameter, the three-dimensional point that corresponds to the rational four-dimensional point `point4D`.

Q3Vector2D_To3D

You can use the `Q3Vector2D_To3D` function to convert a two-dimensional vector to a three-dimensional vector.

```
TQ3Vector3D *Q3Vector2D_To3D (
                    const TQ3Vector2D *vector2D,
                    TQ3Vector3D *result);
```

`vector2D` A two-dimensional vector.

`result` On exit, a three-dimensional vector.

DESCRIPTION

The `Q3Vector2D_To3D` function returns, as its function result and in the `result` parameter, the three-dimensional vector that corresponds to the two-dimensional vector `vector2D`.

Q3Vector3D_To2D

You can use the `Q3Vector3D_To2D` function to convert a three-dimensional vector to a two-dimensional vector.

```
TQ3Vector2D *Q3Vector3D_To2D (
                    const TQ3Vector3D *vector3D,
                    TQ3Vector2D *result);
```

`vector3D` A three-dimensional vector.

`result` On exit, a two-dimensional vector.

DESCRIPTION

The `Q3Vector3D_To2D` function returns, as its function result and in the `result` parameter, the two-dimensional vector that corresponds to the three-dimensional vector `vector3D`.

Subtracting Points

QuickDraw 3D provides routines that you can use to subtract a point of a given dimension from another of the same dimension. All of these routines return a vector that is the difference of the two points.

Q3Point2D_Subtract

You can use the `Q3Point2D_Subtract` function to subtract one two-dimensional point from another.

```
TQ3Vector2D *Q3Point2D_Subtract (
                    const TQ3Point2D *p1,
                    const TQ3Point2D *p2,
                    TQ3Vector2D *result);
```

`p1` A two-dimensional point.

`p2` A two-dimensional point.

`result` On exit, a two-dimensional vector that is the result of subtracting the point `p2` from `p1`.

DESCRIPTION

The `Q3Point2D_Subtract` function returns, as its function result and in the `result` parameter, the two-dimensional vector that is the result of subtracting the point `p2` from `p1`.

Q3Param2D_Subtract

You can use the `Q3Param2D_Subtract` function to subtract one two-dimensional parametric point from another.

```
TQ3Vector2D *Q3Param2D_Subtract (
                    const TQ3Param2D *p1,
                    const TQ3Param2D *p2,
                    TQ3Vector2D *result);
```

`p1` A two-dimensional parametric point.

`p2` A two-dimensional parametric point.

`result` On exit, a two-dimensional vector that is the result of subtracting the parametric point `p2` from `p1`.

DESCRIPTION

The `Q3Param2D_Subtract` function returns, as its function result and in the `result` parameter, the two-dimensional vector that is the result of subtracting the parametric point `p2` from `p1`.

Q3Point3D_Subtract

You can use the `Q3Point3D_Subtract` function to subtract one three-dimensional point from another.

```
TQ3Vector3D *Q3Point3D_Subtract (
                    const TQ3Point3D *p1,
                    const TQ3Point3D *p2,
                    TQ3Vector3D *result);
```

`p1` A three-dimensional point.

`p2` A three-dimensional point.

`result` On exit, a three-dimensional vector that is the result of subtracting the point `p2` from `p1`.

DESCRIPTION

The `Q3Point3D_Subtract` function returns, as its function result and in the `result` parameter, the three-dimensional vector that is the result of subtracting the point `p2` from `p1`.

Calculating Distances Between Points

QuickDraw 3D provides routines that you can use to determine the distance between two points. QuickDraw 3D also provides routines that you can use to determine the square of the distance between two points. These distance-squared routines are much faster than the simple distance routines and are therefore recommended for situations in which only relative distances are important to you.

Q3Point2D_Distance

You can use the `Q3Point2D_Distance` function to determine the distance between two two-dimensional points.

```
float Q3Point2D_Distance (
                    const TQ3Point2D *p1,
                    const TQ3Point2D *p2);
```

`p1` A two-dimensional point.

`p2` A two-dimensional point.

DESCRIPTION

The `Q3Point2D_Distance` function returns, as its function result, the absolute value of the distance between points `p1` and `p2`.

Q3Param2D_Distance

You can use the `Q3Param2D_Distance` function to determine the distance between two two-dimensional parametric points.

```
float Q3Param2D_Distance (
                    const TQ3Param2D *p1,
                    const TQ3Param2D *p2);
```

`p1` A two-dimensional parametric point.

`p2` A two-dimensional parametric point.

DESCRIPTION

The `Q3Param2D_Distance` function returns, as its function result, the absolute value of the distance between parametric points `p1` and `p2`.

Q3Point3D_Distance

You can use the `Q3Point3D_Distance` function to determine the distance between two three-dimensional points.

```
float Q3Point3D_Distance (
                    const TQ3Point3D *p1,
                    const TQ3Point3D *p2);
```

`p1` A three-dimensional point.

`p2` A three-dimensional point.

DESCRIPTION

The `Q3Point3D_Distance` function returns, as its function result, the absolute value of the distance between points `p1` and `p2`.

Q3RationalPoint3D_Distance

You can use the `Q3RationalPoint3D_Distance` function to determine the distance between two three-dimensional rational points.

```
float Q3RationalPoint3D_Distance (
                    const TQ3RationalPoint3D *p1,
                    const TQ3RationalPoint3D *p2);
```

`p1` A rational three-dimensional point.

`p2` A rational three-dimensional point.

DESCRIPTION

The `Q3RationalPoint3D_Distance` function returns, as its function result, the absolute value of the distance between points `p1` and `p2`. The distance returned is a two-dimensional distance.

Q3RationalPoint4D_Distance

You can use the `Q3RationalPoint4D_Distance` function to determine the distance between two four-dimensional rational points.

```
float Q3RationalPoint4D_Distance (
                    const TQ3RationalPoint4D *p1,
                    const TQ3RationalPoint4D *p2);
```

`p1` A rational four-dimensional point.

`p2` A rational four-dimensional point.

DESCRIPTION

The `Q3RationalPoint4D_Distance` function returns, as its function result, the absolute value of the distance between points `p1` and `p2`. The distance returned is a three-dimensional distance.

Q3Point2D_DistanceSquared

You can use the `Q3Point2D_DistanceSquared` function to determine the square of the distance between two two-dimensional points.

```
float Q3Point2D_DistanceSquared (
                    const TQ3Point2D *p1,
                    const TQ3Point2D *p2);
```

`p1` A two-dimensional point.

`p2` A two-dimensional point.

DESCRIPTION

The `Q3Point2D_DistanceSquared` function returns, as its function result, the square of the distance between points `p1` and `p2`.

Q3Param2D_DistanceSquared

You can use the `Q3Param2D_DistanceSquared` function to determine the square of the distance between two two-dimensional parametric points.

```
float Q3Param2D_DistanceSquared (
                    const TQ3Param2D *p1,
                    const TQ3Param2D *p2);
```

`p1` A two-dimensional parametric point.

`p2` A two-dimensional parametric point.

DESCRIPTION

The `Q3Param2D_DistanceSquared` function returns, as its function result, the square of the distance between parametric points `p1` and `p2`.

Q3Point3D_DistanceSquared

You can use the `Q3Point3D_DistanceSquared` function to determine the square of the distance between two three-dimensional points.

```
float Q3Point3D_DistanceSquared (
                    const TQ3Point3D *p1,
                    const TQ3Point3D *p2);
```

`p1` A three-dimensional point.

`p2` A three-dimensional point.

DESCRIPTION

The `Q3Point3D_DistanceSquared` function returns, as its function result, the square of the distance between points `p1` and `p2`.

Q3RationalPoint3D_DistanceSquared

You can use the `Q3RationalPoint3D_DistanceSquared` function to determine the square of the distance between two rational three-dimensional points.

```
float Q3RationalPoint3D_DistanceSquared (
                    const TQ3RationalPoint3D *p1,
                    const TQ3RationalPoint3D *p2);
```

`p1` A rational three-dimensional point.

`p2` A rational three-dimensional point.

DESCRIPTION

The `Q3RationalPoint3D_DistanceSquared` function returns, as its function result, the square of the distance between points `p1` and `p2`. The distance returned is a two-dimensional distance.

Q3RationalPoint4D_DistanceSquared

You can use the `Q3RationalPoint4D_DistanceSquared` function to determine the square of the distance between two rational four-dimensional points.

```
float Q3RationalPoint4D_DistanceSquared (
                    const TQ3RationalPoint4D *p1,
                    const TQ3RationalPoint4D *p2);
```

`p1` A rational four-dimensional point.

`p2` A rational four-dimensional point.

DESCRIPTION

The `Q3RationalPoint4D_DistanceSquared` function returns, as its function result, the square of the distance between points `p1` and `p2`. The distance returned is a three-dimensional distance.

Determining Point Relative Ratios

QuickDraw 3D provides routines that you can use to determine point-relative ratios between two points. These routines return a point on the line segment defined by those two points that is at a desired distance from the first point.

Q3Point2D_RRatio

You can use the `Q3Point2D_RRatio` function to find a point lying between two given two-dimensional points that is at a desired distance ratio from one of those points.

```
TQ3Point2D *Q3Point2D_RRatio (
                    const TQ3Point2D *p1,
                    const TQ3Point2D *p2,
                    float r1,
                    float r2,
                    TQ3Point2D *result);
```

`p1` A two-dimensional point.

`p2` A two-dimensional point.

`r1` A floating-point number.

`r2` A floating-point number.

`result` On exit, the two-dimensional point that is at a desired distance ratio from `p1` along the line segment between `p1` and `p2`.

DESCRIPTION

The `Q3Point2D_RRatio` function returns, as its function result and in the `result` parameter, the two-dimensional point that lies on the line segment between the points `p1` and `p2` and that is at a distance from the first point determined by the ratio `r1`/(`r1` + `r2`).

Q3Param2D_RRatio

You can use the `Q3Param2D_RRatio` function to find a point lying between two given two-dimensional parametric points that is at a desired distance ratio from one of those points.

```
TQ3Param2D *Q3Param2D_RRatio (
                    const TQ3Param2D *p1,
                    const TQ3Param2D *p2,
                    float r1,
                    float r2,
                    TQ3Param2D *result);
```

`p1` A two-dimensional parametric point.

`p2` A two-dimensional parametric point.

`r1` A floating-point number.

`r2` A floating-point number.

`result` On exit, the two-dimensional parametric point that is at a desired distance ratio from `p1` along the line segment between `p1` and `p2`.

DESCRIPTION

The `Q3Param2D_RRatio` function returns, as its function result and in the `result` parameter, the two-dimensional parametric point that lies on the line segment

between the points `p1` and `p2` and that is at a distance from the first parametric point determined by the ratio `r1`/(`r1` + `r2`).

Q3Point3D_RRatio

You can use the `Q3Point3D_RRatio` function to find a point lying between two given three-dimensional points that is at a desired distance ratio from one of those points.

```
TQ3Point3D *Q3Point3D_RRatio (
                    const TQ3Point3D *p1,
                    const TQ3Point3D *p2,
                    float r1,
                    float r2,
                    TQ3Point3D *result);
```

`p1` A three-dimensional point.

`p2` A three-dimensional point.

`r1` A floating-point number.

`r2` A floating-point number.

`result` On exit, the three-dimensional point that is at a desired distance ratio from `p1` along the line segment between `p1` and `p2`.

DESCRIPTION

The `Q3Point3D_RRatio` function returns, as its function result and in the `result` parameter, the three-dimensional point that lies on the line segment between the points `p1` and `p2` and that is at a distance from the first point determined by the ratio `r1`/(`r1` + `r2`).

Q3RationalPoint4D_RRatio

You can use the `Q3RationalPoint4D_RRatio` function to find a point lying between two given four-dimensional points that is at a desired distance ratio from one of those points.

```
TQ3RationalPoint4D *Q3RationalPoint4D_RRatio (
                    const TQ3RationalPoint4D *p1,
                    const TQ3RationalPoint4D *p2,
                    float r1,
                    float r2,
                    TQ3RationalPoint4D *result);
```

`p1` A rational four-dimensional point.

`p2` A rational four-dimensional point.

`r1` A floating-point number.

`r2` A floating-point number.

`result` On exit, the four-dimensional point that is at a desired distance ratio from `p1` along the line segment between `p1` and `p2`.

DESCRIPTION

The `Q3RationalPoint4D_RRatio` function returns, as its function result and in the `result` parameter, the four-dimensional point that lies on the line segment lying between the points `p1` and `p2` and that is at a distance from the first point determined by the ratio `r1`/(`r1` + `r2`).

Adding and Subtracting Points and Vectors

QuickDraw 3D provides routines that you can use to add a vector to a point or subtract a vector from a point. For increased floating-point precision, it is better to use the vector-point subtraction routines than to reverse a vector and then add it to a point.

Q3Point2D_Vector2D_Add

You can use the `Q3Point2D_Vector2D_Add` function to add a two-dimensional vector to a two-dimensional point.

```
TQ3Point2D *Q3Point2D_Vector2D_Add (
                    const TQ3Point2D *point2D,
                    const TQ3Vector2D *vector2D,
                    TQ3Point2D *result);
```

`point2D` A two-dimensional point.

`vector2D` A two-dimensional vector.

`result` On exit, a two-dimensional point that is the result of adding `vector2D` to `point2D`.

DESCRIPTION

The `Q3Point2D_Vector2D_Add` function returns, as its function result and in the `result` parameter, the two-dimensional point that is the result of adding the vector `vector2D` to the point `point2D`.

Q3Param2D_Vector2D_Add

You can use the `Q3Param2D_Vector2D_Add` function to add a two-dimensional vector to a two-dimensional parametric point.

```
TQ3Param2D *Q3Param2D_Vector2D_Add (
                    const TQ3Param2D *param2D,
                    const TQ3Vector2D *vector2D,
                    TQ3Param2D *result);
```

`param2D` A two-dimensional parametric point.

`vector2D` A two-dimensional vector.

`result` On exit, a two-dimensional point that is the result of adding `vector2D` to `param2D`.

DESCRIPTION

The `Q3Param2D_Vector2D_Add` function returns, as its function result and in the `result` parameter, the two-dimensional parametric point that is the result of adding the vector `vector2D` to the parametric point `param2D`.

Q3Point3D_Vector3D_Add

You can use the `Q3Point3D_Vector3D_Add` function to add a three-dimensional vector to a three-dimensional point.

```
TQ3Point3D *Q3Point3D_Vector3D_Add (
                    const TQ3Point3D *point3D,
                    const TQ3Vector3D *vector3D,
                    TQ3Point3D *result);
```

`point3D`	A three-dimensional point.
`vector3D`	A three-dimensional vector.
`result`	On exit, a three-dimensional point that is the result of adding `vector3D` to `point3D`.

DESCRIPTION

The `Q3Point3D_Vector3D_Add` function returns, as its function result and in the `result` parameter, the three-dimensional point that is the result of adding the vector `vector3D` to the point `point3D`.

Q3Point2D_Vector2D_Subtract

You can use the `Q3Point2D_Vector2D_Subtract` function to subtract a two-dimensional vector from a two-dimensional point.

```
TQ3Point2D *Q3Point2D_Vector2D_Subtract (
                    const TQ3Point2D *point2D,
                    const TQ3Vector2D *vector2D,
                    TQ3Point2D *result);
```

point2D	A two-dimensional point.
vector2D	A two-dimensional vector.
result	On exit, a two-dimensional point that is the result of subtracting vector2D from point2D.

DESCRIPTION

The Q3Point2D_Vector2D_Subtract function returns, as its function result and in the result parameter, the two-dimensional point that is the result of subtracting the vector vector2D from the point point2D.

Q3Param2D_Vector2D_Subtract

You can use the Q3Param2D_Vector2D_Subtract function to subtract a two-dimensional vector from a two-dimensional parametric point.

```
TQ3Param2D *Q3Param2D_Vector2D_Subtract (
                    const TQ3Param2D *param2D,
                    const TQ3Vector2D *vector2D,
                    TQ3Param2D *result);
```

param2D	A two-dimensional parametric point.
vector2D	A two-dimensional vector.
result	On exit, a two-dimensional parametric point that is the result of subtracting vector2D from param2D.

DESCRIPTION

The Q3Param2D_Vector2D_Subtract function returns, as its function result and in the result parameter, the two-dimensional parametric point that is the result of subtracting the vector vector2D from the point param2D.

Q3Point3D_Vector3D_Subtract

You can use the `Q3Point3D_Vector3D_Subtract` function to subtract a three-dimensional vector from a three-dimensional point.

```
TQ3Point3D *Q3Point3D_Vector3D_Subtract (
                    const TQ3Point3D *point3D,
                    const TQ3Vector3D *vector3D,
                    TQ3Point3D *result);
```

`point3D` A three-dimensional point.

`vector3D` A three-dimensional vector.

`result` On exit, a three-dimensional point that is the result of subtracting `vector3D` from `point3D`.

DESCRIPTION

The `Q3Point3D_Vector3D_Subtract` function returns, as its function result and in the `result` parameter, the three-dimensional point that is the result of subtracting the vector `vector3D` from the point `point3D`.

Scaling Vectors

QuickDraw 3D provides routines that you can use to multiply a vector by a floating-point scalar value.

Q3Vector2D_Scale

You can use the `Q3Vector2D_Scale` function to scale a two-dimensional vector.

```
TQ3Vector2D *Q3Vector2D_Scale (
                    const TQ3Vector2D *vector2D,
                    float scalar,
                    TQ3Vector2D *result);
```

vector2D	A two-dimensional vector.
scalar	A floating-point number.
result	On exit, a two-dimensional vector that is the result of multiplying each of the components of vector2D by the value of the scalar parameter.

DESCRIPTION

The Q3Vector2D_Scale function returns, as its function result and in the result parameter, the two-dimensional vector that is the result of multiplying each of the components of the vector vector2D by the value of the scalar parameter. Note that on entry the result parameter can be the same as the vector2D parameter.

Q3Vector3D_Scale

You can use the Q3Vector3D_Scale function to scale a three-dimensional vector.

```
TQ3Vector3D *Q3Vector3D_Scale (
                    const TQ3Vector3D *vector3D,
                    float scalar,
                    TQ3Vector3D *result);
```

vector3D	A three-dimensional vector.
scalar	A floating-point number.
result	On exit, a three-dimensional vector that is the result of multiplying each of its components by the value of the scalar parameter.

DESCRIPTION

The Q3Vector3D_Scale function returns, as its function result and in the result parameter, the three-dimensional vector that is the result of multiplying each of the components of the vector vector3D by the value of the scalar parameter. Note that on entry the result parameter can be the same as the vector3D parameter.

Determining the Lengths of Vectors

QuickDraw 3D provides routines that you can use to determine the length of a vector.

Q3Vector2D_Length

You can use the `Q3Vector2D_Length` function to determine the length of a two-dimensional vector.

```
float Q3Vector2D_Length (const TQ3Vector2D *vector2D);
```

`vector2D` A two-dimensional vector.

DESCRIPTION

The `Q3Vector2D_Length` function returns, as its function result, the length of the vector `vector2D`.

Q3Vector3D_Length

You can use the `Q3Vector3D_Length` function to determine the length of a three-dimensional vector.

```
float Q3Vector3D_Length (const TQ3Vector3D *vector3D);
```

`vector3D` A three-dimensional vector.

DESCRIPTION

The `Q3Vector3D_Length` function returns, as its function result, the length of the vector `vector3D`.

Normalizing Vectors

QuickDraw 3D provides routines that you can use to normalize a vector. The normalized form of a vector is the vector having the same direction as the given vector but a length equal to 1.0.

Q3Vector2D_Normalize

You can use the `Q3Vector2D_Normalize` function to normalize a two-dimensional vector.

```
TQ3Vector2D *Q3Vector2D_Normalize (
                    const TQ3Vector2D *vector2D,
                    TQ3Vector2D *result);
```

`vector2D` A two-dimensional vector.

`result` On exit, the normalized form of the specified vector.

DESCRIPTION

The `Q3Vector2D_Normalize` function returns, as its function result and in the `result` parameter, the normalized form of the vector `vector2D`. Note that on entry the `result` parameter can be the same as the `vector2D` parameter.

Q3Vector3D_Normalize

You can use the `Q3Vector3D_Normalize` function to normalize a three-dimensional vector.

```
TQ3Vector3D *Q3Vector3D_Normalize (
                    const TQ3Vector3D *vector3D,
                    TQ3Vector3D *result);
```

`vector3D` A three-dimensional vector.

`result` On exit, the normalized form of the specified vector.

DESCRIPTION

The `Q3Vector3D_Normalize` function returns, as its function result and in the `result` parameter, the normalized form of the vector `vector3D`. Note that on entry the `result` parameter can be the same as the `vector3D` parameter.

Adding and Subtracting Vectors

QuickDraw 3D provides routines that you can use to add a vector to a vector or to subtract a vector from a vector.

Q3Vector2D_Add

You can use the `Q3Vector2D_Add` function to add two two-dimensional vectors.

```
TQ3Vector2D *Q3Vector2D_Add (
                    const TQ3Vector2D *v1,
                    const TQ3Vector2D *v2,
                    TQ3Vector2D *result);
```

`v1` A two-dimensional vector.

`v2` A two-dimensional vector.

`result` On exit, the sum of `v1` and `v2`.

DESCRIPTION

The `Q3Vector2D_Add` function returns, as its function result and in the `result` parameter, the two-dimensional vector that is the sum of the two vectors `v1` and `v2`. Note that on entry the `result` parameter can be the same as either `v1` or `v2` (or both).

Q3Vector3D_Add

You can use the `Q3Vector3D_Add` function to add two three-dimensional vectors.

```
TQ3Vector3D *Q3Vector3D_Add (
                    const TQ3Vector3D *v1,
                    const TQ3Vector3D *v2,
                    TQ3Vector3D *result);
```

v1	A three-dimensional vector.
v2	A three-dimensional vector.
result	On exit, the sum of v1 and v2.

DESCRIPTION

The Q3Vector3D_Add function returns, as its function result and in the result parameter, the three-dimensional vector that is the sum of the two vectors v1 and v2. Note that on entry the result parameter can be the same as either v1 or v2 (or both).

Q3Vector2D_Subtract

You can use the Q3Vector2D_Subtract function to subtract a two-dimensional vector from a two-dimensional vector.

```
TQ3Vector2D *Q3Vector2D_Subtract (
                    const TQ3Vector2D *v1,
                    const TQ3Vector2D *v2,
                    TQ3Vector2D *result);
```

v1	A two-dimensional vector.
v2	A two-dimensional vector.
result	On exit, the result of subtracting v2 from v1.

DESCRIPTION

The Q3Vector2D_Subtract function returns, as its function result and in the result parameter, the two-dimensional vector that is the result of subtracting vector v2 from vector v1. Note that on entry the result parameter can be the same as either v1 or v2 (or both).

Q3Vector3D_Subtract

You can use the `Q3Vector3D_Subtract` function to subtract a three-dimensional vector from a three-dimensional vector.

```
TQ3Vector3D *Q3Vector3D_Subtract (
                    const TQ3Vector3D *v1,
                    const TQ3Vector3D *v2,
                    TQ3Vector3D *result);
```

`v1` A three-dimensional vector.

`v2` A three-dimensional vector.

`result` On exit, the result of subtracting `v2` from `v1`.

DESCRIPTION

The `Q3Vector3D_Subtract` function returns, as its function result and in the `result` parameter, the three-dimensional vector that is the result of subtracting vector `v2` from vector `v1`. Note that on entry the `result` parameter can be the same as either `v1` or `v2` (or both).

Determining Vector Cross Products

QuickDraw 3D provides routines that you can use to calculate cross products of vectors.

Q3Vector2D_Cross

You can use the `Q3Vector2D_Cross` function to determine the cross product of two two-dimensional vectors.

```
float Q3Vector2D_Cross (
                    const TQ3Vector2D *v1,
                    const TQ3Vector2D *v2);
```

v1	A two-dimensional vector.
v2	A two-dimensional vector.

DESCRIPTION

The Q3Vector2D_Cross function returns, as its function result, the cross product of the vectors v1 and v2.

Q3Vector3D_Cross

You can use the Q3Vector3D_Cross function to determine the cross product of two three-dimensional vectors.

```
TQ3Vector3D *Q3Vector3D_Cross (
                    const TQ3Vector3D *v1,
                    const TQ3Vector3D *v2,
                    TQ3Vector3D *result);
```

v1	A three-dimensional vector.
v2	A three-dimensional vector.
result	On exit, the cross product of v1 and v2.

DESCRIPTION

The Q3Vector3D_Cross function returns, as its function result and in the result parameter, the cross product of the vectors v1 and v2.

Q3Point3D_CrossProductTri

You can use the Q3Point3D_CrossProductTri function to determine the cross product of the two vectors defined by three three-dimensional points.

```
TQ3Vector3D *Q3Point3D_CrossProductTri (
                    const TQ3Point3D *point1,
                    const TQ3Point3D *point2,
                    const TQ3Point3D *point3,
                    TQ3Vector3D *crossVector);
```

`point1` A three-dimensional point.

`point2` A three-dimensional point.

`point3` A three-dimensional point.

`crossVector` On exit, the cross product of the two vectors determined by subtracting `point2` from `point1` and `point3` from `point1`.

DESCRIPTION

The `Q3Point3D_CrossProductTri` function returns, as its function result and in the `crossVector` parameter, the cross product of the two vectors determined by subtracting `point2` from `point1` and `point3` from `point2`.

Determining Vector Dot Products

QuickDraw 3D provides routines that you can use to calculate dot (or *scalar*, or *inner*) products of vectors.

Q3Vector2D_Dot

You can use the `Q3Vector2D_Dot` function to determine the dot product of two two-dimensional vectors.

```
float Q3Vector2D_Dot (
                    const TQ3Vector2D *v1,
                    const TQ3Vector2D *v2);
```

`v1` A two-dimensional vector.

`v2` A two-dimensional vector.

DESCRIPTION

The `Q3Vector2D_Dot` function returns, as its function result, a floating-point value that is the dot product of the two vectors `v1` and `v2`.

Q3Vector3D_Dot

You can use the `Q3Vector3D_Dot` function to determine the dot product of two three-dimensional vectors.

```
float Q3Vector3D_Dot (
                    const TQ3Vector3D *v1,
                    const TQ3Vector3D *v2);
```

`v1` A three-dimensional vector.

`v2` A three-dimensional vector.

DESCRIPTION

The `Q3Vector3D_Dot` function returns, as its function result, a floating-point value that is the dot product of the two vectors `v1` and `v2`.

Transforming Points and Vectors

QuickDraw 3D provides routines that you can use to multiply a point or vector by a matrix, thereby applying a transform to that point or vector. QuickDraw 3D also provides routines that you can use to apply a transform to each point in an array of points.

Q3Vector2D_Transform

You can use the `Q3Vector2D_Transform` function to apply a transform to a two-dimensional vector.

```
TQ3Vector2D *Q3Vector2D_Transform (
                    const TQ3Vector2D *vector2D,
                    const TQ3Matrix3x3 *matrix3x3,
                    TQ3Vector2D *result);
```

`vector2D`	A two-dimensional vector.
`matrix3x3`	A 3-by-3 matrix.
`result`	On exit, the vector that is the result of multiplying `vector2D` by `matrix3x3`.

DESCRIPTION

The `Q3Vector2D_Transform` function returns, as its function result and in the `result` parameter, the vector that is the result of multiplying the vector `vector2D` by the matrix transform `matrix3x3`. Note that on entry the `result` parameter can be the same as the `vector2D` parameter.

Q3Vector3D_Transform

You can use the `Q3Vector3D_Transform` function to apply a transform to a three-dimensional vector.

```
TQ3Vector3D *Q3Vector3D_Transform (
                    const TQ3Vector3D *vector3D,
                    const TQ3Matrix4x4 *matrix4x4,
                    TQ3Vector3D *result);
```

`vector3D`	A three-dimensional vector.
`matrix4x4`	A 4-by-4 matrix.
`result`	On exit, the vector that is the result of multiplying `vector3D` by `matrix4x4`.

DESCRIPTION

The `Q3Vector3D_Transform` function returns, as its function result and in the `result` parameter, the vector that is the result of multiplying the vector `vector3D`

by the matrix transform `matrix4x4`. Note that on entry the `result` parameter can be the same as the `vector3D` parameter.

Q3Point2D_Transform

You can use the `Q3Point2D_Transform` function to apply a transform to a two-dimensional point.

```
TQ3Point2D *Q3Point2D_Transform (
                    const TQ3Point2D *point2D,
                    const TQ3Matrix3x3 *matrix3x3,
                    TQ3Point2D *result);
```

`point2D` A two-dimensional point.

`matrix3x3` A 3-by-3 matrix.

`result` On exit, the point that is the result of multiplying `point2D` by `matrix3x3`.

DESCRIPTION

The `Q3Point2D_Transform` function returns, as its function result and in the `result` parameter, the point that is the result of multiplying the point `point2D` by the matrix transform `matrix3x3`. Note that on entry the `result` parameter can be the same as the `point2D` parameter.

Q3Param2D_Transform

You can use the `Q3Param2D_Transform` function to apply a transform to a two-dimensional parametric point.

```
TQ3Param2D *Q3Param2D_Transform (
                    const TQ3Param2D *param2D,
                    const TQ3Matrix3x3 *matrix3x3,
                    TQ3Param2D *result);
```

param2D	A two-dimensional parametric point.
matrix3x3	A 3-by-3 matrix.
result	On exit, the point that is the result of multiplying param2D by matrix3x3.

DESCRIPTION

The Q3Param2D_Transform function returns, as its function result and in the result parameter, the parametric point that is the result of multiplying the parametric point param2D by the matrix transform matrix3x3. Note that on entry the result parameter can be the same as the param2D parameter.

Q3Point3D_Transform

You can use the Q3Point3D_Transform function to apply a transform to a three-dimensional point.

```
TQ3Point3D *Q3Point3D_Transform (
                    const TQ3Point3D *point3D,
                    const TQ3Matrix4x4 *matrix4x4,
                    TQ3Point3D *result);
```

point3D	A three-dimensional point.
matrix4x4	A 4-by-4 matrix.
result	On exit, the point that is the result of multiplying point3D by matrix4x4.

DESCRIPTION

The Q3Point3D_Transform function returns, as its function result and in the result parameter, the point that is the result of multiplying the point point3D by the matrix transform matrix4x4. Note that on entry the result parameter can be the same as the point3D parameter.

Q3RationalPoint4D_Transform

You can use the `Q3RationalPoint4D_Transform` function to apply a transform to a four-dimensional rational point.

```
TQ3RationalPoint4D *Q3RationalPoint4D_Transform (
                    const TQ3RationalPoint4D *point4D,
                    const TQ3Matrix4x4 *matrix4x4,
                    TQ3RationalPoint4D *result);
```

`point4D` A four-dimensional point.

`matrix4x4` A 4-by-4 matrix.

`result` On exit, the point that is the result of multiplying `point4D` by `matrix4x4`.

DESCRIPTION

The `Q3RationalPoint4D_Transform` function returns, as its function result and in the `result` parameter, the point that is the result of multiplying the rational point `point4D` by the matrix transform `matrix4x4`. Note that on entry the `result` parameter can be the same as the `point4D` parameter.

Q3Point3D_To3DTransformArray

You can use the `Q3Point3D_To3DTransformArray` function to apply a transform to each point in an array of three-dimensional points.

```
TQ3Status Q3Point3D_To3DTransformArray (
                    const TQ3Point3D *inVertex,
                    const TQ3Matrix4x4 *matrix,
                    TQ3Point3D *outVertex,
                    long numVertices,
                    unsigned long inStructSize,
                    unsigned long outStructSize);
```

`inVertex` A pointer to an array of three-dimensional points. This is the source array.

`matrix`	A 4-by-4 matrix.
`outVertex`	A pointer to an array of three-dimensional points. This is the destination array.
`numVertices`	The number of vertices.
`inStructSize`	The size of an element in the source array. Effectively, this is the distance, in bytes, between successive points in the source array.
`outStructSize`	The size of an element in the destination array. Effectively, this is the distance, in bytes, between successive points in the destination array.

DESCRIPTION

The `Q3Point3D_To3DTransformArray` function returns, in the `outVertex` parameter, an array of three-dimensional points, each of which is the result of multiplying a point in the `inVertex` array by the matrix transform `matrix`. The `outVertex` array contains the same number of points (that is, vertices) as the `inVertex` array, as specified by the `numVertices` parameter. The `inStructSize` and `outStructSize` parameters specify the sizes of an element in the `inVertex` and `outVertex` arrays, respectively.

Q3Point3D_To4DTransformArray

You can use the `Q3Point3D_To4DTransformArray` function to apply a transform to each point in an array of three-dimensional points, while changing the dimension of each point from three to four dimensions.

```
TQ3Status Q3Point3D_To4DTransformArray (
                    const TQ3Point3D *inVertex,
                    const TQ3Matrix4x4 *matrix,
                    TQ3RationalPoint4D *outVertex,
                    long numVertices,
                    unsigned long inStructSize,
                    unsigned long outStructSize);
```

`inVertex`	A pointer to an array of three-dimensional points. This is the source array.
`matrix`	A 4-by-4 matrix.
`outVertex`	A pointer to an array of four-dimensional points. This is the destination array.
`numVertices`	The number of vertices.
`inStructSize`	The size of an element in the source array. Effectively, this is the distance, in bytes, between successive points in the source array.
`outStructSize`	The size of an element in the destination array. Effectively, this is the distance, in bytes, between successive points in the destination array.

DESCRIPTION

The `Q3Point3D_To4DTransformArray` function returns, in the `outVertex` parameter, an array of four-dimensional points, each of which is the result of changing the dimensionality of a point in the `inVertex` array from three to four and multiplying by the matrix transform `matrix`. The `outVertex` array contains the same number of points (that is, vertices) as the `inVertex` array, as specified by the `numVertices` parameter. The `inStructSize` and `outStructSize` parameters specify the sizes of an element in the `inVertex` and `outVertex` arrays, respectively.

Q3RationalPoint4D_To4DTransformArray

You can use the `Q3RationalPoint4D_To4DTransformArray` function to apply a transform to each point in an array of four-dimensional points.

```
TQ3Status Q3RationalPoint4D_To4DTransformArray (
                    const TQ3RationalPoint4D *inVertex,
                    const TQ3Matrix4x4 *matrix,
                    TQ3RationalPoint4D *outVertex,
                    long numVertices,
                    unsigned long inStructSize,
                    unsigned long outStructSize);
```

`inVertex`	A pointer to an array of four-dimensional points. This is the source array.
`matrix`	A 4-by-4 matrix.
`outVertex`	A pointer to an array of four-dimensional points. This is the destination array.
`numVertices`	The number of vertices.
`inStructSize`	The size of an element in the source array. Effectively, this is the distance, in bytes, between successive points in the source array.
`outStructSize`	The size of an element in the destination array. Effectively, this is the distance, in bytes, between successive points in the destination array.

DESCRIPTION

The `Q3RationalPoint4D_To4DTransformArray` function returns, in the `outVertex` parameter, an array of four-dimensional points, each of which is the result of multiplying a point in the `inVertex` array by the matrix transform `matrix`. The `outVertex` array contains the same number of points (that is, vertices) as the `inVertex` array, as specified by the `numVertices` parameter. The `inStructSize` and `outStructSize` parameters specify the sizes of an element in the `inVertex` and `outVertex` arrays, respectively.

Negating Vectors

QuickDraw 3D provides routines that you can use to negate (or reverse) vectors. The result of negating a vector is a vector having the same magnitude but the opposite direction as the original vector.

Q3Vector2D_Negate

You can use the `Q3Vector2D_Negate` function to negate a two-dimensional vector.

```
TQ3Vector2D *Q3Vector2D_Negate (
                    const TQ3Vector2D *vector2D,
                    TQ3Vector2D *result);
```

`vector2D` A two-dimensional vector.

`result` On exit, the negation of the specified vector.

DESCRIPTION

The `Q3Vector2D_Negate` function returns, as its function result and in the `result` parameter, the vector that is the negation of the vector `vector2D`.

Q3Vector3D_Negate

You can use the `Q3Vector3D_Negate` function to negate a three-dimensional vector.

```
TQ3Vector3D *Q3Vector3D_Negate (
                    const TQ3Vector3D *vector3D,
                    TQ3Vector3D *result);
```

`vector3D` A three-dimensional vector.

`result` On exit, the negation of the specified vector.

DESCRIPTION

The `Q3Vector3D_Negate` function returns, as its function result and in the `result` parameter, the vector that is the negation of the vector `vector3D`.

Converting Points from Cartesian to Polar or Spherical Form

QuickDraw 3D provides routines that you can use to convert two-dimensional points from Cartesian form (x, y) to polar form (r, θ), and vice versa. QuickDraw 3D also provides routines that you can use to convert three-dimensional points from Cartesian form (x, y, z) to spherical form (ρ, θ, ϕ), and vice versa.

Q3Point2D_ToPolar

You can use the `Q3Point2D_ToPolar` function to convert a two-dimensional point from Cartesian form to polar form.

```
TQ3PolarPoint *Q3Point2D_ToPolar (
                    const TQ3Point2D *point2D,
                    TQ3PolarPoint *result);
```

`point2D` A two-dimensional point.

`result` On exit, a polar point.

DESCRIPTION

The `Q3Point2D_ToPolar` function returns, as its function result and in the `result` parameter, a polar point that is the same point as the two-dimensional point specified by the `point2D` parameter.

Q3PolarPoint_ToPoint2D

You can use the `Q3PolarPoint_ToPoint2D` function to convert a polar point to Cartesian form.

```
TQ3Point2D *Q3PolarPoint_ToPoint2D (
                    const TQ3PolarPoint *polarPoint,
                    TQ3Point2D *result);
```

`polarPoint` A polar point.

`result` On exit, a two-dimensional point.

DESCRIPTION

The `Q3PolarPoint_ToPoint2D` function returns, as its function result and in the `result` parameter, the two-dimensional point that is the same point as the polar point specified by the `polarPoint` parameter.

Q3Point3D_ToSpherical

You can use the `Q3Point3D_ToSpherical` function to convert a three-dimensional point from Cartesian form to spherical form.

```
TQ3SphericalPoint *Q3Point3D_ToSpherical (
                    const TQ3Point3D *point3D,
                    TQ3SphericalPoint *result);
```

`point3D` A three-dimensional point.

`result` On exit, a spherical point.

DESCRIPTION

The `Q3Point3D_ToSpherical` function returns, as its function result and in the `result` parameter, a spherical point that is the same point as the three-dimensional point specified by the `point3D` parameter.

Q3SphericalPoint_ToPoint3D

You can use the `Q3SphericalPoint_ToPoint3D` function to convert a spherical point to Cartesian form.

```
TQ3Point3D *Q3SphericalPoint_ToPoint3D (
                    const TQ3SphericalPoint *sphericalPoint,
                    TQ3Point3D *result);
```

`sphericalPoint` A spherical point.

`result` On exit, a three-dimensional point.

DESCRIPTION

The `Q3SphericalPoint_ToPoint3D` function returns, as its function result and in the `result` parameter, the three-dimensional point that is the same point as the spherical point specified by the `sphericalPoint` parameter.

Determining Point Affine Combinations

QuickDraw 3D provides routines that you can use to determine a point that is the affine combination of some given points.

Q3Point2D_AffineComb

You can use the `Q3Point2D_AffineComb` function to determine the two-dimensional point that is the affine combination of an array of points.

```
TQ3Point2D *Q3Point2D_AffineComb (
                    const TQ3Point2D *points2D,
                    const float *weights,
                    unsigned long nPoints,
                    TQ3Point2D *result);
```

`points2D` A pointer to an array of two-dimensional points.

`weights` A pointer to an array of weights. The sum of the weights must be 1.0.

`nPoints` The number of points in the `points2D` array.

`result` On exit, the point that is the affine combination of the points in `points2D` having the weights in the `weights` array.

DESCRIPTION

The `Q3Point2D_AffineComb` function returns, as its function result and in the `result` parameter, the point that is the affine combination of the points in the array `points2D` having the weights in the array `weights`.

Q3Param2D_AffineComb

You can use the `Q3Param2D_AffineComb` function to determine the two-dimensional parametric point that is the affine combination of an array of parametric points.

```
TQ3Param2D *Q3Param2D_AffineComb (
                    const TQ3Param2D *params2D,
                    const float *weights,
                    unsigned long nPoints,
                    TQ3Param2D *result);
```

`params2D` A pointer to an array of two-dimensional parametric points.

`weights` A pointer to an array of weights. The sum of the weights must be 1.0.

`nPoints` The number of points in the `params2D` array.

`result` On exit, the parametric point that is the affine combination of the parametric points in `params2D` having the weights in the `weights` array.

DESCRIPTION

The `Q3Param2D_AffineComb` function returns, as its function result and in the `result` parameter, the parametric point that is the affine combination of the parametric points in the array `params2D` having the weights in the array `weights`.

Q3Point3D_AffineComb

You can use the `Q3Point3D_AffineComb` function to determine the three-dimensional point that is the affine combination of an array of points.

```
TQ3Point3D *Q3Point3D_AffineComb (
                    const TQ3Point3D *points3D,
                    const float *weights,
                    unsigned long nPoints,
                    TQ3Point3D *result);
```

`points3D` A pointer to an array of three-dimensional points.

`weights` A pointer to an array of weights. The sum of the weights must be 1.0.

`nPoints` The number of points in the `points3D` array.

`result` On exit, the point that is the affine combination of the points in `points3D` having the weights in the `weights` array.

DESCRIPTION

The `Q3Point3D_AffineComb` function returns, as its function result and in the `result` parameter, the point that is the affine combination of the points in the array `points3D` having the weights in the array `weights`.

Q3RationalPoint3D_AffineComb

You can use the `Q3RationalPoint3D_AffineComb` function to determine the rational three-dimensional point that is the affine combination of an array of points.

```
TQ3RationalPoint3D *Q3RationalPoint3D_AffineComb (
                    const TQ3RationalPoint3D *points3D,
                    const float *weights,
                    unsigned long nPoints,
                    TQ3RationalPoint3D *result);
```

`points3D` A pointer to an array of rational three-dimensional points.

`weights` A pointer to an array of weights. The sum of the weights must be 1.0.

`nPoints` The number of points in the `points3D` array.

`result` On exit, the point that is the affine combination of the points in `points3D` having the weights in the `weights` array.

DESCRIPTION

The `Q3RationalPoint3D_AffineComb` function returns, as its function result and in the `result` parameter, the rational point that is the affine combination of the points in the array `points3D` having the weights in the array `weights`.

Q3RationalPoint4D_AffineComb

You can use the `Q3RationalPoint4D_AffineComb` function to determine the rational four-dimensional point that is the affine combination of an array of points.

```
TQ3RationalPoint4D *Q3RationalPoint4D_AffineComb (
                    const TQ3RationalPoint4D *points4D,
                    const float *weights,
                    unsigned long nPoints,
                    TQ3RationalPoint4D *result);
```

`points4D` A pointer to an array of rational four-dimensional points.

`weights` A pointer to an array of weights. The weights must sum to 1.0.

`nPoints` The number of points in the `points4D` array.

`result` On exit, the point that is the affine combination of the points in `points4D` which have the weights in the `weights` array.

DESCRIPTION

The `Q3RationalPoint4D_AffineComb` function returns, as its function result and in the `result` parameter, the rational point that is the affine combination of the points in the array `points4D` which have the weights in the array `weights`.

Managing Matrices

QuickDraw 3D provides routines that you can use to perform standard operations on 3-by-3 and 4-by-4 matrices. Each routine performs some operation on one or more source matrices and returns a pointer to the destination matrix in the `result` parameter. Any of the source or destination matrices may be the same matrix. The source matrices are unchanged, unless one of them is also specified as the destination matrix.

Q3Matrix3x3_Copy

You can use the `Q3Matrix3x3_Copy` function to get a copy of a 3-by-3 matrix.

```
TQ3Matrix3x3 *Q3Matrix3x3_Copy (
                    const TQ3Matrix3x3 *matrix3x3,
                    TQ3Matrix3x3 *result);
```

`matrix3x3` A 3-by-3 matrix.

`result` On exit, a copy of `matrix3x3`.

DESCRIPTION

The `Q3Matrix3x3_Copy` function returns, as its function result and in the `result` parameter, a copy of the matrix `matrix3x3`.

Q3Matrix4x4_Copy

You can use the `Q3Matrix4x4_Copy` function to get a copy of a 4-by-4 matrix.

```
TQ3Matrix4x4 *Q3Matrix4x4_Copy (
                    const TQ3Matrix4x4 *matrix4x4,
                    TQ3Matrix4x4 *result);
```

`matrix4x4` A 4-by-4 matrix.

`result` On exit, a copy of `matrix4x4`.

DESCRIPTION

The `Q3Matrix4x4_Copy` function returns, as its function result and in the `result` parameter, a copy of the matrix `matrix4x4`.

Q3Matrix3x3_SetIdentity

You can use the `Q3Matrix3x3_SetIdentity` function to set a 3-by-3 matrix to the identity matrix.

```
TQ3Matrix3x3 *Q3Matrix3x3_SetIdentity (TQ3Matrix3x3 *matrix3x3);
```

`matrix3x3` On exit, the 3-by-3 identity matrix.

DESCRIPTION

The `Q3Matrix3x3_SetIdentity` function returns, as its function result and in the `matrix3x3` parameter, the 3-by-3 identity matrix.

Q3Matrix4x4_SetIdentity

You can use the `Q3Matrix4x4_SetIdentity` function to set a 4-by-4 matrix to the identity matrix.

```
TQ3Matrix4x4 *Q3Matrix4x4_SetIdentity (TQ3Matrix4x4 *matrix4x4);
```

`matrix4x4` On exit, the 4-by-4 identity matrix.

DESCRIPTION

The `Q3Matrix4x4_SetIdentity` function returns, as its function result and in the `matrix4x4` parameter, the 4-by-4 identity matrix.

Q3Matrix3x3_Transpose

You can use the `Q3Matrix3x3_Transpose` function to transpose a 3-by-3 matrix.

```
TQ3Matrix3x3 *Q3Matrix3x3_Transpose (
                    const TQ3Matrix3x3 *matrix3x3,
                    TQ3Matrix3x3 *result);
```

`matrix3x3` A 3-by-3 matrix.

`result` On exit, the transpose of `matrix3x3`.

DESCRIPTION

The `Q3Matrix3x3_Transpose` function returns, as its function result and in the `result` parameter, the transpose of the matrix `matrix3x3`.

Q3Matrix4x4_Transpose

You can use the `Q3Matrix4x4_Transpose` function to transpose a 4-by-4 matrix.

```
TQ3Matrix4x4 *Q3Matrix4x4_Transpose (
                    const TQ3Matrix4x4 *matrix4x4,
                    TQ3Matrix4x4 *result);
```

`matrix4x4` A 4-by-4 matrix.

`result` On exit, the transpose of `matrix4x4`.

DESCRIPTION

The `Q3Matrix4x4_Transpose` function returns, as its function result and in the `result` parameter, the transpose of the matrix `matrix4x4`.

Q3Matrix3x3_Invert

You can use the `Q3Matrix3x3_Invert` function to invert a 3-by-3 matrix.

```
TQ3Matrix3x3 *Q3Matrix3x3_Invert (
                    const TQ3Matrix3x3 *matrix3x3,
                    TQ3Matrix3x3 *result);
```

`matrix3x3` A 3-by-3 matrix.

`result` On exit, the inverse of `matrix3x3`.

DESCRIPTION

The `Q3Matrix3x3_Invert` function returns, as its function result and in the `result` parameter, the inverse of the matrix `matrix3x3`.

Q3Matrix4x4_Invert

You can use the `Q3Matrix4x4_Invert` function to invert a 4-by-4 matrix.

```
TQ3Matrix4x4 *Q3Matrix4x4_Invert (
                    const TQ3Matrix4x4 *matrix4x4,
                    TQ3Matrix4x4 *result);
```

`matrix4x4` A 4-by-4 matrix.

`result` On exit, the inverse of `matrix4x4`.

DESCRIPTION

The `Q3Matrix4x4_Invert` function returns, as its function result and in the `result` parameter, the inverse of the matrix `matrix4x4`.

Q3Matrix3x3_Adjoint

You can use the `Q3Matrix3x3_Adjoint` function to adjoin a 3-by-3 matrix.

```
TQ3Matrix3x3 *Q3Matrix3x3_Adjoint (
                    const TQ3Matrix3x3 *matrix3x3,
                    TQ3Matrix3x3 *result);
```

`matrix3x3` A 3-by-3 matrix.

`result` On exit, the adjoint of `matrix3x3`.

DESCRIPTION

The `Q3Matrix3x3_Adjoint` function returns, as its function result and in the `result` parameter, the adjoint of the matrix `matrix3x3`.

Q3Matrix3x3_Multiply

You can use the `Q3Matrix3x3_Multiply` function to multiply two 3-by-3 matrices.

```
TQ3Matrix3x3 *Q3Matrix3x3_Multiply (
                    const TQ3Matrix3x3 *matrixA,
                    const TQ3Matrix3x3 *matrixB,
                    TQ3Matrix3x3 *result);
```

`matrixA` A 3-by-3 matrix.

`matrixB` A 3-by-3 matrix.

`result` On exit, the product of `matrixA` and `matrixB`.

DESCRIPTION

The `Q3Matrix3x3_Multiply` function returns, as its function result and in the `result` parameter, the product of the two 3-by-3 matrices `matrixA` and `matrixB`.

Q3Matrix4x4_Multiply

You can use the `Q3Matrix4x4_Multiply` function to multiply two 4-by-4 matrices.

```
TQ3Matrix4x4 *Q3Matrix4x4_Multiply (
                    const TQ3Matrix4x4 *matrixA,
                    const TQ3Matrix4x4 *matrixB,
                    TQ3Matrix4x4 *result);
```

`matrixA` A 4-by-4 matrix.

`matrixB` A 4-by-4 matrix.

`result` On exit, the product of `matrixA` and `matrixB`.

DESCRIPTION

The `Q3Matrix4x4_Multiply` function returns, as its function result and in the `result` parameter, the product of the two 4-by-4 matrices `matrixA` and `matrixB`.

Q3Matrix3x3_Determinant

You can use the `Q3Matrix3x3_Determinant` function to get the determinant of a 3-by-3 matrix.

```
float Q3Matrix3x3_Determinant (const TQ3Matrix3x3 *matrix3x3);
```

`matrix3x3` A 3-by-3 matrix.

DESCRIPTION

The `Q3Matrix3x3_Determinant` function returns, as its function result, the determinant of the matrix `matrix3x3`.

Q3Matrix4x4_Determinant

You can use the `Q3Matrix4x4_Determinant` function to get the determinant of a 4-by-4 matrix.

```
float Q3Matrix4x4_Determinant (const TQ3Matrix4x4 *matrix4x4);
```

`matrix4x4` A 4-by-4 matrix.

DESCRIPTION

The `Q3Matrix4x4_Determinant` function returns, as its function result, the determinant of the matrix `matrix4x4`.

Setting Up Transformation Matrices

QuickDraw 3D provides routines that you can use to configure matrices to be used as geometric transformations. You must already have allocated the memory for a matrix before calling one of these routines.

All functions operating on 3-by-3 matrices assume that the resulting transform matrices are to be used to transform only homogeneous two-dimensional data types (such as `TQ3RationalPoint3D`). Similarly, all functions operating on 4-by-4 matrices assume that the resulting transform matrices are to be used to

transform only homogeneous three-dimensional data types (such as `TQ3RationalPoint4D`).

You specify an angle (for example, for `Q3Matrix3x3_SetRotateAboutPoint`) by passing a value that is interpreted in radians. If you prefer to use degrees, QuickDraw 3D provides C language macros that convert radians into degrees.

Q3Matrix3x3_SetTranslate

You can use the `Q3Matrix3x3_SetTranslate` function to configure a 3-by-3 translation transformation matrix.

```
TQ3Matrix3x3 *Q3Matrix3x3_SetTranslate (
                    TQ3Matrix3x3 *matrix3x3,
                    float xTrans,
                    float yTrans);
```

`matrix3x3` A 3-by-3 matrix.

`xTrans` The desired amount of translation along the *x* coordinate axis.

`yTrans` The desired amount of translation along the *y* coordinate axis.

DESCRIPTION

The `Q3Matrix3x3_SetTranslate` function returns, as its function result and in the `matrix3x3` parameter, a transformation matrix that translates an object by the amount `xTrans` along the *x* coordinate axis and by the amount `yTrans` along the *y* coordinate axis.

Q3Matrix3x3_SetScale

You can use the `Q3Matrix3x3_SetScale` function to configure a 3-by-3 scaling transformation matrix.

```
TQ3Matrix3x3 *Q3Matrix3x3_SetScale (
                    TQ3Matrix3x3 *matrix3x3,
                    float xScale,
                    float yScale);
```

matrix3x3	A 3-by-3 matrix.
xScale	The desired amount of scaling along the *x* coordinate axis.
yScale	The desired amount of scaling along the *y* coordinate axis.

DESCRIPTION

The Q3Matrix3x3_SetScale function returns, as its function result and in the matrix3x3 parameter, a scaling matrix that scales an object by the amount xScale along the *x* coordinate axis and by the amount yScale along the *y* coordinate axis.

Q3Matrix3x3_SetRotateAboutPoint

You can use the Q3Matrix3x3_SetRotateAboutPoint function to configure a 3-by-3 rotation transformation matrix.

```
TQ3Matrix3x3 *Q3Matrix3x3_SetRotateAboutPoint (
                    TQ3Matrix3x3 *matrix3x3,
                    const TQ3Point2D *origin,
                    float angle);
```

matrix3x3	A 3-by-3 matrix.
origin	The desired origin of rotation.
angle	The desired angle of rotation, in radians.

DESCRIPTION

The Q3Matrix3x3_SetRotateAboutPoint function returns, as its function result and in the matrix3x3 parameter, a rotation matrix that rotates an object by the angle angle around the point origin.

Q3Matrix4x4_SetTranslate

You can use the `Q3Matrix4x4_SetTranslate` function to configure a 4-by-4 translation transformation matrix.

```
TQ3Matrix4x4 *Q3Matrix4x4_SetTranslate (
                    TQ3Matrix4x4 *matrix4x4,
                    float xTrans,
                    float yTrans,
                    float zTrans);
```

`matrix4x4` A 4-by-4 matrix.

`xTrans` The desired amount of translation along the *x* coordinate axis.

`yTrans` The desired amount of translation along the *y* coordinate axis.

`zTrans` The desired amount of translation along the *z* coordinate axis.

DESCRIPTION

The `Q3Matrix4x4_SetTranslate` function returns, as its function result and in the `matrix4x4` parameter, a transformation matrix that translates an object by the amount `xTrans` along the *x* coordinate axis, by the amount `yTrans` along the *y* coordinate axis, and by the amount `zTrans` along the *z* coordinate axis.

Q3Matrix4x4_SetScale

You can use the `Q3Matrix4x4_SetScale` function to configure a 4-by-4 scaling transformation matrix.

```
TQ3Matrix4x4 *Q3Matrix4x4_SetScale (
                    TQ3Matrix4x4 *matrix4x4,
                    float xScale,
                    float yScale,
                    float zScale);
```

`matrix4x4` A 4-by-4 matrix.

`xScale` The desired amount of scaling along the *x* coordinate axis.

yScale	The desired amount of scaling along the *y* coordinate axis.
zScale	The desired amount of scaling along the *z* coordinate axis.

DESCRIPTION

The Q3Matrix4x4_SetScale function returns, as its function result and in the matrix4x4 parameter, a scaling matrix that scales an object by the amount xScale along the *x* coordinate axis, by the amount yScale along the *y* coordinate axis, and by the amount zScale along the *z* coordinate axis.

Q3Matrix4x4_SetRotateAboutPoint

You can use the Q3Matrix4x4_SetRotateAboutPoint function to configure a 4-by-4 rotation transformation matrix.

```
TQ3Matrix4x4 *Q3Matrix4x4_SetRotateAboutPoint (
                    TQ3Matrix4x4 *matrix4x4,
                    const TQ3Point3D *origin,
                    float xAngle,
                    float yAngle,
                    float zAngle);
```

matrix4x4	A 4-by-4 matrix.
origin	The desired origin of rotation.
xAngle	The desired angle of rotation around the *x* component of origin, in radians.
yAngle	The desired angle of rotation around the *y* component of origin, in radians.
zAngle	The desired angle of rotation around the *z* component of origin, in radians.

DESCRIPTION

The Q3Matrix4x4_SetRotateAboutPoint function returns, as its function result and in the matrix4x4 parameter, a rotation matrix that rotates an object by the specified angle around the point origin.

Q3Matrix4x4_SetRotateAboutAxis

You can use the `Q3Matrix4x4_SetRotateAboutAxis` function to configure a 4-by-4 rotate-about-axis transformation matrix.

```
TQ3Matrix4x4 *Q3Matrix4x4_SetRotateAboutAxis (
                    TQ3Matrix4x4 *matrix4x4,
                    const TQ3Point3D *origin,
                    const TQ3Vector3D *orientation,
                    float angle);
```

`matrix4x4` A 4-by-4 matrix.

`origin` The desired origin of rotation.

`orientation` The desired orientation of the axis of rotation.

`angle` The desired angle of rotation, in radians.

DESCRIPTION

The `Q3Matrix4x4_SetRotateAboutAxis` function returns, as its function result and in the `matrix4x4` parameter, an rotate-about-axis matrix that rotates an object by the angle `angle` around the axis determined by the point `origin` and the orientation `orientation`.

Q3Matrix4x4_SetRotate_X

You can use the `Q3Matrix4x4_SetRotate_X` function to configure a 4-by-4 transformation matrix that rotates objects around the *x* axis.

```
TQ3Matrix4x4 *Q3Matrix4x4_SetRotate_X (
                    TQ3Matrix4x4 *matrix4x4,
                    float angle);
```

`matrix4x4` A 4-by-4 matrix.

`angle` The desired angle of rotation around the *x* coordinate axis, in radians.

DESCRIPTION

The `Q3Matrix4x4_SetRotate_X` function returns, as its function result and in the `matrix4x4` parameter, a rotational matrix that rotates an object by the angle `angle` around the *x* axis.

Q3Matrix4x4_SetRotate_Y

You can use the `Q3Matrix4x4_SetRotate_Y` function to configure a 4-by-4 transformation matrix that rotates objects around the *y* axis.

```
TQ3Matrix4x4 *Q3Matrix4x4_SetRotate_Y (
                    TQ3Matrix4x4 *matrix4x4,
                    float angle);
```

`matrix4x4` A 4-by-4 matrix.

`angle` The desired angle of rotation around the *y* coordinate axis, in radians.

DESCRIPTION

The `Q3Matrix4x4_SetRotate_Y` function returns, as its function result and in the `matrix4x4` parameter, a rotational matrix that rotates an object by the angle `angle` around the *y* axis.

Q3Matrix4x4_SetRotate_Z

You can use the `Q3Matrix4x4_SetRotate_Z` function to configure a 4-by-4 transformation matrix that rotates objects around the *z* axis.

```
TQ3Matrix4x4 *Q3Matrix4x4_SetRotate_Z (
                    TQ3Matrix4x4 *matrix4x4,
                    float angle);
```

`matrix4x4`	A 4-by-4 matrix.
`angle`	The desired angle of rotation around the *z* coordinate axis, in radians.

DESCRIPTION

The `Q3Matrix4x4_SetRotate_Z` function returns, as its function result and in the `matrix4x4` parameter, a rotational matrix that rotates an object by the angle `angle` around the *z* axis.

Q3Matrix4x4_SetRotate_XYZ

You can use the `Q3Matrix4x4_SetRotate_XYZ` function to configure a 4-by-4 transformation matrix that rotates objects around all three coordinate axes.

```
TQ3Matrix4x4 *Q3Matrix4x4_SetRotate_XYZ (
                    TQ3Matrix4x4 *matrix4x4,
                    float xAngle,
                    float yAngle,
                    float zAngle);
```

`matrix4x4`	A 4-by-4 matrix.
`xAngle`	The desired angle of rotation around the *x* axis, in radians.
`yAngle`	The desired angle of rotation around the *y* axis, in radians.
`zAngle`	The desired angle of rotation around the *z* axis, in radians.

DESCRIPTION

The `Q3Matrix4x4_SetRotate_XYZ` function returns, as its function result and in the `matrix4x4` parameter, a rotational matrix that rotates an object by the specified angles around the *x*, *y*, and *z* axes.

Q3Matrix4x4_SetRotateVectorToVector

You can use the `Q3Matrix4x4_SetRotateVectorToVector` function to configure a 4-by-4 transformation matrix that rotates objects around the origin in such a way that a transformed vector matches a given vector.

```
TQ3Matrix4x4 *Q3Matrix4x4_SetRotateVectorToVector (
                    TQ3Matrix4x4 *matrix4x4,
                    const TQ3Vector3D *v1,
                    const TQ3Vector3D *v2);
```

`matrix4x4` A 4-by-4 matrix.

`v1` A three-dimensional vector.

`v2` A three-dimensional vector.

DESCRIPTION

The `Q3Matrix4x4_SetRotateVectorToVector` function returns, as its function result and in the `matrix4x4` parameter, a rotational matrix that rotates objects around the origin in such a way that the transformed vector `v1` matches the vector `v2`. Both `v1` and `v2` should be normalized.

Q3Matrix4x4_SetQuaternion

You can use the `Q3Matrix4x4_SetQuaternion` function to configure a 4-by-4 quaternion transformation matrix.

```
TQ3Matrix4x4 *Q3Matrix4x4_SetQuaternion (
                    TQ3Matrix4x4 *matrix,
                    const TQ3Quaternion *quaternion);
```

`matrix` A 4-by-4 matrix.

`quaternion` A quaternion.

DESCRIPTION

The `Q3Matrix4x4_SetQuaternion` function returns, as its function result and in the `matrix` parameter, a 4-by-4 matrix that represents the quaternion specified by the `quaternion` parameter.

Utility Functions

QuickDraw 3D provides several mathematical utility functions. You can use the following two macros to convert degrees to radians, and vice versa. These functions use the constant `kQ3Pi`, equal to π.

```
#define Q3Math_DegreesToRadians(x)              ((x) * kQ3Pi / 180.0)

#define Q3Math_RadiansToDegrees(x)              ((x) * 180.0 / kQ3Pi)
```

You can use the following two macros to get the minimum and maximum of two values.

```
#define Q3Math_Min(x,y)             ((x) <= (y) ? (x) : (y))

#define Q3Math_Max(x,y)             ((x) >= (y) ? (x) : (y))
```

Managing Quaternions

QuickDraw 3D provides routines that you can use to operate on quaternions.

Q3Quaternion_Set

You can use the `Q3Quaternion_Set` function to set the components of a quaternion.

```
TQ3Quaternion *Q3Quaternion_Set (
                    TQ3Quaternion *quaternion,
                    float w,
                    float x,
                    float y,
                    float z);
```

quaternion	A quaternion.
w	The desired *w* component of a quaternion.
x	The desired *x* component of a quaternion.
y	The desired *y* component of a quaternion.
z	The desired *z* component of a quaternion.

DESCRIPTION

The Q3Quaternion_Set function returns, as its function result and in the quaternion parameter, the quaternion whose components are specified by the w, x, y, and z parameters.

Q3Quaternion_SetIdentity

You can use the Q3Quaternion_SetIdentity function to set a quaternion to the identity quaternion.

```
TQ3Quaternion *Q3Quaternion_SetIdentity (
                    TQ3Quaternion *quaternion);
```

quaternion	On exit, the identity quaternion.

DESCRIPTION

The Q3Quaternion_SetIdentity function returns, as its function result and in the quaternion parameter, the identity quaternion.

Q3Quaternion_Copy

You can use the Q3Quaternion_Copy function to get a copy of a quaternion.

```
TQ3Quaternion *Q3Quaternion_Copy (
                    const TQ3Quaternion *quaternion,
                    TQ3Quaternion *result);
```

`quaternion` A quaternion.

`result` On exit, a copy of `quaternion`.

DESCRIPTION

The `Q3Quaternion_Copy` function returns, as its function result and in the `result` parameter, a copy of the quaternion `quaternion`.

Q3Quaternion_IsIdentity

You can use the `Q3Quaternion_IsIdentity` function to determine whether a quaternion is the identity quaternion.

```
TQ3Boolean Q3Quaternion_IsIdentity (
                    const TQ3Quaternion *quaternion);
```

`quaternion` A quaternion.

DESCRIPTION

The `Q3Quaternion_IsIdentity` function returns `kQ3True` if the `quaternion` parameter is the identity quaternion; `Q3Quaternion_IsIdentity` returns `kQ3False` otherwise.

Q3Quaternion_Invert

You can use the `Q3Quaternion_Invert` function to invert a quaternion.

```
TQ3Quaternion *Q3Quaternion_Invert (
                    const TQ3Quaternion *quaternion,
                    TQ3Quaternion *result);
```

`quaternion` A quaternion.

`result` On exit, the inverse of `quaternion`.

DESCRIPTION

The `Q3Quaternion_Invert` function returns, as its function result and in the `result` parameter, the inverse of the quaternion specified by the `quaternion` parameter.

Q3Quaternion_Normalize

You can use the `Q3Quaternion_Normalize` function to normalize a quaternion.

```
TQ3Quaternion *Q3Quaternion_Normalize (
                    const TQ3Quaternion *quaternion,
                    TQ3Quaternion *result);
```

`quaternion` A quaternion.

`result` On exit, the normalized form of `quaternion`.

DESCRIPTION

The `Q3Quaternion_Normalize` function returns, as its function result and in the `result` parameter, the normalized form of the quaternion `quaternion`. Note that on entry the `result` parameter can be the same as the `quaternion` parameter.

Q3Quaternion_Dot

You can use the `Q3Quaternion_Dot` function to determine the dot product of two quaternions.

```
float Q3Quaternion_Dot (
                    const TQ3Quaternion *q1,
                    const TQ3Quaternion *q2);
```

`q1` A quaternion.

`q2` A quaternion.

DESCRIPTION

The `Q3Quaternion_Dot` function returns, as its function result, a floating-point value that is the dot product of the two quaternions `q1` and `q2`.

Q3Quaternion_Multiply

You can use the `Q3Quaternion_Multiply` function to multiply two quaternions.

```
TQ3Quaternion *Q3Quaternion_Multiply (
                    const TQ3Quaternion *q1,
                    const TQ3Quaternion *q2,
                    TQ3Quaternion *result);
```

`q1`	A quaternion.
`q2`	A quaternion.
`result`	On exit, the product of `q1` and `q2`.

DESCRIPTION

The `Q3Quaternion_Multiply` function returns, as its function result and in the `result` parameter, the product of the two quaternions `q1` and `q2`.

If you want to rotate an object by the quaternion `qFirst` and then rotate the resulting object by the quaternion `qSecond`, you can accomplish both rotations at once by applying the quaternion `qResult` that is obtained as follows:

```
Q3Quaternion_Multiply(qSecond, qFirst, qResult);
```

Note the order of the quaternion multiplicands.

Q3Quaternion_SetRotateAboutAxis

You can use the `Q3Quaternion_SetRotateAboutAxis` function to configure a rotate-about-axis quaternion.

```
TQ3Quaternion *Q3Quaternion_SetRotateAboutAxis (
                    TQ3Quaternion *quaternion,
                    const TQ3Vector3D *axis,
                    float angle);
```

`quaternion`	A quaternion.
`axis`	The desired axis of rotation.
`angle`	The desired angle of rotation, in radians.

DESCRIPTION

The `Q3Quaternion_SetRotateAboutAxis` function returns, as its function result and in the `quaternion` parameter, a rotate-about-axis quaternion that rotates an object by the angle `angle` around the axis specified by the `axis` parameter.

Q3Quaternion_SetRotate_X

You can use the `Q3Quaternion_SetRotate_X` function to configure a quaternion that rotates objects around the *x* axis.

```
TQ3Quaternion *Q3Quaternion_SetRotate_X (
                    TQ3Quaternion *quaternion,
                    float angle);
```

`quaternion`	A quaternion.
`angle`	The desired angle of rotation around the *x* coordinate axis, in radians.

DESCRIPTION

The `Q3Quaternion_SetRotate_X` function returns, as its function result and in the `quaternion` parameter, a quaternion that rotates an object by the angle `angle` around the *x* axis.

Q3Quaternion_SetRotate_Y

You can use the `Q3Quaternion_SetRotate_Y` function to configure a quaternion that rotates objects around the *y* axis.

```
TQ3Quaternion *Q3Quaternion_SetRotate_Y (
                    TQ3Quaternion *quaternion,
                    float angle);
```

`quaternion` A quaternion.

`angle` The desired angle of rotation around the *y* coordinate axis, in radians.

DESCRIPTION

The `Q3Quaternion_SetRotate_Y` function returns, as its function result and in the `quaternion` parameter, a quaternion that rotates an object by the angle `angle` around the *y* axis.

Q3Quaternion_SetRotate_Z

You can use the `Q3Quaternion_SetRotate_Z` function to configure a quaternion that rotates objects around the *z* axis.

```
TQ3Quaternion *Q3Quaternion_SetRotate_Z (
                    TQ3Quaternion *quaternion,
                    float angle);
```

`quaternion` A quaternion.

`angle` The desired angle of rotation around the *z* coordinate axis, in radians.

DESCRIPTION

The `Q3Quaternion_SetRotate_Z` function returns, as its function result and in the `quaternion` parameter, a quaternion that rotates an object by the angle `angle` around the *z* axis.

Q3Quaternion_SetRotate_XYZ

You can use the `Q3Quaternion_SetRotate_XYZ` function to configure a quaternion having a specified rotation around the *x*, *y*, and *z* axes.

```
TQ3Quaternion *Q3Quaternion_SetRotate_XYZ (
                    TQ3Quaternion *quaternion,
                    float xAngle,
                    float yAngle,
                    float zAngle);
```

`quaternion` A quaternion.

`xAngle` The desired angle of rotation around the *x* axis, in radians.

`yAngle` The desired angle of rotation around the *y* axis, in radians.

`zAngle` The desired angle of rotation around the *z* axis, in radians.

DESCRIPTION

The `Q3Quaternion_SetRotate_XYZ` function returns, as its function result and in the `quaternion` parameter, a quaternion that rotates an object by the specified angles around the *x*, *y*, and *z* axes.

Q3Quaternion_SetMatrix

You can use the `Q3Quaternion_SetMatrix` function to configure a quaternion from a matrix.

```
TQ3Quaternion *Q3Quaternion_SetMatrix (
                    TQ3Quaternion *quaternion,
                    const TQ3Matrix4x4 *matrix);
```

`quaternion` A quaternion.

`matrix` A 4-by-by matrix.

DESCRIPTION

The `Q3Quaternion_SetMatrix` function returns, as its function result and in the `quaternion` parameter, a quaternion that has the same transformational properties as the matrix specified by the `matrix` parameter.

Q3Quaternion_SetRotateVectorToVector

You can use the `Q3Quaternion_SetRotateVectorToVector` function to configure a quaternion that rotates objects around the origin in such a way that a transformed vector matches a given vector.

```
TQ3Quaternion *Q3Quaternion_SetRotateVectorToVector (
                    TQ3Quaternion *quaternion,
                    const TQ3Vector3D *v1,
                    const TQ3Vector3D *v2);
```

`quaternion`	A quaternion.
`v1`	A three-dimensional vector.
`v2`	A three-dimensional vector.

DESCRIPTION

The `Q3Quaternion_SetRotateVectorToVector` function returns, as its function result and in the `quaternion` parameter, a quaternion that rotates objects around the origin in such a way that the transformed vector `v1` matches the vector `v2`. Both `v1` and `v2` should be normalized.

Q3Quaternion_MatchReflection

You can use the `Q3Quaternion_MatchReflection` function to match the orientation of a quaternion.

```
TQ3Quaternion *Q3Quaternion_MatchReflection (
                    const TQ3Quaternion *q1,
                    const TQ3Quaternion *q2,
                    TQ3Quaternion *result);
```

`q1`	A quaternion.
`q2`	A quaternion.
`result`	On exit, a quaternion that is either `q1` or the negative of `q1`, and that matches the orientation of `q2`.

DESCRIPTION

The `Q3Quaternion_MatchReflection` function returns, as its function result and in the `result` parameter, a quaternion that is either identical to the quaternion specified by the `q1` parameter or is the negative of `q1`, depending on whether `q1` or its negative matches the orientation of the quaternion specified by the `q2` parameter.

Q3Quaternion_InterpolateFast

You can use the `Q3Quaternion_InterpolateFast` function to interpolate quickly between two quaternions.

```
TQ3Quaternion *Q3Quaternion_InterpolateFast (
                    const TQ3Quaternion *q1,
                    const TQ3Quaternion *q2,
                    float t,
                    TQ3Quaternion *result);
```

`q1`	A quaternion.
`q2`	A quaternion.
`t`	An interpolation factor. This parameter should contain a value between 0.0 and 1.0.
`result`	On exit, a quaternion that is a fast interpolation between the two specified quaternions.

DESCRIPTION

The `Q3Quaternion_InterpolateFast` function returns, as its function result and in the `result` parameter, a quaternion that interpolates between the two quaternions specified by the `q1` and `q2` parameters, according to the factor specified by the `t` parameter. If the value of `t` is 0.0, `Q3Quaternion_InterpolateFast` returns a quaternion identical to `q1`. If the value of `t` is 1.0, `Q3Quaternion_InterpolateFast` returns a quaternion identical to `q2`. If `t` is any other value in the range [0.0, 1.0], `Q3Quaternion_InterpolateFast` returns a quaternion that is interpolated between the two quaternions.

The interpolation returned by `Q3Quaternion_InterpolateFast` is not as smooth or constant as that returned by `Q3Quaternion_InterpolateLinear`, but `Q3Quaternion_InterpolateFast` is usually faster than `Q3Quaternion_InterpolateLinear`.

Q3Quaternion_InterpolateLinear

You can use the `Q3Quaternion_InterpolateLinear` function to interpolate linearly between two quaternions.

```
TQ3Quaternion *Q3Quaternion_InterpolateLinear (
                    const TQ3Quaternion *q1,
                    const TQ3Quaternion *q2,
                    float t,
                    TQ3Quaternion *result) ;
```

`q1` A quaternion.

`q2` A quaternion.

`t` An interpolation factor. This parameter should contain a value between 0.0 and 1.0.

`result` On exit, a quaternion that is a smooth and constant interpolation between the two specified quaternions.

DESCRIPTION

The `Q3Quaternion_InterpolateLinear` function returns, as its function result and in the `result` parameter, a quaternion that interpolates smoothly between the

two quaternions specified by the `q1` and `q2` parameters, according to the factor specified by the `t` parameter. If the value of `t` is 0.0, `Q3Quaternion_InterpolateLinear` returns a quaternion identical to `q1`. If the value of `t` is 1.0, `Q3Quaternion_InterpolateLinear` returns a quaternion identical to `q2`. If `t` is any other value in the range [0.0, 1.0], `Q3Quaternion_InterpolateLinear` returns a quaternion that is interpolated between the two quaternions in a smooth and constant manner.

Q3Vector3D_TransformQuaternion

You can use the `Q3Vector3D_TransformQuaternion` function to transform a vector by a quaternion.

```
TQ3Vector3D *Q3Vector3D_TransformQuaternion (
                    const TQ3Vector3D *vector,
                    const TQ3Quaternion *quaternion,
                    TQ3Vector3D *result);
```

`vector`	A three-dimensional vector.
`quaternion`	A quaternion.
`result`	On exit, a three-dimensional vector that is the result of transforming the specified vector by the specified quaternion.

DESCRIPTION

The `Q3Vector3D_TransformQuaternion` function returns, as its function result and in the `result` parameter, a three-dimensional vector that is the result of transforming the vector specified by the `vector` parameter using the quaternion specified by the `quaternion` parameter.

Q3Point3D_TransformQuaternion

You can use the `Q3Point3D_TransformQuaternion` function to transform a point by a quaternion.

```
TQ3Point3D *Q3Point3D_TransformQuaternion (
                    const TQ3Point3D *point,
                    const TQ3Quaternion *quaternion,
                    TQ3Point3D *result);
```

`point` A three-dimensional point.

`quaternion` A quaternion.

`result` On exit, a three-dimensional point that is the result of transforming the specified point by the specified quaternion.

DESCRIPTION

The `Q3Point3D_TransformQuaternion` function returns, as its function result and in the `result` parameter, a three-dimensional point that is the result of transforming the point specified by the `point` parameter using the quaternion specified by the `quaternion` parameter.

Managing Bounding Boxes

QuickDraw 3D provides routines that you can use to manage bounding boxes.

Q3BoundingBox_Copy

You can use the `Q3BoundingBox_Copy` function to make a copy of a bounding box.

```
TQ3BoundingBox *Q3BoundingBox_Copy (
                    const TQ3BoundingBox *src,
                    TQ3BoundingBox *dest);
```

`src` A pointer to the bounding box to be copied.

`dest` On entry, a pointer to a buffer large enough to hold a bounding box. On exit, a pointer to a copy of the bounding box specified by the `src` parameter.

DESCRIPTION

The `Q3BoundingBox_Copy` function returns, as its function result and in the `dest` parameter, a copy of the bounding box specified by the `src` parameter. `Q3BoundingBox_Copy` does not allocate any memory for the destination bounding box; the `dest` parameter must point to space allocated in the heap or on the stack before you call `Q3BoundingBox_Copy`.

Q3BoundingBox_Union

You can use the `Q3BoundingBox_Union` function to find the union of two bounding boxes.

```
TQ3BoundingBox *Q3BoundingBox_Union (
                    const TQ3BoundingBox *v1,
                    const TQ3BoundingBox *v2,
                    TQ3BoundingBox *result);
```

`v1`	A pointer to a bounding box.
`v2`	A pointer to a bounding box.
`result`	On exit, a pointer to the union of the bounding boxes `v1` and `v2`.

DESCRIPTION

The `Q3BoundingBox_Union` function returns, as its function result and in the `result` parameter, a pointer to the bounding box that is the union of the two bounding boxes specified by the parameters `v1` and `v2`. The `result` parameter can point to the memory occupied by either `v1` or `v2`, thereby performing the union operation in place.

Q3BoundingBox_Set

You can use the `Q3BoundingBox_Set` function to set the defining points of a bounding box.

```
TQ3BoundingBox *Q3BoundingBox_Set (
                    TQ3BoundingBox *bBox,
                    const TQ3Point3D *min,
                    const TQ3Point3D *max,
                    TQ3Boolean isEmpty);
```

`bBox`	A pointer to a bounding box.
`min`	A pointer to a three-dimensional point.
`max`	A pointer to a three-dimensional point.
`isEmpty`	A Boolean value that indicates whether the specified bounding box is empty (`kQ3True`) or not (`kQ3False`).

DESCRIPTION

The `Q3BoundingBox_Set` function assigns the values `min` and `max` to the `min` and `max` fields of the bounding box specified by the `bBox` parameter. `Q3BoundingBox_Set` also assigns the value of the `isEmpty` parameter to the `isEmpty` field of the bounding box.

Q3BoundingBox_UnionPoint3D

You can use the `Q3BoundingBox_UnionPoint3D` function to find the union of a bounding box and a three-dimensional point.

```
TQ3BoundingBox *Q3BoundingBox_UnionPoint3D (
                    const TQ3BoundingBox *bBox,
                    const TQ3Point3D *pt3D,
                    TQ3BoundingBox *result);
```

`bBox`	A pointer to a bounding box.
`pt3D`	A three-dimensional point.
`result`	On exit, a pointer to the union of the specified bounding box and the specified point.

DESCRIPTION

The `Q3BoundingBox_UnionPoint3D` function returns, as its function result and in the `result` parameter, a pointer to the bounding box that is the union of the bounding box specified by the `bBox` parameter and the three-dimensional point specified by the `pt3D` parameter. The `result` parameter can point to the memory pointed to by `bBox`, thereby performing the union operation in place.

Q3BoundingBox_UnionRationalPoint4D

You can use the `Q3BoundingBox_UnionRationalPoint4D` function to find the union of a bounding box and a rational four-dimensional point.

```
TQ3BoundingBox *Q3BoundingBox_UnionRationalPoint4D (
                    const TQ3BoundingBox *bBox,
                    const TQ3RationalPoint4D *pt4D,
                    TQ3BoundingBox *result);
```

`bBox` A pointer to a bounding box.

`pt4D` A rational four-dimensional point.

`result` On exit, a pointer to the union of the specified bounding box and the specified point.

DESCRIPTION

The `Q3BoundingBox_UnionRationalPoint4D` function returns, as its function result and in the `result` parameter, a pointer to the bounding box that is the union of the bounding box specified by the `bBox` parameter and the rational four-dimensional point specified by the `pt4D` parameter. The `result` parameter can point to the memory pointed to by `bBox`, thereby performing the union operation in place.

Q3BoundingBox_SetFromPoints3D

You can use the `Q3BoundingBox_SetFromPoints3D` function to find the bounding box that bounds an arbitrary list of three-dimensional points.

```
TQ3BoundingBox *Q3BoundingBox_SetFromPoints3D (
                    TQ3BoundingBox *bBox,
                    const TQ3Point3D *pts,
                    unsigned long nPts,
                    unsigned long structSize);
```

`bBox`	A pointer to a bounding box.
`pts`	A pointer to a list of three-dimensional points.
`nPts`	The number of points in the specified list.
`structSize`	The number of bytes of data that separate two successive points in the specified list of points.

DESCRIPTION

The `Q3BoundingBox_SetFromPoints3D` function returns, as its function result and in the `bBox` parameter, a pointer to a bounding box that contains all the points in the list of three-dimensional points specified by the `pts` parameter. The `nPts` parameter indicates how many points are in that list, and the `structSize` parameter indicates the offset between any two successive points in the list. By suitably specifying the value of the `structSize` parameter, you can have QuickDraw 3D extract points that are embedded in an array of larger data structures.

Q3BoundingBox_SetFromRationalPoints4D

You can use the `Q3BoundingBox_SetFromRationalPoints4D` function to find the bounding box that bounds an arbitrary list of rational four-dimensional points.

```
TQ3BoundingBox *Q3BoundingBox_SetFromRationalPoints4D (
                    TQ3BoundingBox *bBox,
                    const TQ3RationalPoint4D *pts,
                    unsigned long nPts,
                    unsigned long structSize);
```

`bBox`	A pointer to a bounding box.
`pts`	A pointer to a list of rational four-dimensional points.

nPts	The number of points in the specified list.
structSize	The number of bytes of data that separate two successive points in the specified list of points.

DESCRIPTION

The Q3BoundingBox_SetFromRationalPoints4D function returns, as its function result and in the bBox parameter, a pointer to a bounding box that contains all the points in the list of rational four-dimensional points specified by the pts parameter. The nPts parameter indicates how many points are in that list, and the structSize parameter indicates the offset between any two successive points in the list. By suitably specifying the value of the structSize parameter, you can have QuickDraw 3D extract points that are embedded in an array of larger data structures.

Managing Bounding Spheres

QuickDraw 3D provides routines that you can use to manage bounding spheres.

Q3BoundingSphere_Copy

You can use the Q3BoundingSphere_Copy function to make a copy of a bounding sphere.

```
TQ3BoundingSphere *Q3BoundingSphere_Copy (
                    const TQ3BoundingSphere *src,
                    TQ3BoundingSphere *dest);
```

src	A pointer to the bounding sphere to be copied.
dest	On entry, a pointer to a buffer large enough to hold a bounding sphere. On exit, a pointer to a copy of the bounding sphere specified by the src parameter.

DESCRIPTION

The `Q3BoundingSphere_Copy` function returns, as its function result and in the `dest` parameter, a copy of the bounding sphere specified by the `src` parameter. `Q3BoundingSphere_Copy` does not allocate any memory for the destination bounding sphere; the `dest` parameter must point to space allocated in the heap or on the stack before you call `Q3BoundingSphere_Copy`.

Q3BoundingSphere_Union

You can use the `Q3BoundingSphere_Union` function to find the union of two bounding spheres.

```
TQ3BoundingSphere *Q3BoundingSphere_Union (
                    const TQ3BoundingSphere *s1,
                    const TQ3BoundingSphere *s2,
                    TQ3BoundingSphere *result);
```

`s1`	A pointer to a bounding sphere.
`s2`	A pointer to a bounding sphere.
`result`	On exit, a pointer to the union of the bounding spheres `s1` and `s2`.

DESCRIPTION

The `Q3BoundingSphere_Union` function returns, as its function result and in the `result` parameter, a pointer to the bounding sphere that is the union of the two bounding spheres specified by the parameters `s1` and `s2`. The `result` parameter can point to the memory occupied by either `s1` or `s2`, thereby performing the union operation in place.

Q3BoundingSphere_Set

You can use the `Q3BoundingSphere_Set` function to set the defining origin and radius of a bounding sphere.

```
TQ3BoundingSphere *Q3BoundingSphere_Set (
                    TQ3BoundingSphere *bSphere,
                    const TQ3Point3D *origin,
                    float radius,
                    TQ3Boolean isEmpty);
```

`bSphere` A pointer to a bounding sphere.

`origin` A pointer to a three-dimensional point.

`radius` A floating-point value that specifies the desired radius of the bounding sphere.

`isEmpty` A Boolean value that indicates whether the specified bounding sphere is empty (`kQ3True`) or not (`kQ3False`).

DESCRIPTION

The `Q3BoundingSphere_Set` function assigns the values `origin` and `radius` to the `origin` and `radius` fields of the bounding sphere specified by the `bSphere` parameter. `Q3BoundingSphere_Set` also assigns the value of the `isEmpty` parameter to the `isEmpty` field of the bounding sphere.

Q3BoundingSphere_UnionPoint3D

You can use the `Q3BoundingSphere_UnionPoint3D` function to find the union of a bounding sphere and a three-dimensional point.

```
TQ3BoundingSphere *Q3BoundingSphere_UnionPoint3D (
                    const TQ3BoundingSphere *bSphere,
                    const TQ3Point3D *pt3D,
                    TQ3BoundingSphere *result);
```

`bSphere` A pointer to a bounding sphere.

`pt3D` A three-dimensional point.

`result` On exit, a pointer to the union of the specified bounding sphere and the specified point.

DESCRIPTION

The `Q3BoundingSphere_UnionPoint3D` function returns, as its function result and in the `result` parameter, a pointer to the bounding sphere that is the union of the bounding sphere specified by the `bSphere` parameter and the three-dimensional point specified by the `pt3D` parameter. The `result` parameter can point to the memory pointed to by `bSphere`, thereby performing the union operation in place.

Q3BoundingSphere_UnionRationalPoint4D

You can use the `Q3BoundingSphere_UnionRationalPoint4D` function to find the union of a bounding sphere and a rational four-dimensional point.

```
TQ3BoundingSphere *Q3BoundingSphere_UnionRationalPoint4D (
                    const TQ3BoundingSphere *bSphere,
                    const TQ3RationalPoint4D *pt4D,
                    TQ3BoundingSphere *result);
```

`bSphere` A pointer to a bounding sphere.

`pt4D` A rational four-dimensional point.

`result` On exit, a pointer to the union of the specified bounding sphere and the specified point.

DESCRIPTION

The `Q3BoundingSphere_UnionRationalPoint4D` function returns, as its function result and in the `result` parameter, a pointer to the bounding sphere that is the union of the bounding sphere specified by the `bSphere` parameter and the rational four-dimensional point specified by the `pt4D` parameter. The `result` parameter can point to the memory pointed to by `bSphere`, thereby performing the union operation in place.

Q3BoundingSphere_SetFromPoints3D

You can use the `Q3BoundingSphere_SetFromPoints3D` function to find the bounding sphere that bounds an arbitrary list of three-dimensional points.

```
TQ3BoundingSphere *Q3BoundingSphere_SetFromPoints3D (
                    TQ3BoundingSphere *bSphere,
                    const TQ3Point3D *pts,
                    unsigned long nPts,
                    unsigned long structSize);
```

`bSphere` A pointer to a bounding sphere.

`pts` A pointer to a list of three-dimensional points.

`nPts` The number of points in the specified list.

`structSize` The number of bytes of data that separate two successive points in the specified list of points.

DESCRIPTION

The `Q3BoundingSphere_SetFromPoints3D` function returns, as its function result and in the `bSphere` parameter, a pointer to a bounding sphere that contains all the points in the list of three-dimensional points specified by the `pts` parameter. The `nPts` parameter indicates how many points are in that list, and the `structSize` parameter indicates the offset between any two successive points in the list. By suitably specifying the value of the `structSize` parameter, you can have QuickDraw 3D extract points that are embedded in an array of larger data structures.

Q3BoundingSphere_SetFromRationalPoints4D

You can use the `Q3BoundingSphere_SetFromRationalPoints4D` function to find the bounding sphere that bounds an arbitrary list of rational four-dimensional points.

```
TQ3BoundingSphere *Q3BoundingSphere_SetFromRationalPoints4D (
                    TQ3BoundingSphere *bSphere,
                    const TQ3RationalPoint4D *pts,
                    unsigned long nPts,
                    unsigned long structSize);
```

`bSphere`	A pointer to a bounding sphere.
`pts`	A pointer to a list of rational four-dimensional points.
`nPts`	The number of points in the specified list.
`structSize`	The number of bytes of data that separate two successive points in the specified list of points.

DESCRIPTION

The `Q3BoundingSphere_SetFromRationalPoints4D` function returns, as its function result and in the `bSphere` parameter, a pointer to a bounding sphere that contains all the points in the list of rational four-dimensional points specified by the `pts` parameter. The `nPts` parameter indicates how many points are in that list, and the `structSize` parameter indicates the offset between any two successive points in the list. By suitably specifying the value of the `structSize` parameter, you can have QuickDraw 3D extract points that are embedded in an array of larger data structures.

CHAPTER 20

Mathematical Utilities

CHAPTER 21

Color Utilities

This chapter describes the QuickDraw 3D Color Utilities, a set of functions that you can use to manage colors. You can use these functions to develop distinctive color schemes for the user interface elements of your application.

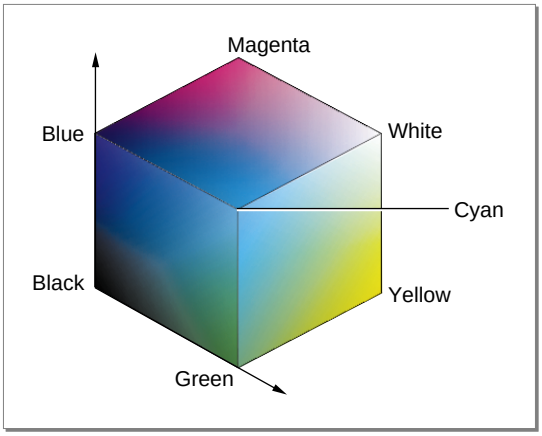
About the Color Utilities

QuickDraw 3D provides a set of utility routines that you can use to manage colors. You can use these routines to add, subtract, scale, interpolate, and perform other operations on colors. These utilities are intended to facilitate the creation of distinctive color schemes (that is, sets of correlated colors) for user interface elements in your application. You can, however, use these routines to manage colors anywhere in your application.

See the chapter "Pointing Device Manager" for complete information on creating and manipulating color schemes.

QuickDraw 3D supports one color space, the **RGB color space** defined by three color component values (one each for red, green, and blue). The RGB color space can be visualized as a cube, as in Figure 21-1, with corners of black, the three primary colors (red, green, and blue), the three secondary colors (cyan, magenta, and yellow), and white.

Figure 21-1 RGB color space



You specify a single color in the RGB color space by filling in a structure of type `TQ3ColorRGB`:

```
typedef struct TQ3ColorRGB {
    float           r;                  /*red component*/
    float           g;                  /*green component*/
    float           b;                  /*blue component*/
} TQ3ColorRGB;
```

The QuickDraw 3D Color utilities all operate on structures of type `TQ3ColorRGB`. Each field in an `TQ3ColorRGB` structure should contain a value in the range 0.0 to 1.0, inclusive.

Using the QuickDraw 3D Color Utilities

You can use the `Q3ColorRGB_Set` function to set the fields of an RGB color structure. For example, to specify the color white, you can call `Q3ColorRGB_Set` as shown in Listing 21-1.

Listing 21-1 Specifying the color white

```
TQ3ColorRGB             myColor;

Q3ColorRGB_Set(&myColor, 1.0, 1.0, 1.0);
```

Most of the QuickDraw 3D Color Utilities operate on two existing colors and return a third color. For example, you can call the `Q3ColorRGB_Add` function to add together two colors, as shown in Listing 21-2.

Listing 21-2 Adding two colors

```
TQ3ColorRGB             myColor1, myColor2, myResult;
TQ3ColorRGB             *myResultPtr;

myResultPtr = Q3ColorRGB_Add(&myColor1, &myColor2, &myResult);
```

As you can see, `Q3ColorRGB_Add` returns the address of the resulting RGB color structure both in the `myResult` parameter and as its function result. This allows you to nest calls to the QuickDraw 3D Color Utilities in function calls, as follows:

```
Q3ColorRGB_Add(Q3ColorRGB_Add(&myColor1, &myColor2, &myResult),
                   &myColor3, &myResult);
```

This line of code adds the colors specified by the `myColor1` and `myColor2` parameters and adds that sum to the color specified by the `myColor3` parameter. If this line of code completes successfully, the parameter `myResult` is a pointer to an RGB color structure that contains the sum of all three colors.

QuickDraw 3D Color Utilities Reference

This section describes the color utilities provided by QuickDraw 3D, as well as the basic color data structures.

Data Structures

This section describes the data structures that you use to specify colors.

Color Structures

You use an **RGB color structure** to specify a color. The RGB color structure is defined by the `TQ3ColorRGB` data type.

```
typedef struct TQ3ColorRGB {
    float           r;                  /*red component*/
    float           g;                  /*green component*/
    float           b;                  /*blue component*/
} TQ3ColorRGB;
```

Field descriptions

`r` The red component of the color. The value in this field should be between 0.0 and 1.0.

`g` The green component of the color. The value in this field should be between 0.0 and 1.0.

`b` The blue component of the color. The value in this field should be between 0.0 and 1.0.

You use an **ARGB color structure** to specify a color together with an alpha channel. The ARGB color structure is defined by the `TQ3ColorARGB` data type.

```
typedef struct TQ3ColorARGB {
    float           a;                  /*alpha channel*/
    float           r;                  /*red component*/
    float           g;                  /*green component*/
    float           b;                  /*blue component*/
} TQ3ColorARGB;
```

Field descriptions

`a` The alpha channel of the color. The value in this field should be between 0.0 (transparent) and 1.0. (solid).

`r` The red component of the color. The value in this field should be between 0.0 and 1.0.

`g` The green component of the color. The value in this field should be between 0.0 and 1.0.

b	The blue component of the color. The value in this field should be between 0.0 and 1.0.

QuickDraw 3D Color Utilities

This section describes the QuickDraw 3D utilities you can use to handle colors. Because most of these routines return a pointer to an RGB color structure both as a function result and through the result parameter, you can nest these routines.

Q3ColorRGB_Set

You can use the Q3ColorRGB_Set function to set the fields of an RGB color structure.

```
TQ3ColorRGB *Q3ColorRGB_Set (
                    TQ3ColorRGB *color,
                    float r,
                    float g,
                    float b);
```

color	On exit, a pointer to an RGB color structure.
r	The red component of the color.
g	The green component of the color.
b	The blue component of the color.

DESCRIPTION

The Q3ColorRGB_Set function returns, as its function result and in the color parameter, a pointer to an RGB color structure whose fields contain the values in the r, g, and b parameters.

Q3ColorARGB_Set

You can use the `Q3ColorARGB_Set` function to set the fields of an ARGB color structure.

```
TQ3ColorARGB *Q3ColorARGB_Set (
                    TQ3ColorARGB *color,
                    float a,
                    float r,
                    float g,
                    float b);
```

`color`	On exit, a pointer to an ARGB color structure.
`a`	The alpha channel of the color.
`r`	The red component of the color.
`g`	The green component of the color.
`b`	The blue component of the color.

DESCRIPTION

The `Q3ColorARGB_Set` function returns, as its function result and in the `color` parameter, a pointer to an ARGB color structure whose fields contain the values in the `a`, `r`, `g`, and `b` parameters.

Q3ColorRGB_Add

You can use the `Q3ColorRGB_Add` function to add two colors.

```
TQ3ColorRGB *Q3ColorRGB_Add (
                    const TQ3ColorRGB *c1,
                    const TQ3ColorRGB *c2,
                    TQ3ColorRGB *result);
```

`c1`	An RGB color structure.
`c2`	An RGB color structure.

`result` On exit, a pointer to an RGB color structure for the color that is the sum of the two specified colors.

DESCRIPTION

The `Q3ColorRGB_Add` function returns, as its function result and in the `result` parameter, a pointer to an RGB color structure that represents the sum of the colors specified by the `c1` and `c2` parameters.

Q3ColorRGB_Subtract

You can use the `Q3ColorRGB_Subtract` function to subtract one color from another.

```
TQ3ColorRGB *Q3ColorRGB_Subtract (
                    const TQ3ColorRGB *c1,
                    const TQ3ColorRGB *c2,
                    TQ3ColorRGB *result);
```

`c1` An RGB color structure.

`c2` An RGB color structure.

`result` On exit, a pointer to an RGB color structure for the color that is the difference of the two specified colors.

DESCRIPTION

The `Q3ColorRGB_Subtract` function returns, as its function result and in the `result` parameter, a pointer to an RGB color structure that represents the result of subtracting the color specified by the `c2` parameter from the color specified by the `c1` parameter.

Q3ColorRGB_Scale

You can use the `Q3ColorRGB_Scale` function to scale a color.

```
TQ3ColorRGB *Q3ColorRGB_Scale (
                    const TQ3ColorRGB *color,
                    float scale,
                    TQ3ColorRGB *result);
```

`color` An RGB color structure.

`scale` A scaling factor.

`result` On exit, a pointer to an RGB color structure for the color that is the scale of the specified color.

DESCRIPTION

The `Q3ColorRGB_Scale` function returns, as its function result and in the `result` parameter, a pointer to an RGB color structure that represents the result of scaling the color specified by the `color` parameter by the factor specified by the `scale` parameter.

Q3ColorRGB_Clamp

You can use the `Q3ColorRGB_Clamp` function to clamp a color.

```
TQ3ColorRGB *Q3ColorRGB_Clamp (
                    const TQ3ColorRGB *color,
                    TQ3ColorRGB *result);
```

`color` An RGB color structure.

`result` On exit, a pointer to an RGB color structure for the color that is the clamped version of the specified color.

DESCRIPTION

The `Q3ColorRGB_Clamp` function returns, as its function result and in the `result` parameter, a pointer to an RGB color structure that clamps each component of the color specified by the `color` parameter. A clamped component lies between 0.0 and 1.0, inclusive.

Q3ColorRGB_Lerp

You can use the `Q3ColorRGB_Lerp` function to interpolate two colors linearly.

```
TQ3ColorRGB *Q3ColorRGB_Lerp (
                    const TQ3ColorRGB *first,
                    const TQ3ColorRGB *last,
                    float alpha,
                    TQ3ColorRGB *result);
```

`first` An RGB color structure.

`last` An RGB color structure.

`alpha` An alpha value.

`result` On exit, a pointer to an RGB color structure for the color that is the linear interpolation, by the specified alpha value, of the two specified colors.

DESCRIPTION

The `Q3ColorRGB_Lerp` function returns, as its function result and in the `result` parameter, a pointer to an RGB color structure that is linearly interpolated between the two colors specified by the `first` and `last` parameters. The `alpha` parameter specifies the desired alpha value for the interpolation.

Q3ColorRGB_Accumulate

You can use the `Q3ColorRGB_Accumulate` function to accumulate colors.

```
TQ3ColorRGB *Q3ColorRGB_Accumulate (
                    const TQ3ColorRGB *src,
                    TQ3ColorRGB *result);
```

`src` An RGB color structure.

`result` On entry, an RGB color structure. On exit, a pointer to an RGB color structure for the color that is the result of adding the source color to the result color.

DESCRIPTION

The `Q3ColorRGB_Accumulate` function returns, as its function result and in the `result` parameter, a pointer to an RGB color structure that is the result of adding the color specified by the `src` parameter to the color specified by the `result` parameter.

Q3ColorRGB_Luminance

You can use the `Q3ColorRGB_Luminance` function to compute the luminance of a color.

```
float *Q3ColorRGB_Luminance (
                    const TQ3ColorRGB *color,
                    float *luminance);
```

`color` An RGB color structure.

`luminance` On exit, the luminance of the specified color.

DESCRIPTION

The `Q3ColorRGB_Luminance` function returns, as its function result and in the `luminance` parameter, the luminance of the color specified by the `color` parameter. A color's luminance is computed using this formula:

$$luminance = (0.30078125 \times \texttt{color.r}) + (0.58984375 \times \texttt{color.g}) + (0.109375 \times \texttt{color.b})$$

CHAPTER 21

Color Utilities

3D Metafile 1.5 Reference

This document describes the 3D Metafile, a file format designed to permit the storage and interchange of 3D data.

▲ **WARNING**
This information in this document is preliminary and is subject to change. ▲

Introduction

The 3D Metafile is a file format for 3D graphics applications that makes use of the Apple QuickDraw 3D (QD3D) 1.5 graphics library or other 3D graphics libraries. This document describes the 3D Metafile file format, Version 1.5.

The purpose of the metafile is to establish a standard file format for 3D graphics files. This includes establishing canonical forms for descriptions of familiar 3D graphics objects.

This standard is put forward to promote compatibility among 3D graphics applications and is meant to facilitate the transfer and exchange of data between distinct applications. The file format also permits a project to be saved to a file so that it may be resumed or altered at a later time.

The canonical forms for descriptions of 3D graphics objects outlined in this document embody an object– and class–based approach to 3D graphics. The 3D metafile objects are defined using a small number of basic data types and some object formation devices. Each object is a member of a class; the class structure reflects the structure of the QuickDraw 3D class hierarchy. (But it is worth pointing out that QD3D also supports immediate mode, which does not require creation of objects, and QD3D can write objects in immediate mode.)

Each class of objects, and thus each object, is correlated with a particular node in that structure. We use the terms *parent* and *child* to describe the relationships among objects located at immediately adjacent and connected nodes in the structure. For example, a color attribute may be included in a set of attributes that is assigned to a geometric object. In that case, the geometric object is a parent of the attribute set, which in turn is a parent of the color attribute, while the color attribute is a child of the attribute set, which in turn is a child of the geometric object. See the book *3D Graphics Programming With QuickDraw 3D* for complete details on this approach to the classification of 3D graphics objects.

A metafile is simply a sequence or list of one or more valid metafile objects. Each metafile must contain exactly one 3D metafile header, and this header must be the first object to occur in the file. Objects following the header may occur in any order permitted by the metafile class hierarchy. Currently, every object that begins in a metafile must be wholly contained in that file; thus, it is not legal to truncate the description of an object at the end of a file.

Note
For examples of complete 3D metafiles, see "Polyhedra," beginning on page 1331, and "Attribute Arrays," beginning on page 1350. ◆

A metafile object's data can take two forms:

- data that is itself another metafile object
- data that is not another metafile object

Collectively the 2nd type of data makes up the **root** object. The data in a root object is some combination of the basic data types (see "Basic Data Types," beginning on page 1261). If a metafile object contains other metafile objects (called **subobjects**), then the entire object is enclosed in a **container**. The first item in the container is the root object, and the subobjects take up the rest of the container. See "Containers," beginning on page 1292.

This document defines a format for ASCII text files and also defines a format for binary files. The two formats incorporate the same functional features, and there is a close correspondence between their components. Most objects are represented very similarly in the two formats. However, some objects, such as file pointers, are represented differently, as described below. Any text metafile can be converted to a binary metafile, and vice versa, without loss of information.

The metafile file format permits objects to be labeled and referenced: if the same object appears more than once in a metafile, only the first occurrence need be written out fully. All other occurrences take the form of a reference to that first occurrence. The referencing machinery makes use of three special entities: file pointers, reference objects, and table of contents objects. A table of contents provides a complete or partial catalog of the items contained in a metafile. For details, see "File Pointers," beginning on page 1272, "Reference Objects," beginning on page 1285, and "Tables of Contents," beginning on page 1279.

Note that a metafile is not itself a database and does not have the capabilities of a database. Applications that wish to apply the capabilities of a database to the contents of a metafile must connect that file to a preexisting database program.

If desired, the objects in a metafile can be organized by adding them to groups. Almost all objects can be added to one or another of the available groups. Groups are themselves objects, and they can be nested, so that more complex objects can be created that have as much hierarchical structure as desired. See "Groups," beginning on page 1473.

The strategy of exposition is as follows: we begin with the basic data types. From these we define the defined data types. This is followed by a discussion of abstract (or structural) data types, which are part of the structure of a metafile object. Next is a section containing descriptions of six special metafile objects. This is followed by a section that contains examples illustrating the structure of metafile objects, especially the cross-referencing machinery. The remainder of this document describes the remaining metafile objects.

Basic Data Types

All metafile object specifications, including specifications of custom objects you define yourself, must use only the following basic data types. All other data types are defined from these. This means that your read and write code need only contain routines to read and write these data types.

If you are using Apple QuickDraw 3D, special metafile types corresponding to the first ten types below are declared in QD3DIO.h. (`TQ3Uns8`, `TQ3Uns16`, etc.) QD3DIO.h also contains prototypes for routines to read and write each of these types. Additional basic data types may be introduced in the future if the need for them arises.

Unsigned Integer Data Types

`Uns8`	An unsigned 8-bit integer.
`Uns16`	An unsigned 16-bit integer.
`Uns32`	An unsigned 32-bit integer.
`Uns64`	An unsigned 64-bit integer.

In binary metafiles, `Uns64` is a 64-bit quantity; in text metafiles, it is represented by the following struct:

```
typedef struct Uns64 {
Uns32     hi
Uns32     lo
} Uns64
```

Signed Integer Data Types

`Int8`	A signed 8-bit integer.
`Int16`	A signed 16-bit integer.
`Int32`	A signed 32-bit integer.
`Int64`	A signed 64-bit integer.

In binary metafiles, `Int64` is a 64-bit quantity; in text metafiles, it is represented by the following struct:

```
typedef struct Int64 {
Int32     hi
Uns32     lo
} Int64
```

Floating-Point Integer Data Types

`Float32`	A single-precision 32-bit floating-point number.
`Float64`	A double-precision 64-bit floating-point number.

Floating point numbers must be represented as specified by the IEEE floating-point standard (IEEE 754). For details, consult the standard itself. If you are working with Power Macintosh computers, the book *Inside Macintosh: PowerPC Numerics* may be useful.

Strings

In a text file, a string is a sequence of ASCII text symbols enclosed in double quotation marks.

Only the following escape sequences may occur in a text file string:

`\n`	newline
`\r`	carriage return
`\t`	horizontal tab
`\v`	vertical tab
`\\`	backslash
`\'`	single quotation mark
`\"`	double quotation mark

In a binary file, a string is represented by a string of zero-terminated padded characters. The size of a string in a binary file is determined as follows (note that we add 1 to the length to account for the terminating \0):

```
len = strlen(string) + 1;
remainder = len % 4;
pad = ( (remainder > 0) ? (4 - remainder) : 0 );
size = len + pad;
```

Raw Data

Raw data is used to store information that is platform-dependent or is inherently not alphanumeric.

In a text file, raw data is stored as hexadecimal strings prefixed by the characters `'0x'`. Strings of raw data are not padded in text files. However, your application may pad them if you wish.

In a binary file, raw data is stored as sequences of bytes, padded to a 4-byte boundary. The size of raw data is computed as follows (the computation differs from that for string in that raw data doesn't require a terminating \0):

```
remainder = radDataSize % 4;
pad = ( (remainder > 0) ? (4 - remainder) : 0 );
size = radDataSize + pad;
```

Symbolic Constants

The metafile format for many 3DMF objects includes fields that can take values from a predefined range of constants. In the C programming language, such constants would be defined by means of enumerations or #defines. (For examples, see the QD3D interface files.) In C, this means that a symbolic constant is associated with a particular numerical value. This kind of correlation between symbolic constant and numerical value is also maintained in the metafile, in the following way: the symbolic constant appears in the text metafile, and the corresponding numerical value appears in the binary metafile.

The symbolic constants that appear in text metafiles may include either text characters or digits, but may not include blank spaces or punctuation marks. The logical 'or' symbol (|) is used to catenate symbolic constants that function as bitfields.

The corresponding numbers that appear in binary metafiles take the form of unsigned 32-bit integers such as (in hexadecimal) `0x00000001`, `0x0000000E`.

3DMF contains two basic symbolic constants, `Boolean` and `Switch`. The metafile representation of an enumerated `Boolean` type is:

Text	**Binary**
`False`	0x00000000
`True`	0x00000001

The metafile representation of an enumerated `Switch` type is:

Text	**Binary**
`Off`	0x00000000
`On`	0x00000001

Defined 3D Data Types

The following 3D data types are defined using the basic data types described in the previous section. These definitions are obviously convenient when giving the specifications below. But it's good to keep in mind that these types are not distinguished in the metafile format itself. For example, a three-dimensional point in a metafile is just three successive 32 bit quantities; the metafile contains no grouping device that would indicate that these 96 bits make up a point. Given the metafile itself, one may not be able to tell whether these are three

Float32, three Int32, or three Uns32. In a binary metafile, additionally, one would not be able to tell whether these are six Int16, twelve Uns8, etc. And in a text metafile, additionally, one may not be able to tell whether these are three Int16 or three Uns8.
Also, Macintosh QuickDraw 3D does not have distinguished metafile types for the defined data types in this section. QD3DIO.h does contain routines to read and write many of them, but they takes as arguments the regular data types used in the core API—for example, TQ3Point2D.

Two-Dimensional Points

```
typedef struct Point2D {
    Float32                     x;
    Float32                     y;
} Point2D;

typedef struct DPoint2D {
    Float64                     x;
    Float64                     y;
} DPoint2D;
```

Three-Dimensional Points

```
typedef struct Point3D {
    Float32                     x;
    Float32                     y;
    Float32                     z;
} Point3D;

typedef struct DPoint3D {
    Float64                     x;
    Float64                     y;
    Float64                     z;
} DPoint3D;
```

Three-Dimensional Rational Points

```
typedef struct RationalPoint3D {
    Float32                     x;
    Float32                     y;
    Float32                     w;
} RationalPoint3D;

typedef struct DRationalPoint3D {
    Float64                     x;
    Float64                     y;
    Float64                     w;
} DRationalPoint3D;
```

Four-Dimensional Rational Points

```
typedef struct RationalPoint4D {
    Float32                     x;
    Float32                     y;
    Float32                     z;
    Float32                     w;
} RationalPoint4D;

typedef struct DRationalPoint4D {
    Float64                     x;
    Float64                     y;
    Float64                     z;
    Float64                     w;
} DRationalPoint4D;
```

Note
Three– and four–dimensional points are used to represent two– and three–dimensional points respectively in homogeneous coordinate systems. ◆

Color Data Types

```
typedef struct RGBColor {
    Float32                     red;
    Float32                     green;
    Float32                     blue;
} RGBColor;

typedef struct ARGBColor {
    Float32                     alpha;
    Float32                     red;
    Float32                     green;
    Float32                     blue;
} ARGBColor;
```

IMPORTANT

The values in the fields of a color data type must lie in the closed interval [0, 1]. 0 is the minimum value; 1 is the maximum value. ▲

The 3D metafile currently supports only the RGB (red, green, blue) color model (as opposed to color models such as HSV, LAB, etc.).

Two-Dimensional Vectors

```
typedef struct Vector2D {
    Float32                     x;
    Float32                     y;
} Vector2D;

typedef struct DVector2D {
    Float64                     x;
    Float64                     y;
} DVector2D;
```

Three-Dimensional Vectors

```
typedef struct Vector3D {
    Float32                     x;
    Float32                     y;
    Float32                     z;
} Vector3D;

typedef struct DVector3D {
    Float64                     x;
    Float64                     y;
    Float64                     z;
} DVector3D;
```

Parameterizations

```
typedef struct Param2D {
    Float32       u;
    Float32       v;
} Param2D;

typedef struct Param3D {
    Float32       u;
    Float32       v;
    Float32       w;
} Param3D;

typedef struct DParam2D {
    Float64       u;
    Float64       v;
} DParam2D;
```

```
typedef struct DParam3D {
    Float64       u;
    Float64       v;
    Float64       w;
} DParam3D;
```

Tangents

```
typedef struct Tangent2D {
    Vector3D                    uTangent;
    Vector3D                    vTangent;
} Tangent2D;

typedef struct Tangent3D {
    Vector3D                    uTangent;
    Vector3D                    vTangent;
    Vector3D                    wTangent;
} Tangent3D;

typedef struct DTangent2D {
    DVector3D                   uTangent;
    DVector3D                   vTangent;
} DTangent2D;

typedef struct DTangent3D {
    DVector3D                   uTangent;
    DVector3D                   vTangent;
    DVector3D                   wTangent;
} DTangent3D;
```

Matrices

```
typedef Float32 Matrix3x3           [3][3];

typedef Float32 Matrix4x4           [4][4];
```

```
typedef Float64 DMatrix3x3          [3][3];

typedef Float64 DMatrix4x4          [4][4];
```

Abstract Data Types

The 3D Metafile file format defines the following three **abstract** (more accurately, *structural*) data types: object type, size, and file pointer. They are called structural because they are part of the structure needed to represent objects.

Object Type

Every metafile object has a type. In a text file, an object type is expressed by a character string, such as `Polygon`. In a binary file, an object type is expressed by a 4-byte code, such as `plyg`. In both text and binary files, every object specification begins with an object type.

The metafile file format allows you to introduce new types of custom objects. A new type can be introduced anywhere in a file, so long as its format, and the locations of its occurrences, meet the basic conditions described in this document. For Version 1.5 of the Metafile specification, in binary files only, if the binary 4-byte code is <= -1, then the first occurrence of the 4-byte code must be preceded by a `Type` object. The `Type` object establishes a correlation between the character string encoding of the type and the 4-byte code. In Version 1.5, the 4-byte code is dynamically allocated on a per-metafile basis, and may vary from metafile to metafile. It is the character string encoding of the type that remains fixed. See “Types,” beginning on page 1290.

The following section contains an example of an object type.

Size

Size fields appear only in binary metafiles. They specify the size (i.e. the extent) of an object, and so determine its end. In a text file, the extent of an object is determined by parentheses; (specifies its beginning and) its end. Here’s an example of a text file:

```
Polygon (                       # object type
   3                            # number of vertices
   0 0 0                        # first vertex
   1 0 0                        # second vertex
   0 1 0                        # third vertex
)
```

This polygon can be viewed as a structure having two fields. The value in the first field is an unsigned 32-bit integer and the value in the second field is an array of three three-dimensional points. The size of an unsigned 32-bit integer is 4 bytes and the size of a three-dimensional point is 12 bytes, so the size of the above polygon is 40 bytes.

In a binary file, an object specification begins with a 4-byte type code. That is immediately followed by 4 bytes that specify the size of the object. The size does not include the 4-byte type code, nor the 4-byte size specification itself. Thus, the size of the above polygon is 40 bytes, not 44 or 48.

The above polygon would be specified in a binary file as follows:

```
00: 706C6967      plyg          # object type
04: 00000028      40            # object size
08: 00000003      3             # number of vertices
0A: 00000000      0.0           # x coordinate of first vertex
10: 00000000      0.0           # y coordinate of first vertex
14: 00000000      0.0           # z coordinate of first vertex
18: 3F800000      1.0           # x coordinate of second vertex
.
.
```

An object may be of size 0. In a text file, an object of size 0 is described by the type-identifying string followed by a pair of empty parentheses. For example, `AttributeSet ( )` specifies an object of size 0. Some objects have a defined default specification. If such an object is represented as being of size 0, it is understood that the default specification is intended. A binary metafile object of size 0 consists of its 4-byte type ID, followed by the 4-byte size specification (which has value 0) and nothing else.

All binary metafile objects are padded to 4-byte boundaries; thus, the size of an object is always a multiple of 4.

File Pointers

DESCRIPTION

A metafile file pointer indicates the location of another object in that metafile, to which it points. A file pointer and the object to which it points (called the **target object**) must occur in the same file. A target object may occur before or after an associated file pointer in a metafile. A file pointer may fail to have a target object; such a file pointer is *null*. File pointers may occur both in ASCII text metafiles and in binary metafiles. A file pointer is neither declared nor initialized; it is identified as such by the positions in which it may appear and (in a text file) by the type of expression used to represent it.

In a binary metafile, a file pointer is an unsigned 64-bit integer that specifies the address or location of its target object in the metafile. The generator of a binary metafile must determine the number of bytes by which the beginning of the target object is offset from the beginning of the header in order to write the correct value of a pointer to that object. (The beginning of the target object is the begining of its 4-byte type identifier.) A metafile generator must update that file pointer whenever any new objects are inserted between the beginning of the header and the beginning of the target object.

Note
A file pointer is offset relative to the beginning of the 3DMF header of the metafile in which it occurs, not relative to the end of the header . ◆

In an ASCII text metafile, a file pointer is represented by a character string composed of at least two characters, the last of which is a right angle bracket (>). Thus `p>` and `Arrow>` are file pointers; `p`, `>`, and `Arrow` are not. In a text file, the target object of a file pointer must bear a label corresponding to that file pointer. The label corresponding to a file pointer is the result of omitting the final right angle bracket from the string representing that file pointer. For example, the label corresponding to `string5>` is `string5`. Such a label is always followed immediately by a colon and then, on a new line, by the target object:

```
string5:
targetobject
```

Each file pointer may correspond to at most one label, and each label may correspond to at most one file pointer. Metafiles of type `normal` will typically contain labels that are not being pointed to by a file pointer, because the code

that writes the metafile must write a label for every shared object. It must do this because it cannot predict whether the object about to be written will be written a second time (at which point it will be written as a reference and so will have to make use of a file pointer to that label).

Two types of file pointers may occur in a metafile, corresponding to two types of target object a file pointer may have. The target object of a file pointer of the first type is a table of contents; such a file pointer is meant to indicate the location of a table of contents and serves no other purpose. A file pointer of this type must occur in the fourth field of each header. A file pointer of this type must also occur in the first field of each table of contents; this pointer points to the location of the next subsequent table of contents, if one exists. A file pointer of this type may occur in no other position.

The target object of a file pointer of the second type must be either an object of type shared, or a container with root object of type shared. (To determine whether an object is of type shared, check its description in this document.) The root object of a container may not be the target object of a file pointer. The purpose of a file pointer of this type is to enable the metafile writer to make repeated reference to a target object without repeating that object's definition. (A file pointer of this type may occur only in the second field of a table of contents entry; thus, a metafile that contains file pointers of this type must include at least one table of contents.) The way in which repeated reference to an object is accomplished through the use of file pointers of this type is explained in the next paragraph.

An application may permit a user to make reference in one context to an object specified in another context. The 3DMF specification supports both reference to another object in the same metafile, and reference to an object in another metafile. We use the term **external reference** whenever we wish to talk about the second of these; the first is simply termed a **reference**. This section discusses references; external references are discussed in "External Reference Objects," beginning on page 1286.

A typical case of using references is to repeatedly use the same geometry but make it appear in different locations through appropriate uses of transforms (an example is given in "Examples of Metafile Structures," beginning on page 1295). Another case is to apply the same texture shader to several different objects, or different faces of objects.

In a metafile, reference to objects involves several components: a file pointer, a target object, an integer, an entry in a table of contents, and a special metafile object called a **reference object.** (In a text file, the label that immediately

precedes the object pointed to by the file pointer must also be present.) The object to be referenced must be the target object of a file pointer. That file pointer must appear together with an appropriately chosen integer, called a **reference ID**, in an entry in a table of contents located in the file containing the target object. (If that file contains no table of contents, then a table of contents must be created.) The reference ID associated with that file pointer must be the data of a reference object, one occurrence of which must be placed at each position at which the target object is to be referenced. In a text file, a reference object looks like this:

```
Reference ( 2 )
```

The target object, file pointer, table of contents, and reference object must all occur in the same file. There may be at most one file pointer to any target object; thus, once a reference ID has been associated with a pointer to a target object in a table of contents entry, that `refID` is the only integer that may be used to reference that target object. If a metafile contains a reference object with a reference ID that does not appear in the table of contents, then obviously the reference cannot be resolved and the read call on the reference object should return `NULL`. Many metafile readers would view this as an error in reading, so metafiles containing reference objects with unresolved `refID` values are regarded as incorrect.

Clearly, a metafile reader must be programmed to recognize and to respond appropriately to reference objects, tables of contents, file pointers (and perhaps labels) and not to confuse them with other types of objects. As noted, a metafile may contain file pointers and labels that are idle. A metafile reader cannot determine whether a file pointer or label is idle by inspection of that object alone, so it must be able to read these objects whether or not they're idle.

EXAMPLE

Here is an example of the legal use of file pointers in an ASCII text metafile:

Note
In this and other examples, the text metafile is followed by its equivalent binary metafile. ◆

```
3DMetafile ( 1 5 Normal tableofcontents0> )# header; includes pointer to TOC
    line2:                                  # label
    Line (                                  # target object
        0 0 0 1 0 0 )
    translate3:
    Translate ( 0 1 0 )
    Reference ( 1 )                         # reference object with refID
    tableofcontents0:                       # label for TOC
    TableOfContents (
        tableofcontents1>                   # next TOC; may be idle
        2                                   # reference seed
        -1                                  # typeSeed
        1                                   # tocEntryType
        16                                  # tocEntrySize
        1                                   # nEntries
        1 line2>                            # TOC entry; includes refID,
                                            # filePtr, Line# and type
                                            # identifier
    )
```

Offset	Hexadecimal code	ASCII
0000	3344 4D46 2020 2010 2001 2005 2020 2020	3DMF············
0010	2020 2020 2020 2058 6C69 6E65 2020 2018	·······Xline····
0020	2020 2020 2020 2020 2020 2020 3F80 2020	···········?···
0030	2020 2020 2020 2020 7472 6E73 2020 200C	·····trns····
0040	2020 2020 3F80 2020 2020 2020 7266 726E	····?·····rfrn
0050	2020 2004 2020 2001 746F 6320 2020 202C	·····toc····,
0060	2020 2020 2020 2020 2020 2002 FFFF FFFF	·····
0070	2020 2001 2020 2010 2020 2001 2020 2001	·····
0080	2020 2020 2020 2018 6C69 6E65	·····line

The file pointer `line2>` is used to place its target object within the scope of a `Translate` object; thus, it adds to the model a copy of the original line that's been transformed by a translation.

Metafile Object Specifications

The following sections contain descriptions of all currently valid metafile objects. Each section concerns a particular type of metafile object, and indicates the required form of specification for objects of that type in text files and in binary files. Each section also includes an example of a valid text file object specification and other pertinent information.

Special Metafile Objects

This section describes six special metafile objects: headers, tables of contents, reference objects, external reference objects, types, and containers.

3D Metafile Header

LABELS

ASCII	`3DMetafile`
Binary	`3DMF (= 0x33444D46)`

METAFILE FLAGS

Each metafile header includes a flag that indicates the uses to which file pointers and reference objects are put in that metafile. The left column below gives the form of the flag found in text metafiles, while the right colun gives the form found in binary metafiles.

`Normal`	`0x00000000`
`Stream`	`0x00000001`
`Database`	`0x00000002`

Constant descriptions

`Normal` This flag indicates that objects in the metafile can be instantiated by reference, using the mechanism of file

pointers and reference objects described above. The table of contents contains only entries for objects that actually have at least one instantiation by reference. (Note that normal metafiles are not prohibited from having the same full description of an object occur in two different places in the file.) In order to read a normal metafile, a parser should have random access to that file.

`Stream` This flag indicates that there are no internal references in the metafile. Objects cannot be instantiated by reference; the complete specification of an object must occur at each place in the file at which that object is to be instantiated. In order to read a stream metafile, a parser need have sequential access only.

`Database` This flag indicates that every shared object in the metafile that is not itself a reference object is the target object of a file pointer appearing in a table of contents in the metafile. That is, every object that could be instantiated by reference and is not itself a reference object must be listed in a table of contents (whether or not that object has actually been instantiated by reference in this file). All of the shared contents of a database metafile may be discovered by a parser through examination of its tables of contents. Note that an object can be both stream and database; this means that there are no reference objects but the metafile contains a complete table of contents for the shared objects.

DATA FORMAT

```
Uns16           majorVersion
Uns16           minorVersion
MetafileFlags   flags
FilePointer     tocLocation
```

Field descriptions

`majorVersion` The version number of the metafile. Currently, the version number is 1.

`minorVersion` The revision number of the metafile. Currently, the revision number is 5.

`flags` The metafile header flag.

`tocLocation`	A file pointer to the location (in the metafile) of a table of contents object. If the value in this field is `NULL`, then the entire metafile must be parsed in order to find any extant tables of contents.

DATA SIZE

20

DESCRIPTION

A metafile header is a structure having four fields. The first two fields specify the version and revision numbers of the metafile. The third field contains a flag indicating the type of the metafile (normal, stream, or database). The fourth field contains a pointer to the location of a table of contents for the metafile. A metafile header in a file indicates that the file is a metafile and provides some information about its contents.

Each metafile must contain exactly one metafile header, and this header must precede every other object in that file. Though each metafile header contains a pointer to the location of a table of contents, there need be no corresponding table of contents in the metafile.

PARENT HIERARCHY

3DMF.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
3DMetafile ( 1 5 Normal tableofcontents0> )
    box2:
    Box (
        0 0 1                   # orientation
        1 0 0                   # majorAxis
        0 0 0                   # minorAxis
        0 1 0                   # origin
    )
```

Offset	Hexadecimal code	ASCII
0000	3344 4D46 2020 2010 2001 2005 2020 2020	3DMF····· ·····
0010	2020 2020 2020 2020 626F 7820 2020 2030	····box····0
0020	2020 2020 2020 2020 3F80 2020 3F80 2020	····?···?···
0030	2020 2020 2020 2020 2020 2020 2020 2020	···· ···· ····
0040	2020 2020 2020 2020 3F80 2020 2020 2020	····?·····

Tables of Contents

LABELS

ASCII	`TableOfContents`
Binary	`toc (= 0x746F6320)`

DATA TYPE DEFINITION: TOC ENTRY TYPE 0

```
TOCEntry (
    Uns32                   refID
    FilePointer             objLocation
)
```

SIZE

12

DATA TYPE DEFINITION: TOC ENTRY TYPE 1

```
TOCEntry (
    Uns32                   refID
    FilePointer             objLocation
    ObjectType              objType
)
```

SIZE

16

Field descriptions

`refID`	The value of the `refID` field of a reference object.
`objLocation`	A pointer to the location of a metafile object that can be referenced.
`objType`	The type identifier of the target object of the file pointer listed in the `objLocation` field. In a text metafile, this field should appear on a separate line.

Note

Type 1 TOC entries allow a parser to determine the type of a referenced object by inspecting tables of contents; type 0 TOC entries do not. Because QD3D 1.5 contains a new call (`Q3File_GetExternalReferences`) that makes use of this feature, in QD3D 1.5 the table of contents in both normal and database metafiles are written as type 1. In QD3D 1.0, the TOC entries in a normal metafile were written as type 0; the TOC entries in a database metafile were written as type 1. ◆

DATA FORMAT

```
FilePointer             nextTOC
Uns32                   refSeed
Int32                   typeSeed
Uns32                   tocEntryType
```

```
Uns32                   tocEntrySize
Uns32                   nEntries
TOCEntry                tocEntries[nEntries]
```

Field descriptions

`nextTOC` A pointer to the location of the next table of contents in the metafile. (If there is no subsequent table of contents, then this pointer is idle. In a text file this means there is a file pointer but no label that resolves it. In a binary file this means that the file pointer is zero.)

`refSeed` The least integer that may occur in the `refID` field of a reference object added to the metafile after this table of contents is written. The value in this field must be greater than 0 and is incremented whenever a new reference object is added to the preceding section of the metafile or is listed in a TOC entry added to this table of contents.

`typeSeed` The greatest integer that may occur in the `typeID` field of a type object added to the metafile after this table of contents is written. The value in this field must be less than 0 and is decremented whenever a new type object is added to the preceding section of the metafile.

`tocEntryType` A numerical constant that indicates the type of the entries contained in the table of contents. The permitted values of this field are 0 and 1. A value of 0 indicates that all entries in the array `tocEntries[]` are of type 0; a value of 1 indicates that all entries in that array are of type 1. The occurrence of this constant should cause no confusion, as all entries in any particular table of contents must be of the same type.

`tocEntrySize` A numerical constant that indicates the binary sizes of the entries contained in the table of contents. The permitted values of this field are 12 and 16. If the value in the previous field is 0, then the value in this field must be 12; if the value in the previous field is 1, then the value in this field must be 16. Again, this constant should cause no confusion, as all entries in any particular table of contents must be of the same size.

`nEntries`	The number of entries contained in the table of contents; that is, the size of the array `tocEntries[]`. If the value in this field is 0, then that array is empty.
`tocEntries[]`	An array of `TOCEntry` objects, all of which are of the same entry type.

DATA SIZE

```
20 + (tocEntrySize * nEntries)
```

DESCRIPTION

A table of contents is a structure that provides a record of associations between reference IDs and file pointers. These associations are specified by the TOC entries of the table of contents. A metafile reader must use its tables of contents to discover linkages between reference objects and file pointers, as there is no other record of those associations. See "File Pointers," beginning on page 1272 and "Reference Objects," beginning on page 1285 for complete details regarding these objects.

A metafile that contains a reference to another object (by means of a reference object) must include at least one table of contents.

If a metafile contains more than one table of contents, then each table of contents should continue the record provided by the immediately previous table of contents (if such exists) without duplication. A table of contents may contain information about objects occurring before or after it or both, but should not contain information about any object that either precedes an object mentioned in a previous table of contents or follows an object mentioned in a subsequent table of contents.

PARENT HIERARCHY

3DMF.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
3DMetafile ( 1 5 Database
       tableofcontents0> )
    box2:
    Container (
       Box (
          0 0 1                        # orientation
          1 0 0                        # majorAxis
          0 0 0                        # minorAxis
          0 1 0                        # origin
       )
       attributeset3:
       Container (
          AttributeSet ( )
          DiffuseColor ( 0.9 0.9 0.2 )
       )
    )
    translate4:
    Translate ( 3 0 0 )
    Reference ( 1 )
    translate5:
    Translate ( 0 3 0 )
    Reference ( 1 )
    translate6:
    Translate ( -3 1 0 )
    Reference ( 1 )
    tableofcontents0:
    TableOfContents (
       tableofcontents1>                  # next TOC
       6                                  # reference seed
       -1                                 # typeSeed
       1                                  # tocEntryType
       16                                 # tocEntrySize
```

```
        5                                   # nEntries
        1 box2>
        Box
        2 attributeset3>
        AttributeSet
        3 translate4>
        Translate
        4 translate5>
        Translate
        5 translate6>
        Translate
    )
```

Offset	Hexadecimal code	ASCII
0000	3344 4D46 2020 2010 2001 2005 2020 2002	3DMF·····
0010	2020 2020 2020 20DC 636E 7472 2020 205C	·····cntr··\
0020	626F 7820 2020 2030 2020 2020 2020 2020	box···0····
0030	3F80 2020 3F80 2020 2020 2020 2020 2020	?··?·····
0040	2020 2020 2020 2020 2020 2020 2020 2020	·····
0050	3F80 2020 2020 2020 636E 7472 2020 201C	?·····cntr···
0060	6174 7472 2020 2020 6B64 6966 2020 200C	attr···kdif···
0070	3F66 6666 3F66 6666 3E4C CCCD 7472 6E73	?fff?fff>L··trns
0080	2020 200C 4040 2020 2020 2020 2020 2020	···@@·····
0090	7266 726E 2020 2004 2020 2001 7472 6E73	rfrn·····trns
00A0	2020 200C 2020 2020 4040 2020 2020 2020	·····@@·····
00B0	7266 726E 2020 2004 2020 2001 7472 6E73	rfrn·····trns
00C0	2020 200C C040 2020 3F80 2020 2020 2020	····@··?·····
00D0	7266 726E 2020 2004 2020 2001 746F 6320	rfrn·····toc·
00E0	2020 206C 2020 2020 2020 2020 2020 2006	··l·····
00F0	FFFF FFFF 2020 2001 2020 2010 2020 2005	·····
0100	2020 2001 2020 2020 2020 2018 626F 7820	·····box·
0110	2020 2002 2020 2020 2020 2058 6174 7472	·····Xattr
0120	2020 2003 2020 2020 2020 207C 7472 6E73	·····|trns
0130	2020 2004 2020 2020 2020 209C 7472 6E73	·····trns
0140	2020 2005 2020 2020 2020 20BC 7472 6E73	·····trns

Reference Objects

LABELS

ASCII	`Reference`
Binary	`rfrn (= 0x7266726E)`

DATA FORMAT

`Uns32`	`refID`

Field descriptions

`refID`	A positive integer. The reference object with this refID is linked to a file pointer by means of a table of contents entry that contains both the refID and the file pointer.

DATA SIZE

4

DESCRIPTION

A reference object is used to permit an object defined elsewhere to be referenced at one or more locations in a metafile. See “File Pointers,” beginning on page 1272, for details.

PARENT HIERARCHY

Shared.

PARENT OBJECTS

A reference object sometimes but not always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

See the example in "Tables of Contents," beginning on page 1279.

External Reference Objects

LABELS

ASCII	`ExternalReference`
Binary	`rfex (= 0x72666578)`

DATA FORMAT

`Uns32`	`refID`

Field descriptions

`refID` A positive integer. The ExternalReference object with this `refID` is linked to a file pointer in the external file by means of a table of contents entry in the external file that contains both the `refID` and the file pointer. (The external file itself is specified by means of a subobject; see below.)

DATA SIZE

4

DESCRIPTION

An external reference object is used to permit an object defined in another file to be referenced at one or more locations in a metafile.

Version 1.0 of this document described a design for external references that was not implemented in QD3D 1.0. In implementing external references in QD3D 1.5 we have decided on a somewhat different design. Its main advantage is that it allows the table of contents to contain the `ExternalReference` type, which means that tables of contents can be searched for all external references.

Suppose that in metafile F2 you want to reference an object `ObjR` that's contained in some other metafile F1. For this to be possible, F1 must have an

entry to object `ObjR` in its table of contents. In metafile F2, a reference to `ObjR` is made by an external reference object. Its syntax is:

```
Container (
    ExternalReference ( 2 )
    CString ( "ExtRefTransform.TXT"
    )
)
```

In this example, `2` is the reference ID, which is the number used by the TOC to find the object in F1. The string `"ExtRefTransform.TXT"` is the name of file F1 enclosed in quotation marks.

There are two conditions that must be met by the TOC of any metafile that contains `ExternalReference` objects:

- The TOC must have `tocEntryType` = 1, so that the `objectType` of each object appears in the `tocEntry`.
- The TOC must contain an entry for each `ExternalReference` object in the metafile. This is so even if the metafile type is `Normal` and the `ExternalReference` object only appears once in the metafile.

These are needed so that the QD3D call `Q3File_GetExternalReferences` works properly. This call looks at the `objectTypes` in the TOC to determine whether `ExternalReference` objects are present.

PARENT HIERARCHY

Shared.

PARENT OBJECTS

An ExternalReference object sometimes but not always has a parent object.

CHILD OBJECTS

C string that gives the pathname of the file that contains the object to be externally referenced, as described above.

EXAMPLES

```
3DMetafile ( 1 5 Normal tableofcontents4> )
    lambertillumination6:
    LambertIllumination ( )
    translate7:
    Container (
        ExternalReference ( 1 )
        cstring8:
        CString (
            "ExtRefTransformAA.TXT"
        )
    )
    box9:
    Container (
        ExternalReference ( 2 )
        cstring10:
        CString (
            "ExtRefTransformAA.TXT"
        )
    )
    Reference ( 1 )
    tableofcontents4:
    TableOfContents (
        tableofcontents5>               # next TOC
        3                               # reference seed
        -1                              # typeSeed
        1                               # tocEntryType
        16                              # tocEntrySize
        2                               # nEntries
        1 translate7>
        ExternalReference
        2 box9>
        ExternalReference
    )
```

Offset	Hexadecimal code	ASCII
0000	3344 4D46 2020 2010 2001 2005 2020 2020	3DMF············
0010	2020 2020 2020 2094 6C6D 696C 2020 2020	·····lmil····
0020	636E 7472 2020 202C 7266 6578 2020 2004	cntr···,rfex····
0030	2020 2001 7374 7263 2020 2018 4578 7452	····strc····ExtR
0040	6566 5472 616E 7366 6F72 6D41 412E 4249	efTransformAA.BI
0050	4E20 2020 636E 7472 2020 202C 7266 6578	N···cntr···,rfex
0060	2020 2004 2020 2002 7374 7263 2020 2018	·····strc····
0070	4578 7452 6566 5472 616E 7366 6F72 6D41	ExtRefTransformA
0080	412E 4249 4E20 2020 7266 726E 2020 2004	A.BIN···rfrn····
0090	2020 2001 746F 6320 2020 203C 2020 2020	····toc····<····
00A0	2020 2020 2020 2003 FFFF FFFF 2020 2001	·····-------····
00B0	2020 2010 2020 2002 2020 2001 2020 2020	·····-------····
00C0	2020 2020 7266 6578 2020 2002 2020 2020	····rfex····----
00D0	2020 2054 7266 6578	···Trfex

```
3DMetafile ( 1 5 Database
        tableofcontents0> )
    translate2:
    Translate ( -40 30 20 )
    box3:
    Box (
        25 0 0                          # orientation
        0 10 0                          # majorAxis
        0 0 20                          # minorAxis
        -30 -5 -10 # origin
    )
    tableofcontents0:
    TableOfContents (
        tableofcontents1>               # next TOC
        3                               # reference seed
        -1                              # typeSeed
        1                               # tocEntryType
        16                              # tocEntrySize
```

```
        2                                   # nEntries
        1 translate2>
        Translate
        2 box3>
        Box
    )
```

Offset	Hexadecimal code	ASCII
0000	3344 4D46 2020 2010 2001 2005 2020 2002	3DMF····· ·····
0010	2020 2020 2020 2064 7472 6E73 2020 200C	·····dtrns····
0020	C220 2020 41F0 2020 41A0 2020 626F 7820	····A···A···box·
0030	2020 2030 41C8 2020 2020 2020 2020 2020	···0A·····
0040	2020 2020 4120 2020 2020 2020 2020 2020	····A·····
0050	2020 2020 41A0 2020 C1F0 2020 C0A0 2020	····A·····
0060	C120 2020 746F 6320 2020 203C 2020 2020	····toc····<····
0070	2020 2020 2020 2003 FFFF FFFF 2020 2001	·····
0080	2020 2010 2020 2002 2020 2001 2020 2020	·····
0090	2020 2018 7472 6E73 2020 2002 2020 2020	····trns·····
00A0	2020 202C 626F 7820	···,box·

Types

LABELS

ASCII	`Type`
Binary	`type (= 0x74797065)`

DATA FORMAT

`Int32`	`typeID`
`String`	`owner`

Field descriptions

`typeID`	A negative integer. No two type objects in the same file may have the same value in this field.

`owner`	An ISO 9070 owner string. The value of this field may not occur in any other type object. The string must not contain a # character, which is used to demarcate comments in 3DMF.

DATA SIZE

4 + `sizeof(String)`

DESCRIPTION

Type objects are used only in conjunction with custom objects, and have a role only in binary metafiles. The purpose of the type object is to establish a correlation between the dynamically-allocated 4-byte type identifiers used to identify types in binary metafiles, and the text string type identifiers that are the fundamental type identifiers in 3DMF and QD3D. That the text string is the sole identifier of a custom type, and the 4-byte quantity is only dynamic, is a feature that is new in Version 1.5. By *dynamic* we mean that the 4-byte identifier is created anew each time the custom object is registered by an application. A key point is that the 4-byte type identifier need not be constant from metafile to metafile, or from session to session. The 4-byte type identifiers are negative numbers, starting with –1 and decreasing by 1.

The reason there is no need for type objects in text metafiles is that the 4-byte identifiers do not appear at all in text metafiles. So there is no need for a type object that establishes a correlation between a 4-byte identifier and a text string identifier.

This functionality, which is new in Version 1.5, is reflected in the new ways of registering custom objects in QD3D 1.5. Otherwise, it is transparent to the user of the QD3D API.

PARENT HIERARCHY

3DMF.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

Offset	Hexadecimal code	ASCII
0000	3344 4D46 2020 2010 2001 2005 2020 2020	3DMF·····
0010	2020 2020 2020 2070 7479 7065 2020 2014	·····ptype····
0020	FFFF FFFB 4375 7374 6F6D 203A 2045 6C65	····Custom·:·Ele
0030	6D44 2020 636E 7472 2020 2028 7365 7420	mD··cntr···(set·
0040	2020 2020 636E 7472 2020 2018 FFFF FFFB	····cntr·····
0050	2020 2004 2020 2001 696E 7470 2020 2004	·····intp····
0060	2020 2001 7266 726E 2020 2004 2020 2001	····rfrn·····
0070	746F 6320 2020 202C 2020 2020 2020 2020	toc····,·····
0080	2020 2002 FFFF FFFF 2020 2001 2020 2010	·····
0090	2020 2001 2020 2001 2020 2020 2020 2034	·····4
00A0	7365 7420	set·

Containers

LABELS

ASCII	`Container`
Binary	`cntr ( = 0x636E7472 )`

DATA FORMAT

No data.

DATA SIZE

$8k + \Sigma$, where k is the number of elements in the container and Σ is the sum of the sizes of those elements. (For each of the k elements, 4 bytes for the type ID and 4 bytes for the field that holds the element's size, plus the size of the element.)

DESCRIPTION

A container is an ordered collection of objects. Containers are used to form complex objects from simpler objects in ways permitted by the structure of the metafile object hierarchy. In particular, child objects (also called *subobjects*) are attached to parent objects (also called the *root*) through the use of containers. The first object in a container is the parent (or root) object. Every container must contain at least one object. Containers may be nested. An object may be instantiated more than once in a hierarchy of nested containers.

The notation for containers in text metafiles is as follows:

```
Container (
    object0
    .
    .
    .
    objectnobjects-1
)
```

Notations for contained objects are separated by blank spaces rather than by punctuation marks, as is the case in the notation for other objects having nonzero size.

The root object of a container must be a shared object, may not be a container itself, and may not be the target object of a file pointer. The position in the metafile object hierarchy of the root object of a container constrains the number, type, and in some cases the order of occurrence of other elements of that container. Each element of a container other than the root object must be either a legitimate child object of the root object or another container. In the latter case,

the root object of the inner container must be a legitimate child object of the root object of the outer one.

A container may be the target object of a file pointer.

PARENT HIERARCHY

3DMF.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

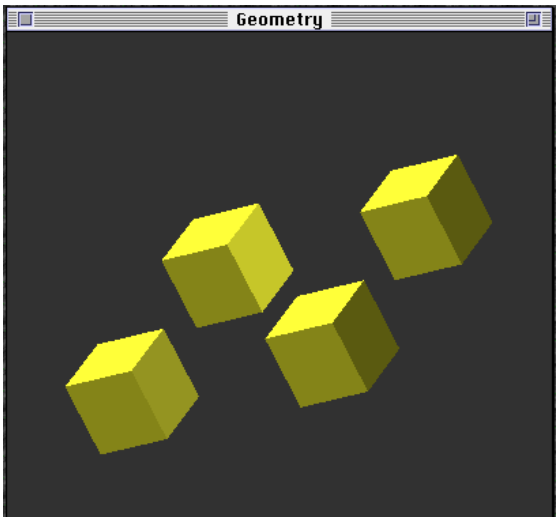
```
3DMetafile ( 1 5 Normal tableofcontents0> )
    box2:
    Container (
        Box (
            0 0 1                   # orientation
            1 0 0                   # majorAxis
            0 0 0                   # minorAxis
            0 1 0                   # origin
        )
        attributeset3:
        Container (
            AttributeSet ( )
            DiffuseColor ( 0.9 0.9 0.2 )
        )
    )
```

Offset	Hexadecimal code	ASCII
0000	3344 4D46 2020 2010 2001 2005 2020 2020	3DMF············
0010	2020 2020 2020 2020 636E 7472 2020 205C	·····cntr···\
0020	626F 7820 2020 2030 2020 2020 2020 2020	box····0·····
0030	3F80 2020 3F80 2020 2020 2020 2020 2020	?···?··········
0040	2020 2020 2020 2020 2020 2020 2020 2020	················
0050	3F80 2020 2020 2020 636E 7472 2020 201C	?·····cntr····
0060	6174 7472 2020 2020 6B64 6966 2020 200C	attr····kdif····
0070	3F66 6666 3F66 6666 3E4C CCCD	?fff?fff>L··

Examples of Metafile Structures

To illustrate the differences among the three types of metafile—stream, normal, and database—we show how a single model (Figure 22-1) is described in a text file of each type. The model consists of four occurrences (at different locations) of a colored box.

Figure 22-1 Four instantiations of a box



The following is a complete specification of each colored box shown in Figure 22-1:

```
3DMetafile ( 1 5 Normal tableofcontents0> )
    box2:
    Container (
        Box (
            0 0 1                   # orientation
            1 0 0                   # majorAxis
            0 0 0                   # minorAxis
            0 1 0                   # origin
        )
        attributeset3:
        Container (
            AttributeSet ( )
            DiffuseColor ( 0.9 0.9 0.2 )
        )
    )
```

Offset	Hexadecimal code	ASCII
0000	3344 4D46 2020 2010 2001 2005 2020 2020	3DMF····· ·····
0010	2020 2020 2020 2020 636E 7472 2020 205C	·····cntr···\
0020	626F 7820 2020 2030 2020 2020 2020 2020	box····0·····
0030	3F80 2020 3F80 2020 2020 2020 2020 2020	?···?·····
0040	2020 2020 2020 2020 2020 2020 2020 2020	·····
0050	3F80 2020 2020 2020 636E 7472 2020 201C	?·····cntr····
0060	6174 7472 2020 2020 6B64 6966 2020 200C	attr····kdif····
0070	3F66 6666 3F66 6666 3E4C CCCD	?fff?fff>L··

The expression `Container (...)` is used subsequently to abbreviate this specification. Transforms are used to place the box in various positions.

In a stream file, the specification of the box must occur four times, as shown in Listing 22-1.

Listing 22-1 A stream metafile

```
3DMetafile ( 1 5 Stream
       tableofcontents0> )
    Container (
        Box (
            0 0 1                       # orientation
            1 0 0                       # majorAxis
            0 0 0                       # minorAxis
            0 1 0                       # origin
        )
        Container (
            AttributeSet ( )
            DiffuseColor ( 0.9 0.9 0.2 )
        )
    )
    Translate ( 3 0 0 )
    Container (
        Box (
            0 0 1                       # orientation
            1 0 0                       # majorAxis
            0 0 0                       # minorAxis
            0 1 0                       # origin
        )
        Container (
            AttributeSet ( )
            DiffuseColor ( 0.9 0.9 0.2 )
        )
    )
    Translate ( 0 3 0 )
    Container (
        Box (
            0 0 1                       # orientation
            1 0 0                       # majorAxis
            0 0 0                       # minorAxis
            0 1 0                       # origin
        )
```

```
        Container (
            AttributeSet ( )
            DiffuseColor ( 0.9 0.9 0.2 )
        )
    )
    Translate ( -3 1 0 )
    Container (
        Box (
            0 0 1                       # orientation
            1 0 0                       # majorAxis
            0 0 0                       # minorAxis
            0 1 0                       # origin
        )
        Container (
            AttributeSet ( )
            DiffuseColor ( 0.9 0.9 0.2 )
        )
    )
```

Offset	Hexadecimal code	ASCII
0000	3344 4D46 2020 2010 2001 2005 2020 2001	3DMF·····
0010	2020 2020 2020 2020 636E 7472 2020 205C	·····cntr···\
0020	626F 7820 2020 2030 2020 2020 2020 2020	box····0·····
0030	3F80 2020 3F80 2020 2020 2020 2020 2020	?···?·····
0040	2020 2020 2020 2020 2020 2020 2020 2020	·····
0050	3F80 2020 2020 2020 636E 7472 2020 201C	?·····cntr····
0060	6174 7472 2020 2020 6B64 6966 2020 200C	attr····kdif····
0070	3F66 6666 3F66 6666 3E4C CCCD 7472 6E73	?fff?fff>L··trns
0080	2020 200C 4040 2020 2020 2020 2020 2020	····@@·····
0090	636E 7472 2020 205C 626F 7820 2020 2030	cntr···\box····0
00A0	2020 2020 2020 2020 3F80 2020 3F80 2020	·····?···?···
00B0	2020 2020 2020 2020 2020 2020 2020 2020	·····
00C0	2020 2020 2020 2020 3F80 2020 2020 2020	·····?·····
00D0	636E 7472 2020 201C 6174 7472 2020 2020	cntr····attr····
00E0	6B64 6966 2020 200C 3F66 6666 3F66 6666	kdif····?fff?fff

```
00F0      3E4C CCCD 7472 6E73 2020 200C 2020 2020      >L··trns·····
0100      4040 2020 2020 2020 636E 7472 2020 205C      @@·····cntr···\
0110      626F 7820 2020 2030 2020 2020 2020 2020      box····0·····
0120      3F80 2020 3F80 2020 2020 2020 2020 2020      ?···?·····
0130      2020 2020 2020 2020 2020 2020 2020 2020      ·····
0140      3F80 2020 2020 2020 636E 7472 2020 201C      ?·····cntr····
0150      6174 7472 2020 2020 6B64 6966 2020 200C      attr····kdif····
0160      3F66 6666 3F66 6666 3E4C CCCD 7472 6E73      ?fff?fff>L··trns
0170      2020 200C C040 2020 3F80 2020 2020 2020      ·····@··?·····
0180      636E 7472 2020 205C 626F 7820 2020 2030      cntr···\box····0
0190      2020 2020 2020 2020 3F80 2020 3F80 2020      ·····?···?···
01A0      2020 2020 2020 2020 2020 2020 2020 2020      ·····
01B0      2020 2020 2020 2020 3F80 2020 2020 2020      ·····?·····
01C0      636E 7472 2020 201C 6174 7472 2020 2020      cntr····attr····
01D0      6B64 6966 2020 200C 3F66 6666 3F66 6666      kdif····?fff?fff
01E0      3E4C CCCD                                    >L··
```

Such repetition can make stream files lengthy. However, a stream file can be read by a parser having only sequential access to that file.

In a normal file, the box is completely specified once and is instantiated by reference three times. The file pointers and reference objects used to effect instantiations by reference are listed together in the table of contents. Other objects able to be referenced (such as the transforms) that are instantiated once only are not listed in the table of contents. The normal metafile permits the most compact representation of the model.

Listing 22-2 A normal metafile

```
3DMetafile ( 1 5 Normal tableofcontents0> )
    box2:
    Container (
        Box (
            0 0 1                   # orientation
            1 0 0                   # majorAxis
            0 0 0                   # minorAxis
            0 1 0                   # origin
        )
```

```
        attributeset3:
        Container (
            AttributeSet ( )
            DiffuseColor ( 0.9 0.9 0.2 )
        )
    )
    translate4:
    Translate ( 3 0 0 )
    Reference ( 1 )
    translate5:
    Translate ( 0 3 0 )
    Reference ( 1 )
    translate6:
    Translate ( -3 1 0 )
    Reference ( 1 )
    tableofcontents0:
    TableOfContents (
        tableofcontents1>               # next TOC
        2                               # reference seed
        -1                              # typeSeed
        1                               # tocEntryType
        16                              # tocEntrySize
        1                               # nEntries
        1 box2>
        Box
    )
```

Offset	Hexadecimal code	ASCII
0000	3344 4D46 2020 2010 2001 2005 2020 2020	3DMF············
0010	2020 2020 2020 20DC 636E 7472 2020 205C	·····cntr···\
0020	626F 7820 2020 2030 2020 2020 2020 2020	box····0·····
0030	3F80 2020 3F80 2020 2020 2020 2020 2020	?···?··········
0040	2020 2020 2020 2020 2020 2020 2020 2020	················
0050	3F80 2020 2020 2020 636E 7472 2020 201C	?·····cntr····
0060	6174 7472 2020 2020 6B64 6966 2020 200C	attr····kdif····
0070	3F66 6666 3F66 6666 3E4C CCCD 7472 6E73	?fff?fff>L··trns
0080	2020 200C 4040 2020 2020 2020 2020 2020	····@@··········

```
0090      7266 726E 2020 2004 2020 2001 7472 6E73      rfrn·····trns
00A0      2020 200C 2020 2020 4040 2020 2020 2020      ·····@@·····
00B0      7266 726E 2020 2004 2020 2001 7472 6E73      rfrn·····trns
00C0      2020 200C C040 2020 3F80 2020 2020 2020      ····@··?····
00D0      7266 726E 2020 2004 2020 2001 746F 6320      rfrn·····toc·
00E0      2020 202C 2020 2020 2020 2020 2020 2002      ···,·····
00F0      FFFF FFFF 2020 2001 2020 2010 2020 2001      ·····
0100      2020 2001 2020 2020 2020 2018 626F 7820      ·····box·
```

The pointer box2> is a file pointer correlated with the label box2 that precedes the specification of the box. `Reference ( 1 )` is a reference object correlated with box2> (and thus with the specification of the box) in the table of contents. See "File Pointers," beginning on page 1272 for an explanation of how instantiation by reference is accomplished through the use of these objects.

In a database file, the box is also instantiated by reference, and the file pointers and reference objects used to instantiate it are listed in the table of contents. With the exception of reference objects themselves, all other objects able to be referenced (the attribute set which contains the box's color attributes, and the transforms) are referenced, and all of these references are listed in the table of contents.

The contents of a database file can be discovered quickly by inspecting its tables of contents.

Listing 22-3 A database metafile

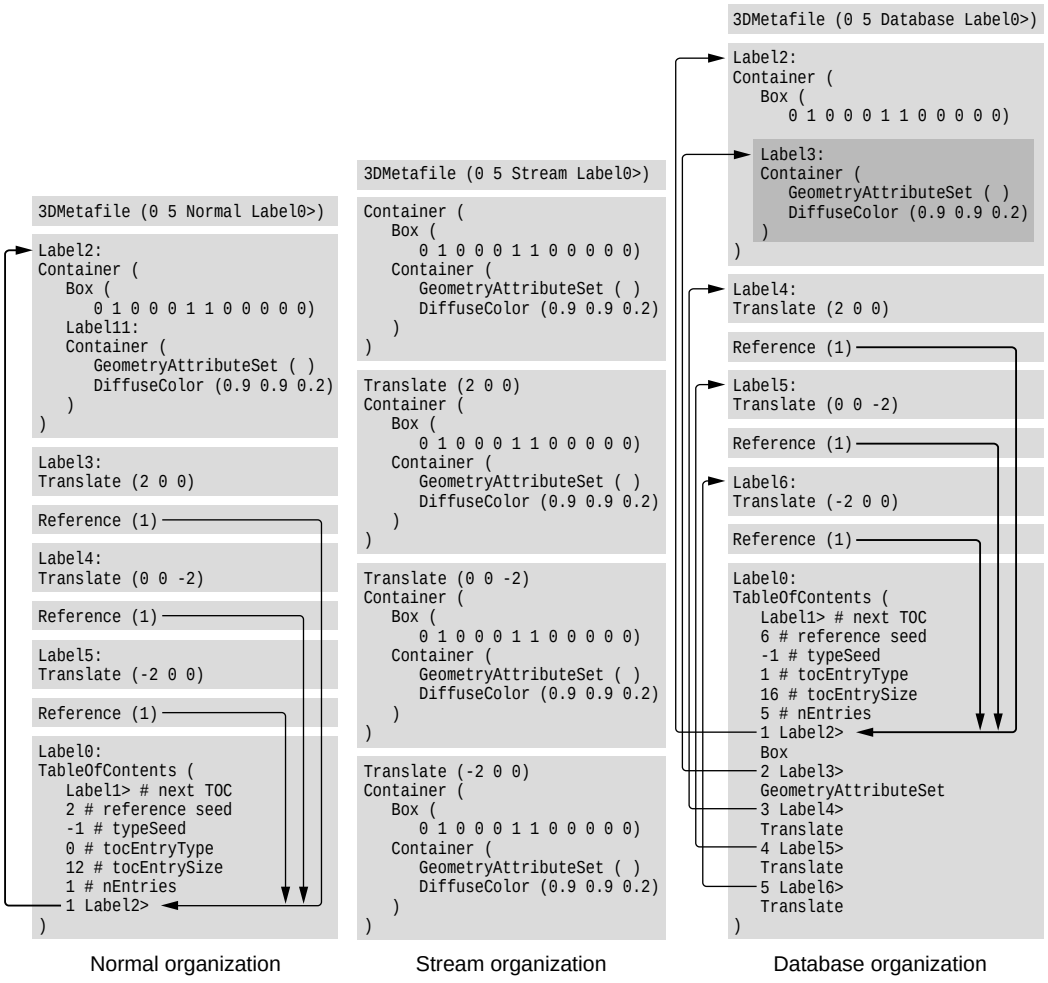
```
3DMetafile ( 1 5 Database
       tableofcontents0> )
   box2:
   Container (
      Box (
         0 0 1                  # orientation
         1 0 0                  # majorAxis
         0 0 0                  # minorAxis
         0 1 0                  # origin
      )
```

```
        attributeset3:
        Container (
            AttributeSet ( )
            DiffuseColor ( 0.9 0.9 0.2 )
        )
    )
    translate4:
    Translate ( 3 0 0 )
    Reference ( 1 )
    translate5:
    Translate ( 0 3 0 )
    Reference ( 1 )
    translate6:
    Translate ( -3 1 0 )
    Reference ( 1 )
    tableofcontents0:
    TableOfContents (
        tableofcontents1>               # next TOC
        6                               # reference seed
        -1                              # typeSeed
        1                               # tocEntryType
        16                              # tocEntrySize
        5                               # nEntries
        1 box2>
        Box
        2 attributeset3>
        AttributeSet
        3 translate4>
        Translate
        4 translate5>
        Translate
        5 translate6>
        Translate
    )
```

Offset	Hexadecimal code	ASCII
0000	3344 4D46 2020 2010 2001 2005 2020 2002	3DMF····· ·····
0010	2020 2020 2020 20DC 636E 7472 2020 205C	····· ···cntr···\
0020	626F 7820 2020 2030 2020 2020 2020 2020	box····0···· ····
0030	3F80 2020 3F80 2020 2020 2020 2020 2020	?···?····· ·····
0040	2020 2020 2020 2020 2020 2020 2020 2020	····· ·····
0050	3F80 2020 2020 2020 636E 7472 2020 201C	?·····cntr···· ··
0060	6174 7472 2020 2020 6B64 6966 2020 200C	attr····kdif···· ··
0070	3F66 6666 3F66 6666 3E4C CCCD 7472 6E73	?fff?fff>L··trns
0080	2020 200C 4040 2020 2020 2020 2020 2020	····@@····· ·····
0090	7266 726E 2020 2004 2020 2001 7472 6E73	rfrn····· ···trns
00A0	2020 200C 2020 2020 4040 2020 2020 2020	····· ···@@·····
00B0	7266 726E 2020 2004 2020 2001 7472 6E73	rfrn····· ···trns
00C0	2020 200C C040 2020 3F80 2020 2020 2020	·····@··?···· ····
00D0	7266 726E 2020 2004 2020 2001 746F 6320	rfrn····· ···toc·
00E0	2020 206C 2020 2020 2020 2020 2020 2006	···l····· ·····
00F0	FFFF FFFF 2020 2001 2020 2010 2020 2005	····· ·····
0100	2020 2001 2020 2020 2020 2018 626F 7820	····· ····box·
0110	2020 2002 2020 2020 2020 2058 6174 7472	····· ····Xattr
0120	2020 2003 2020 2020 2020 207C 7472 6E73	····· ····|trns
0130	2020 2004 2020 2020 2020 209C 7472 6E73	····· ····trns
0140	2020 2005 2020 2020 2020 20BC 7472 6E73	····· ····trns

Figure 22-2 shows, side by side, the three principal forms of a metafile.

Figure 22-2 Types of metafiles



String Objects

C Strings

LABELS

ASCII	`CString`
Binary	`strc ( = 0x73747263)`

DATA FORMAT

`String`	`cString`

Field descriptions

`cString`	A string constant (that is, a sequence of ASCII characters enclosed in double-quotation marks). See "Strings," beginning on page 1263, for a list of the escape sequences that may occur in a `cString` object.

DATA SIZE

```
sizeof(String)
```

DESCRIPTION

A C string may be used to include text in a metafile.

PARENT HIERARCHY

Shared, string.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
cString (
“Copyright Apple Computer, Inc., 1995”
)
```

Unicode Objects

LABELS

ASCII	`Unicode`
Binary	`uncd (= 0x756E6364)`

DATA FORMAT

`uns32`	`length`
`RawData`	`unicode[length * 2]`

Field descriptions

`length`	The length of the encoded text.
`unicode[]`	An array of raw data that encodes text.

DATA SIZE

```
4 + length * 2
```

DESCRIPTION

A unicode object may be used to include text in a binary metafile.

PARENT HIERARCHY

Shared, String.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
Unicode (
    6
    0x457363686572
)
```

Geometric Objects

This section describes the geometric objects currently supported by the metafile specification.

Points

LABELS

ASCII	`Point`
Binary	`pnt ( = 0x706E7420 )`

DATA FORMAT

`Point3D`	`point`

Field descriptions

`point`	A three-dimensional point.

DATA SIZE

12

DESCRIPTION

A point object is used to specify a point in world space. A point object may appear only in a group or as part of the definition of a custom data type. Unlike the corresponding point data type, a geometric point object may be assigned attributes such as color. Thus, an application may use point objects to specify visible dots.

DEFAULT SURFACE PARAMETERIZATION

None.

PARENT HIERARCHY

Shared, shape, geometry.

PARENT OBJECTS

None.

CHILD OBJECTS

Attribute set (optional).

EXAMPLE

```
Point ( 0 0 0 )
```

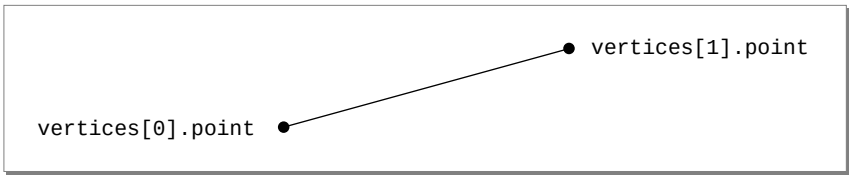
DEFAULT SIZE

None.

Lines

Figure 22-3 shows a line.

Figure 22-3 A line



LABELS

ASCII	`Line`
Binary	`line ( = 0x6C696E65 )`

DATA FORMAT

`Point3D`	`start`
`Point3D`	`end`

Field descriptions

`start`	One endpoint of the line.
`end`	The other endpoint of the line.

DATA SIZE

24

DESCRIPTION

A line is a straight segment in three-dimensional space defined by its two endpoints. Attributes may be assigned to the vertices of a line and to the entire line.

DEFAULT SURFACE PARAMETERIZATION

The default surface parameterization for a line is (0, 0) at `start` and (1, 0) at `end`.

PARENT HIERARCHY

Shared, shape, geometry.

PARENT OBJECTS

None.

CHILD OBJECTS

Attribute set (optional), vertex attribute set list (optional). An attribute set may be used to assign attributes to the entire line. The vertex attribute set list may include attribute sets for one or both vertices of the line. For the purpose of attribute assignment, the `start` and `end` vertices of a line are indexed by the integers 0 and 1 respectively. See "Vertex Attribute Set Lists," beginning on page 1420 for a description of these lists.

EXAMPLE

```
Container (
    Line (
        0 0 0
        1 0 0
    )
    Container (
        VertexAttributeSetList ( 2 Exclude 0 )
        Container (
            AttributeSet ( )
            DiffuseColor ( 1 0 0 )
        )
        Container (
            AttributeSet ( )
            DiffuseColor ( 0 0 1 )
        )
    )
)
```

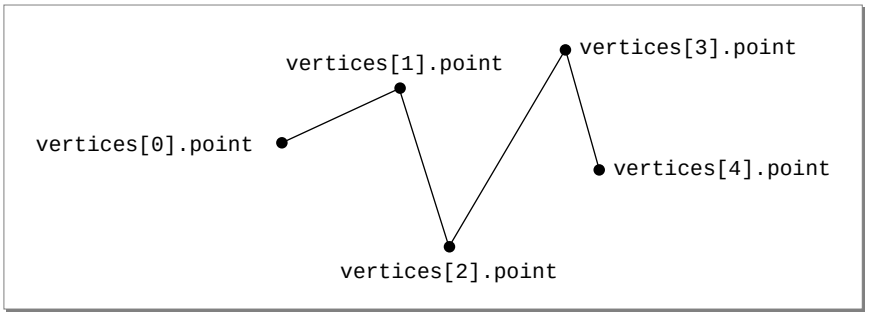
DEFAULT SIZE

None.

Polylines

Figure 22-4 shows a polyline.

Figure 22-4 A polyline



LABELS

ASCII	`Polyline`
Binary	`plin ( = 0x706C696E )`

DATA FORMAT

`Uns32`	`numVertices`
`Point3D`	`vertices[numVertices]`

Field descriptions

`numVertices`	The number of vertices of the polyline.
`vertices[]`	An array of vertices that define the polyline.

DATA SIZE

```
4 + (numVertices * 12)
```

DESCRIPTION

A polyline is a collection of n lines defined by the n+1 points that define the vertices of its segments. For $1 \leq i \leq n-1$, the second vertex of the ith line is the first vertex of the i+1st line; the n+1st vertex of a polyline is not connected to the first. Attributes may be assigned separately to each vertex and to each segment of a polyline as well as to the entire polyline.

DEFAULT SURFACE PARAMETERIZATION

None.

PARENT HIERARCHY

Shared, shape, geometry.

PARENT OBJECTS

None.

CHILD OBJECTS

Attribute set, geometry attribute set list, vertex attribute set list. Use a vertex attribute set list to assign attribute sets to as many vertices as desired; use a geometry attribute set list to assign attribute sets to as many segments as desired. Use an attribute set to assign attributes to the entire polyline.

EXAMPLE

```
PolyLine(
   5                #numVertices
   0   0   0        #first vertex
   1   1   0        #second vertex
   .5  .5  0
   0   1   0
   1   1   0
)
```

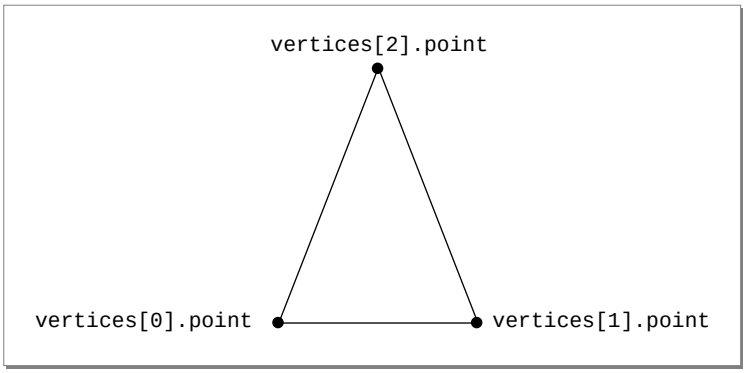
DEFAULT SIZE

None.

Triangles

Figure 22-5 shows a triangle.

Figure 22-5 A triangle



LABELS

ASCII	`Triangle`
Binary	`trng` (= `0x74726E67`)

DATA FORMAT

`Point3D`	`vertices[3]`

Field descriptions

`vertices[]`	An array of triangle vertices.

DATA SIZE

36

DESCRIPTION

A triangle is a closed plane figure defined by three vertices. Attributes may be assigned to each vertex of a triangle and also to its entire face.

DEFAULT SURFACE PARAMETERIZATION

None.

PARENT HIERARCHY

Shared, shape, geometry.

PARENT OBJECTS

None.

CHILD OBJECTS

Vertex attribute set list (optional), attribute set (optional). A vertex attribute set list may be used to attach attributes to one or more vertices of the triangle. An attribute set may be used to attach attributes to the entire face of the triangle.

EXAMPLE

```
Container (
    Triangle (
        -1 -0.5 -0.25
        0 0 0
        -0.5 1.5 0.45
    )
    Container (
        VertexAttributeSetList ( 3 Exclude 0 )
        Container (
            AttributeSet ( )
            DiffuseColor ( 1 0 0 )
        )
        Container (
            AttributeSet ( )
```

```
            DiffuseColor ( 0 1 0 )
        )
        Container (
            AttributeSet ( )
            DiffuseColor ( 0 0 1 )
        )
    )
    Container (
        AttributeSet ( )
        DiffuseColor ( 0.8 0.5 0.2 )
    )
)
```

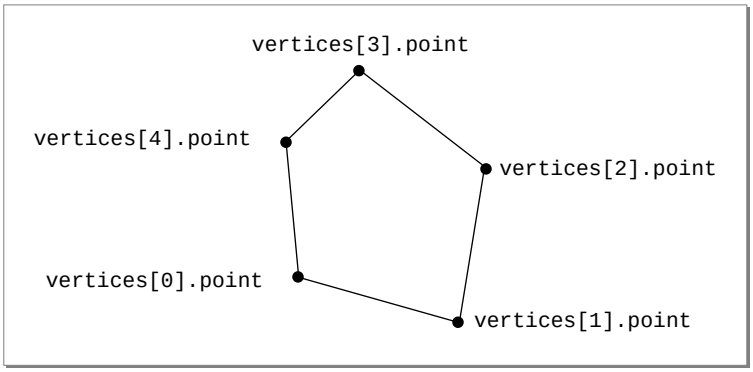
DEFAULT SIZE

None.

Simple Polygons

Figure 22-6 shows a simple polygon.

Figure 22-6 A simple polygon



LABELS

ASCII	`Polygon`
Binary	`plyg ( = 0x706C7967 )`

DATA FORMAT

`uns32`	`nVertices`
`Point3D`	`vertices[nVertices]`

Field descriptions

`nVertices`	The number of vertices of the polygon.
`vertices[]`	An array of vertices that define the polygon.

DATA SIZE

```
4 + (numVertices * 12)
```

DESCRIPTION

A simple polygon is a convex plane figure defined by a list of vertices. In other words, a simple polygon is a polygon defined by a single contour. (Vertices are assumed to be coplanar to within floating-point tolerances.) The lines connecting the vertices of a simple polygon do not cross. Attributes may be assigned to each vertex of a simple polygon and also to its entire face.

DEFAULT SURFACE PARAMETERIZATION

None.

PARENT HIERARCHY

Shared, shape, geometry.

PARENT OBJECTS

None.

CHILD OBJECTS

Vertex attribute set list (optional), attribute set (optional). A vertex attribute set list may be used to attach attribute sets to one or more vertices of the simple polygon. An attribute set may be used to attach attributes to the entire face of the simple polygon. For the purpose of attribute assignment, the vertices of a polygon are indexed by position in the array `vertices[]`; that is, the index of `vertices[i]` is *i*. See "Vertex Attribute Set Lists," beginning on page 1420, for an explanation of the structure and syntax of these objects.

EXAMPLE

```
Polygon(
   5                #nVertices
   0  0  0
   1  0  0
   2  1  0
   1  2  0
   0  1  0
)
```

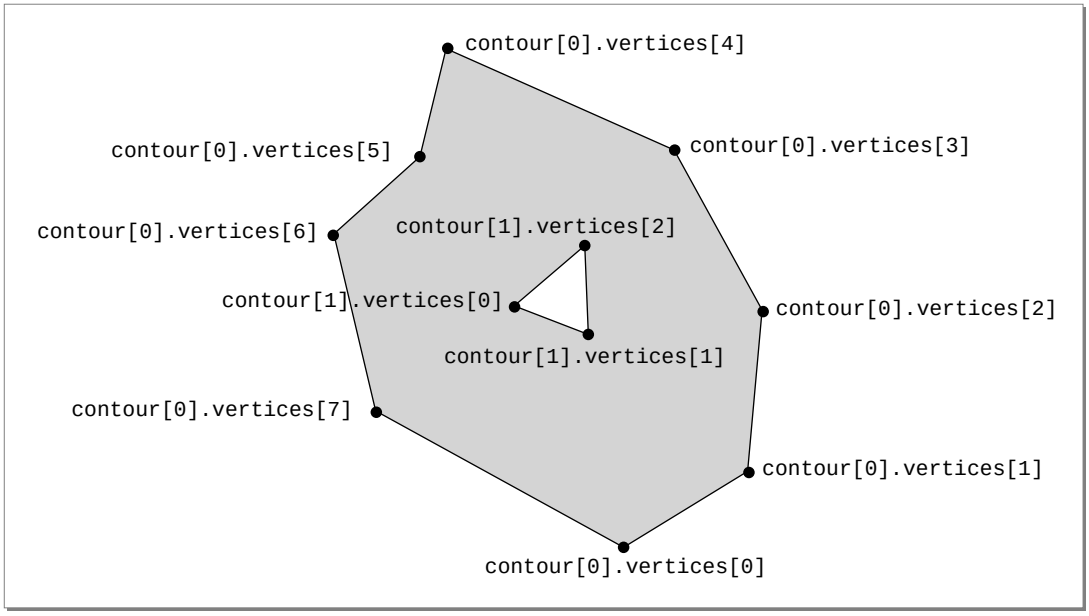
DEFAULT SIZE

None.

General Polygons

Figure 22-7 shows a general polygon.

Figure 22-7 A general polygon



LABELS

ASCII	`GeneralPolygon`
Binary	`gpgn` (= `0x6770676E`)

POLYGON DATA DATA TYPE

`uns32`	`nVertices`
`Point3D`	`vertices[nVertices]`

Field descriptions

`nVertices`	The number of vertices of this contour of the general polygon.
`vertices[]`	An array of vertices that define this contour of the general polygon.

DATA FORMAT

`Uns32`	`nContours`
`PolygonData`	`polygons[nContours]`

Field descriptions

`nContours`	The number of contours of the general polygon.
`polygons[]`	An array of contours that define the general polygon.

DATA SIZE

```
sizeof(PolygonData) = 4 + nVertices * 12
sizeof(GeneralPolygon) = 4 + sizeof(polygons[0...nContours-1])
```

DESCRIPTION

A general polygon is a closed plane figure defined by one or more lists of vertices. In other words, a general polygon is a polygon defined by one or more contours. Each contour may be concave or convex, and contours may be nested. All contours, however, must be coplanar. A general polygon can have holes in it. If it does, the even-odd rule is used to determine which regions are included in the polygon. Attributes may be assigned to each vertex of each contour of a general polygon and to the entire general polygon.

DEFAULT SURFACE PARAMETERIZATION

None.

PARENT HIERARCHY

Shared, shape, geometry.

PARENT OBJECTS

None.

CHILD OBJECTS

Attribute set, general polygon hint, vertex attribute set list (all optional). Use an attribute set to attach attributes to an entire general polygon. Use a general polygon hint to specify whether a general polygon is concave, convex, or complex; see "General Polygon Hints," beginning on page 1322 for complete details on this object. Use a vertex attribute set list to assign attributes to the vertices of the contours of a general polygon. For purposes of attribute assignment, the vertices of a general polygon are indexed in the order of their occurrence in the specification of that polygon; the index does not distinguish between contours. For purposes of attribute assignment, the nth contour of a general polygon is the contour defined by `(polygons[n-1])[1]`, and the index of the nth contour is n–1. The nth vertex of a general polygon is the pth vertex of the mth contour, where

$m = \max\{k \le$ `nContours` $: \Sigma_{0 \le i < k-1}$ `(polygons[`i`])[0]` $< n\}$,

and $n = \Sigma_{0 \le i < m}$ `(polygons[`i`])[0]` $+ p$; the index of the nth vertex of a general polygon is n-1. The pth vertex of the mth contour of a general polygon is the $(\Sigma_{0 \le i < \mu-1}$ `(polygons[`i`])[0]` $+ p)$th vertex of the general polygon; its index is $\Sigma_{0 \le i < \mu-1}$ `(polygons[`i`])[0]` $+ (p - 1)$. See "Face Attribute Set Lists," beginning on page 1416, and "Vertex Attribute Set Lists," beginning on page 1420, for explanations of the structure and syntax of these objects.

EXAMPLE

```
Container (
    GeneralPolygon (
        2                       # nContours
    #contour 0
        3                       # nVertices, contour 0
        -1 0 0                  # vertex 0
        1 0 0                   # vertex 1
        0 1.7 0                 # vertex 2
    #contour 1
        3                       # nVertices, contour 1
        -1 0.4 0                # vertex 3
        1 0.4 0                 # vertex 4
        0 2.1 0                 # vertex 5
    )
```

```
    Container (
        VertexAttributeSetList ( 6 Exclude 2 0 4 )     #see note
        Container (
            AttributeSet ( )                # vertex 1
            DiffuseColor ( 0 0 1 )
        )
        Container (
            AttributeSet ( )                # vertex 2 (contour 0)
            DiffuseColor ( 0 1 1 )
        )
        Container (
            AttributeSet ( )                # vertex 3 (contour 1)
            DiffuseColor ( 1 0 1 )
        )
        Container (
            AttributeSet ( )                # vertex 5 (contour 1)
            DiffuseColor ( 1 1 0 )
        )
    )
    Container (
        AttributeSet ( )
        DiffuseColor ( 1 1 1 )
    )
)
```

Note
In the above example, the general polygon has two contours. Each contour is a triangle. The triangles overlap. The intersection of the triangles is included in an even number of contours; thus, it constitutes a hole in the general polygon. The relative complements of the triangles are included in an odd number of contours; thus, they are included in the general polygon. ◆

DEFAULT SIZE

None.

General Polygon Hints

LABELS

ASCII	`GeneralPolygonHint`
Binary	`gplh` (= `0x67706C68`)

GENERAL POLYGON HINTS

`Complex`	`0x00000000`
`Concave`	`0x00000001`
`Convex`	`0x00000002`

Constant descriptions

`Complex`	The parent general polygon may include concave, convex, and self-intersecting polygons.
`Concave`	All contours of the parent general polygon are concave and none is self-intersecting.
`Convex`	All contours of the parent general polygon are convex and none is self-intersecting.

DATA FORMAT

`GeneralPolygonHintEnum`	`shapeHint`

Field descriptions

`shapeHint`	The value in this field must be one of the constants defined above.

DATA SIZE

4

DESCRIPTION

A general polygon hint object is used to provide a reading application with an indication of the shape of a general polygon.

DEFAULT SURFACE PARAMETERIZATION

None.

PARENT HIERARCHY

Data.

PARENT OBJECTS

General polygon. A general polygon hint object always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
    GeneralPolygon ( ... )
    GeneralPolygonHint ( Complex )
)
```

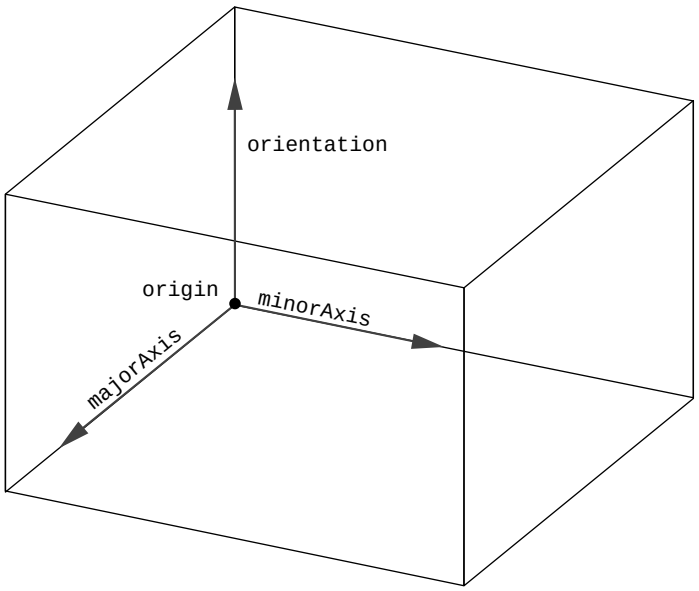
DEFAULT VALUE

`Complex`.

Boxes

Figure 22-8 shows a box.

Figure 22-8 A box



LABELS

ASCII	`Box`
Binary	`box (= 0x626F7820)`

DATA FORMAT

`Vector3D`	`orientation`
`Vector3D`	`majorAxis`
`Vector3D`	`minorAxis`
`Point3D`	`origin`

Field descriptions

`orientation`	The orientation of the box.
`majorAxis`	The major axis of the box.
`minorAxis`	The minor axis of the box.
`origin`	The origin of the box.

DATA SIZE

0 or 48

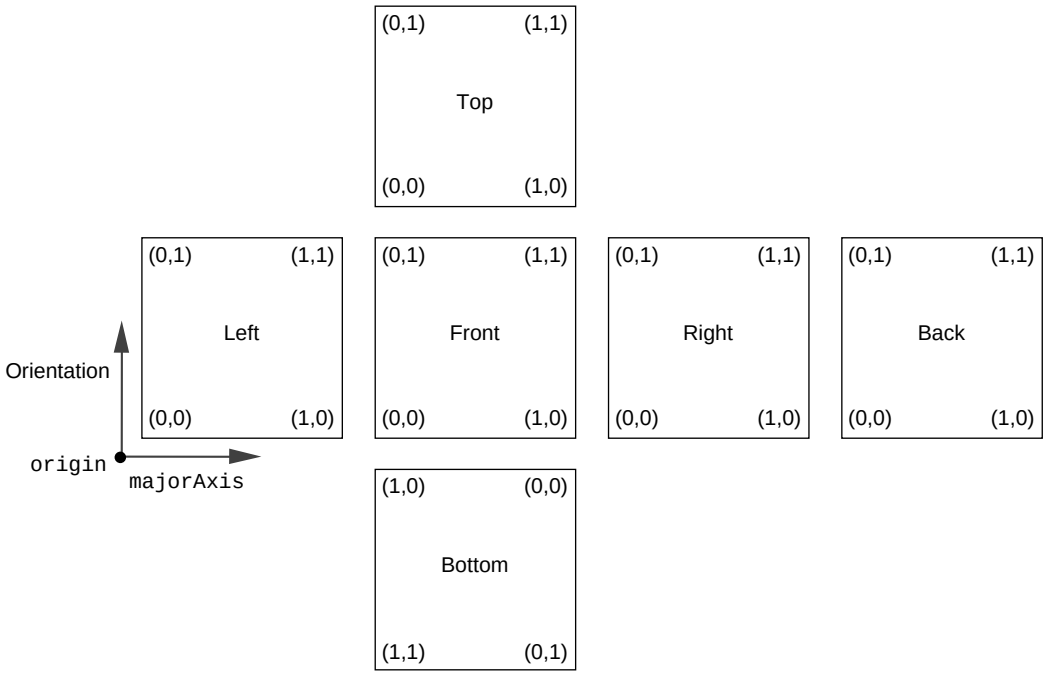
DESCRIPTION

A box is a three-dimensional object defined by an origin (that is, a corner of the box) and three vectors that define the edges of the box that meet in that corner. A box may be used to model a cube, rectangular prism, or other parallelipiped. Attributes may be applied to each of the six faces of a box and to the entire geometry of the box.

DEFAULT SURFACE PARAMETERIZATION

The default surface parameterization for a box is as shown in Figure 22-9.

Figure 22-9 The default surface parameterization of a box



PARENT HIERARCHY

Shared, shape, geometry.

PARENT OBJECTS

None.

CHILD OBJECTS

Face attribute set list (optional), attribute set (optional). For the purpose of attribute assignment, the faces of a box are indexed as follows:

0	The face perpendicular to the orientation vector having the endpoint of the orientation vector as one of its vertices. In Figure 22-9, this is the top face of the box.
1	The face perpendicular to the orientation vector having the origin as one of its vertices. In Figure 22-9, this is the bottom face of the box.
2	The face perpendicular to the major axis having the endpoint of the major axis as one of its vertices. In Figure 22-9, this is the front face of the box.
3	The face perpendicular to the major axis having the origin as one of its vertices. In Figure 22-9, this is the back face of the box.
4	The face perpendicular to the minor axis having the endpoint of the minor axis as one of its vertices. In Figure 22-9, this is the right face of the box.
5	The face perpendicular to the minor axis having the origin as one of its vertices. In Figure 22-9, this is the front face of the box.

EXAMPLE

```
Container (
    Box ( ... )
    Container (
        FaceAttributeSetList (6 Exclude 2 1 4)
```

```
        Container (
            AttributeSet ( )                #left face
            DiffuseColor ( 1 0 0 )
        )
        Container (
            AttributeSet ( )                #bottom face
            DiffuseColor ( 0 1 1 )
        )
        Container (
            AttributeSet ( )                #top face
            DiffuseColor ( 0 1 0 )
        )
        Container (
            AttributeSet ( )                #front face
            DiffuseColor ( 1 0 1 )
        )
    )
    Container (
        AttributeSet ( )
        DiffuseColor(0 0 0)
    )
)
```

DEFAULT SIZE

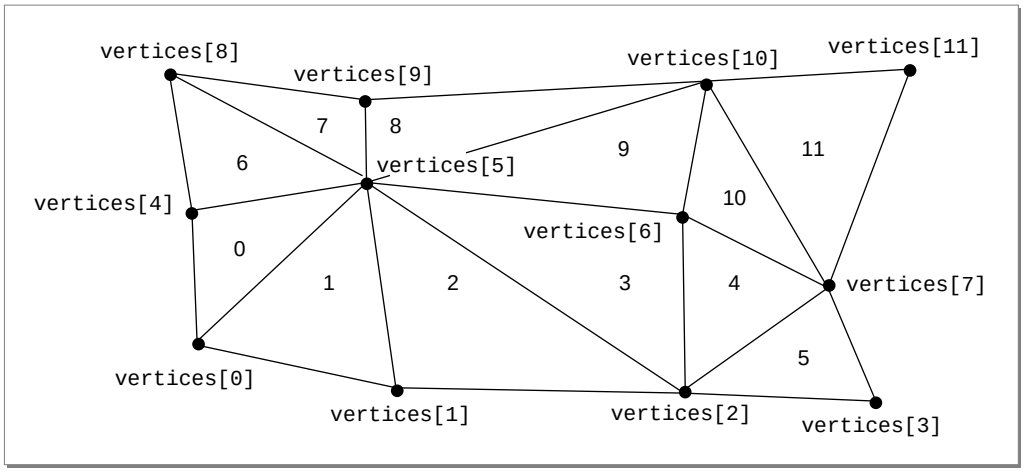
For objects of size 0, the default is

```
1 0 0
0 1 0
0 0 1
0 0 0
```

Trigrids

Figure 22-10 shows a trigrid.

Figure 22-10 A trigrid



LABELS

ASCII	`TriGrid`
Binary	`trig` (= `0x74726967`)

DATA FORMAT

`Uns32`	`numUVertices`
`Uns32`	`numVVertices`
`Point3D`	`vertices[numUVertices * numVVertices]`

Field descriptions

`numUVertices`	The number of vertices in the *u* parametric direction.
`numVVertices`	The number of vertices in the *v* parametric direction.
`vertices[]`	An array of vertices. The size of this array must equal the number of vertices of the trigrid. Vertices are to be listed in a rectangular order, first in the direction of increasing *v*, then in the direction of increasing *u*. That is, the vertex having parametric coordinates (*u*, *v*) precedes the vertex having parametric coordinates (*u′*, *v′*) if and only if either $u < u'$, or $u = u'$ and $v < v'$.

DATA SIZE

```
8 + (numUVertices * numVVertices * 12)
```

DESCRIPTION

A trigrid is a grid composed of triangular facets. The triangulation should be serpentine (that is, quadrilaterals are divided into triangles in an alternating fashion) to reduce shading artifacts when using Gouraud or Phong shading. Attributes may be assigned to each vertex and to each facet of a trigrid, and also to the entire trigrid.

PARENT HIERARCHY

Shared, shape, geometry.

PARENT OBJECTS

None.

CHILD OBJECTS

Vertex attribute set list (optional), face attribute set list (optional), attribute set (optional). A face attribute set list may be used to assign attributes to the facets of a trigrid. The number of facets of a trigrid is the same as the number of its vertices. The vertices and facets of a trigrid are indexed in the manner shown by Figure 22-10. The vertex index prefers u to v and prefers 0 to 1; thus, it follows the canonical lexicographical ordering of the points in uv parametric space. The facet index is less easily defined but is readily apprehended. Consider first the serpentine path through the trigrid along the diagonals belonging to facets of the trigid. Now consider the alternative serpentine path composed of segments connecting all and only those vertices not on the first path. The second path passes through each facet and intersects all of the diagonals on the first path. The facets of the trigrid are numbered in the order that they would be encountered by a traveler along the second serpentine path.

EXAMPLE

```
Container (
    TriGrid (
        3          #numUVertices
        4          #numVVertices
        2  0  0      2  1  0      2  2  0      2  3  0
        1  0  0      1  1  0      1  2  0      1  3  0
        0  0  0      0  1  0      0  2  0      0  3  0
    )
    Container (
        FaceAttributeSetList (12 include 61 3 5 7 9 11)
        Container (
            AttributeSet( )
            DiffuseColor (1 1 1)
        )
        .
        .
        .
        Container (
            AttributeSet( )
            DiffuseColor (1 1 1)
        )
    )
    Container (
        AttributeSet ( )
        DiffuseColor ( 0 0 0 )
    )
)
```

DEFAULT SIZE

None.

Polyhedra

LABELS

ASCII	`Polyhedron`
Binary	`plhd`

DATA FORMAT

The clearest way to understand the metafile format for polyhedra is to begin with the polyhedron data structures in the QD3D file QD3DGeometry.h:

```
typedef enum TQ3PolyhedronEdgeMasks {
    kQ3PolyhedronEdgeNone    = 0,
    kQ3PolyhedronEdge01      = 1 << 0,
    kQ3PolyhedronEdge12      = 1 << 1,
    kQ3PolyhedronEdge20      = 1 << 2,
    kQ3PolyhedronEdgeAll     = kQ3PolyhedronEdge01 |
                               kQ3PolyhedronEdge12 |
                               kQ3PolyhedronEdge20
} TQ3PolyhedronEdgeMasks;

typedef unsigned long TQ3PolyhedronEdge;

typedef struct TQ3PolyhedronEdgeData {
    unsigned long     vertexIndices[2];
    unsigned long     triangleIndices[2];
    TQ3AttributeSet   edgeAttributeSet;
} TQ3PolyhedronEdgeData;

typedef struct TQ3PolyhedronTriangleData {
    unsigned long         vertexIndices[3];
    TQ3PolyhedronEdge     edgeFlag;
    TQ3AttributeSet       triangleAttributeSet;
} TQ3PolyhedronTriangleData;
```

```
typedef struct TQ3PolyhedronData {
    unsigned long               numVertices;
    TQ3Vertex3D                 *vertices;
    unsigned long               numEdges;
    TQ3PolyhedronEdgeData       *edges;
    unsigned long               numTriangles;
    TQ3PolyhedronTriangleData   *triangles;
    TQ3AttributeSet             polyhedronAttributeSet;
} TQ3PolyhedronData;
```

The polyhedron metafile object makes use of the following auxiliary data structures. These differ from the above data structures in that all `attributeSet` fields have been removed. Instead, the `attributeSet` fields are collected in `AttributeSetList` subobjects, which are discussed in "Attribute Set Lists," beginning on page 1414.

```
typedef enum PolyhedronEdge {
    PolyhedronEdgeNone    = 0,
    PolyhedronEdge01      = 1 << 0,
    PolyhedronEdge12      = 1 << 1,
    PolyhedronEdge20      = 1 << 2,
    PolyhedronEdgeAll     = PolyhedronEdge01 |
                            PolyhedronEdge12|
                            PolyhedronEdge20
} PolyhedronEdge;

typedef struct PolyhedronEdgeData {
    unsigned long     vertexIndices[2];
    unsigned long     triangleIndices[2];
} PolyhedronEdgeData;

typedef struct PolyhedronTriangleData {
    unsigned long         vertexIndices[3];
    PolyhedronEdge        edgeFlag;
} PolyhedronTriangleData;
```

Given these, the metafile format of the polyhedron object itself is:

```
Uns32                   numVertices
Uns32                   numEdges
Uns32                   numTriangles
Point3D                 vertices[numVertices]
PolyhedronEdgeData      edges[numEdges]
PolyhedronTriangleData  triangles[numTriangles]

numVertices                 The number of vertices.
```

`numEdges` The number of edges.

`numTriangles` The number of edges.

`vertices[numVertices]`
An array of `Point3D`. See the QD3D polyhedron data structure above to see how it fits into the geometry as a whole.

`edges[numEdges]` An array of `PolyhedronEdgeData`. The `PolyhedronEdgeData` data structure consists two fields. The first field is an array of two indices into the array of vertices; the two specified vertices are the two points that bound the edge. The second field is an array of two indices into the array of triangles; the two specified triangles are those whose common side consitutes the edge. A nonexistent triangle (e.g. one with an edge at the boundary of a planar surface) is indicated by the value 0xFFFFFFFF.

`triangles[numTriangles]`
An array of `PolyhedronTriangleData`. The `PolyhedronTriangleData` data structure consists of two fields. The first field is an array of 3 indices into the array of vertices; these 3 vertices are the vertices of the triangle. The second field is a `PolyhedronEdge` flag. It indicates which sides of the triangle are visible.

DATA SIZE

```
12 + (numVertices * 12) + (numEdges * 16) + (numTriangles * 16)
```

DESCRIPTION

A polyhedron is composed of triangular faces. The basic idea is to have a list of points (the vertices), and then organize those points into a set of triangular

faces. The triangles are specified by indices into the array of points. Since typically a single point is a vertex of 3 or more triangles, referencing the points by index saves space. (For further details on the polyhedron geometry, see *3D Graphics Programming With QuickDraw 3D 1.5.* Also, see *develop* magazine by Apple Computer, Issue 28, Dec. 1996, "New QuickDraw 3D Geometries," p. 32.)

PARENT HIERARCHY

Shared, shape, geometry.

PARENT OBJECTS

None.

CHILD OBJECTS

In addition to the data in its root object, a polyhedron object can have as many as four subobjects: A `VertexAttributeSetList` for the `attributeSets` in the array of `TQ3Vertex3D`; a `GeometryAttributeSetList` for the `attributeSets` in the array of `TQ3PolyhedronEdgeData`; and a `FaceAttributeSetList` for the `attributeSets` in the array of `TQ3PolyhedronTriangleData`; and the usual geometry `AttributeSet`. See the appropriate sections of this document for descriptions of their formats.

DEFAULT SIZE

None.

EXAMPLE

The following example represents a complete metafile written by QD3D 1.5. For information about attribute arrays, see "Attribute Arrays," beginning on page 1350.

```
3DMetafile ( 1 5 Normal tableofcontents0> )
    polyhedron2:
    Container (
```

```
Polyhedron (
    6 0 4                # numVertices
                         # numEdges
                         # numTriangles
    -20 -20 0 -20 10 0 0 0 0 0 30 0 20 -20 0 20 10 0
    0 3 1 Edge01 | Edge12 | Edge20
    0 2 3 Edge01 | Edge12 | Edge20
    2 4 3 Edge01 | Edge12 | Edge20
    4 5 3 Edge01 | Edge12 | Edge20
)
Container (
    VertexAttributeSetList ( 6 Exclude 0 )
    attributeset3:
    Container (
        AttributeSet ( )
        AmbientCoefficient ( 1 )
        DiffuseColor ( 0 1 1 )
    )
    attributeset4:
    Container (
        AttributeSet ( )
        AmbientCoefficient ( 1 )
        DiffuseColor ( 0 0.95 1 )
    )
    attributeset5:
    Container (
        AttributeSet ( )
        AmbientCoefficient ( 1 )
        DiffuseColor ( 0 0.9 1 )
    )
    attributeset6:
    Container (
        AttributeSet ( )
        AmbientCoefficient ( 1 )
        DiffuseColor ( 0 0.85 1 )
    )
```

```
        attributeset7:
        Container (
            AttributeSet ( )
            AmbientCoefficient ( 1 )
            DiffuseColor ( 0 0.8 1 )
        )
        attributeset8:
        Container (
            AttributeSet ( )
            AmbientCoefficient ( 1 )
            DiffuseColor ( 0 0.75 1 )
        )
    )
    Container (
        FaceAttributeSetList ( 4 Exclude 0 )
        attributeset9:
        Container (
            AttributeSet ( )
            AmbientCoefficient ( 1 )
            DiffuseColor ( 0 0.7 1 )
        )
        attributeset10:
        Container (
            AttributeSet ( )
            AmbientCoefficient ( 1 )
            DiffuseColor ( 0 0.65 1 )
        )
        attributeset11:
        Container (
            AttributeSet ( )
            AmbientCoefficient ( 1 )
            DiffuseColor ( 0 0.6 1 )
        )
```

```
            attributeset12:
            Container (
                AttributeSet ( )
                AmbientCoefficient ( 1 )
                DiffuseColor ( 0 0.55 1 )
            )
        )
        attributeset13:
        Container (
            AttributeSet ( )
            AmbientCoefficient ( 0.5 )
            DiffuseColor ( 1 0 0 )
        )
    )
```

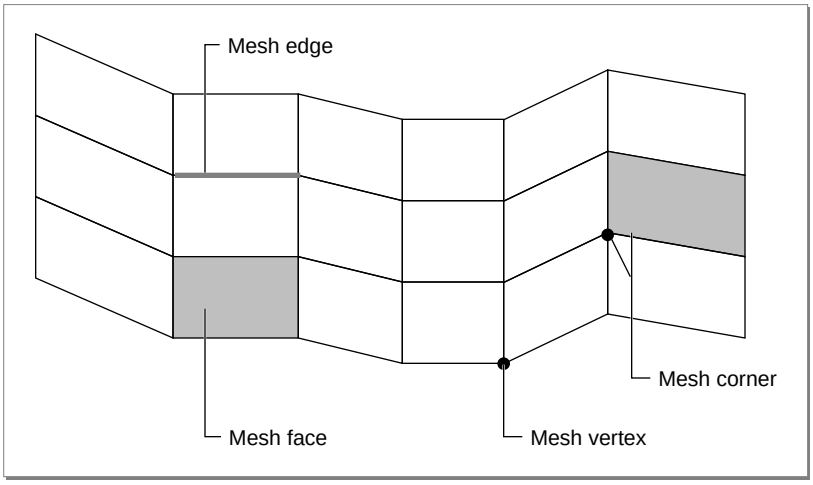
Offset	Hexadecimal code	ASCII
0000	3344 4D46 2020 2010 2001 2005 2020 2020	3DMF···· ····
0010	2020 2020 2020 2020 636E 7472 2020 02E4	···· ···· cntr ····
0020	706C 6864 2020 2094 2020 2006 2020 2020	plhd ···· ····
0030	2020 2004 C1A0 2020 C1A0 2020 2020 2020	···· ···· ····
0040	C1A0 2020 4120 2020 2020 2020 2020 2020	···· A··· ····
0050	2020 2020 2020 2020 2020 2020 41F0 2020	···· ···· ···· A···
0060	2020 2020 41A0 2020 C1A0 2020 2020 2020	···· A··· ····
0070	41A0 2020 4120 2020 2020 2020 2020 2020	A··· A··· ····
0080	2020 2003 2020 2001 2020 2007 2020 2020	···· ···· ····
0090	2020 2002 2020 2003 2020 2007 2020 2002	···· ···· ····
00A0	2020 2004 2020 2003 2020 2007 2020 2004	···· ···· ····
00B0	2020 2005 2020 2003 2020 2007 636E 7472	···· ···· ···· cntr
00C0	2020 0134 7661 736C 2020 200C 2020 2006	··· 4vasl ····
00D0	2020 2001 2020 2020 636E 7472 2020 2028	···· ···· cntr ··· (
00E0	6174 7472 2020 2020 6361 6D62 2020 2004	attr ···· camb ····
00F0	3F80 2020 6B64 6966 2020 200C 2020 2020	? ··· kdif ····
0100	3F80 2020 3F80 2020 636E 7472 2020 2028	? ··· ? ··· cntr ··· (
0110	6174 7472 2020 2020 6361 6D62 2020 2004	attr ···· camb ····
0120	3F80 2020 6B64 6966 2020 200C 2020 2020	? ··· kdif ····
0130	3F73 3333 3F80 2020 636E 7472 2020 2028	?s33? ··· cntr ··· (

```
0140     6174 7472 2020 2020 6361 6D62 2020 2004     attr····camb····
0150     3F80 2020 6B64 6966 2020 200C 2020 2020     ?···kdif········
0160     3F66 6666 3F80 2020 636E 7472 2020 2028     ?fff?···cntr···(
0170     6174 7472 2020 2020 6361 6D62 2020 2004     attr····camb····
0180     3F80 2020 6B64 6966 2020 200C 2020 2020     ?···kdif········
0190     3F59 999A 3F80 2020 636E 7472 2020 2028     ?Y··?···cntr···(
01A0     6174 7472 2020 2020 6361 6D62 2020 2004     attr····camb····
01B0     3F80 2020 6B64 6966 2020 200C 2020 2020     ?···kdif········
01C0     3F4C CCCD 3F80 2020 636E 7472 2020 2028     ?L··?···cntr···(
01D0     6174 7472 2020 2020 6361 6D62 2020 2004     attr····camb····
01E0     3F80 2020 6B64 6966 2020 200C 2020 2020     ?···kdif········
01F0     3F40 2020 3F80 2020 636E 7472 2020 20D4     ?@··?···cntr····
0200     6661 736C 2020 200C 2020 2004 2020 2001     fasl············
0210     2020 2020 636E 7472 2020 2028 6174 7472     ····cntr···(attr
0220     2020 2020 6361 6D62 2020 2004 3F80 2020     ····camb····?···
0230     6B64 6966 2020 200C 2020 2020 3F33 3333     kdif·····?333
0240     3F80 2020 636E 7472 2020 2028 6174 7472     ?···cntr···(attr
0250     2020 2020 6361 6D62 2020 2004 3F80 2020     ····camb····?···
0260     6B64 6966 2020 200C 2020 2020 3F26 6666     kdif·····?&ff
0270     3F80 2020 636E 7472 2020 2028 6174 7472     ?···cntr···(attr
0280     2020 2020 6361 6D62 2020 2004 3F80 2020     ····camb····?···
0290     6B64 6966 2020 200C 2020 2020 3F19 999A     kdif·····?···
02A0     3F80 2020 636E 7472 2020 2028 6174 7472     ?···cntr···(attr
02B0     2020 2020 6361 6D62 2020 2004 3F80 2020     ····camb····?···
02C0     6B64 6966 2020 200C 2020 2020 3F0C CCCD     kdif·····?···
02D0     3F80 2020 636E 7472 2020 2028 6174 7472     ?···cntr···(attr
02E0     2020 2020 6361 6D62 2020 2004 3F20 2020     ····camb····?···
02F0     6B64 6966 2020 200C 3F80 2020 2020 2020     kdif····?·······
0300     2020 2020                                   ····
```

Meshes

Figure 22-11 shows a mesh.

Figure 22-11 A mesh



LABELS

ASCII	`Mesh`
Binary	`mesh ( = 0x6D657368 )`

MESH FACE DATA TYPE

`Int`	`nFaceVertexIndices`
`Uns`	`faceVertexIndices[|nFaceVertexIndices|]`

Field descriptions

`nFaceVertexIndices`

An integer, the absolute value of which is equal to the number of indices to the vertices of a mesh face or mesh contour: that is, equal to the number of vertices of that face or contour. The value of this field may be positive or negative. A positive value indicates that this mesh face object specifies a face (to which attributes may be assigned). A negative value indicates that this mesh face object specifies a hole (here called a *contour*). The absolute value of the value in this field must be at least 3.

`faceVertexIndices[ ]`
: An array of indices to elements of the array `vertices[]`, where *i* is the index of `vertices[i]`. This array specifies a verticed object by giving the indices of its vertices. The specified object is either a face or a contour of the mesh, as determined by the value of `nVertices`. The number of fields of this array must equal the absolute value of the value of the `nVertices` field.

DESCRIPTION

The mesh face data type is used to specify a vertexed object and to specify whether that object is a face or a contour of a mesh. This data type occurs only as the value of a field in the `faces[]` array of a mesh specification.

DATA FORMAT

```
Uns32          nVertices
Vertex3D       vertices[nVertices]
Uns32          nFaces
Uns32          nContours
MeshFace       faces[nFaces + nContours]
```

Field descriptions

`nVertices`
: The number of vertices of the mesh. The value of this field must be at least 3.

`vertices[]`
: An array of vertices.

`nFaces`
: The number of faces of the mesh.

`nContours`
: The number of contours of the mesh (that is, the number of holes in the mesh).

`faces[]`
: An array of mesh face objects, each of which specifies either a face or a contour (hole) of the mesh. The size of this array is equal to the sum of the values of the `nVertices` and `nContours` fields. Each array element that specifies a face should precede all array elements that specify holes in that face; any such latter elements may occur in any order but should be grouped together and should precede any subsequent array element that specifies a face: if the value

of field *i* specifies a face intended to have *n* holes, then the objects that specify those holes must occupy the next *n* fields: that is, fields
i+1, ..., *i*+*n*.

DATA SIZE

```
sizeof(MeshFace) = fabs(Int) * 4
sizeof(Mesh) = 4 + nVertices * 12 + 8 +
sizeof(faces[0...nFaces+nContours-1])
```

DESCRIPTION

A mesh is an object defined by a collection of vertices, faces, and contours. Meshes may be used to model polyhedra, grids, and other faceted objects. A mesh may have a boundary. The term *contour* is used here to refer to a polygonal hole contained in a single face of a mesh. A mesh face (or contour) is a list of vertices that defines a polygonal facet. A face (or contour) need not be planar, and a contour and its surrounding face need not be coplanar; however, rendering of a mesh having a nonplanar face or contour, or having a contour not coplanar with its surrounding face, may lead to unexpected results.

The specification of a mesh includes an array of vertices and an array of faces and contours. The vertices of a mesh are indexed by array position; these indices are used to specify the faces and contours of that mesh. Faces and contours are also indexed by array position; this index does not distinguish between faces and contours. Both of these indices are used in the specification of child objects.

Attributes may be attached separately and selectively to the vertices, faces, face edges, and corners of a mesh.

DEFAULT SURFACE PARAMETERIZATION

None.

PARENT HIERARCHY

Shared, shape, geometry.

PARENT OBJECTS

None.

CHILD OBJECTS

Face attribute set list (optional), vertex attribute set list (optional), mesh corners (optional), mesh edges (optional). See "Mesh Corners," beginning on page 1343, and "Mesh Edges," beginning on page 1345, for descriptions of these objects.

EXAMPLE

```
Mesh (
   10                                  # nVertices
      -1 1 1                           # enumeration of vertices
      -1 1 -1
      1 1 -1
      1 -1 -1
      1 -1 1
      0 -1 1
      -1 -1 0
      -1 -1 -1
      1 1 1
      -1 0 1
   7                                   # nFaces
   0                                   # nContours
      3 6 5 9                          # enumeration of contours
      5 7 6 9 0 1
      4 2 3 7 1
      4 2 8 4 3
      4 1 0 8 2
      5 4 8 0 9 5
      5 3 4 5 6 7
)
```

DEFAULT SIZE

None.

Mesh Corners

LABELS

ASCII	`MeshCorners`
Binary	`crnr ( = 0x63726E72 )`

MESHCORNER DATA TYPE

`Uns32`	`vertexIndex`
`Uns32`	`nFaces`
`Uns32`	`faces[nFaces]`

Field descriptions

`vertexIndex`	The index of a vertex of the parent mesh.
`nFaces`	The number of faces of the parent mesh, sharing the vertex that is the value of the `vertexIndex` field, that are to be correlated with child objects of the mesh corners object. The value of this field must not exceed the number of faces of the parent mesh meeting at the vertex whose index is the value of the `vertexIndex` field.
`faces[]`	An array of face indices representing faces of the parent mesh. The vertex whose index is the value of the `vertexIndex` field must be among the vertices of each face of the parent mesh whose face index appears in this array. The number of fields of this array must equal the value of `nFaces`.

DATA FORMAT

`Uns32`	`nCorners`
`MeshCorner`	`corners[nCorners]`

Field descriptions

`nCorners`	The number of corners of the parent mesh treated by this mesh corners object.

`corners[]` An array of mesh corners data types. The elements of this array are correlated with attribute sets which occur as child objects of the mesh corners object. The number of fields of this array must equal the value of `nCorners`.

DATA SIZE

```
sizeof(MeshCorner) = 8 + nFaces * 4
sizeof(MeshCorners) = 4 + sizeof(corners[0...nCorners-1])
```

DESCRIPTION

The mesh corners object is used to attach more than one attribute set to a vertex of a mesh and to override other attributes inherited by a vertex or assigned to it elsewhere. You can use mesh corners in various ways: for example, to apply different normals and shadings in order to create the appearance of a sharp edge or peak. This object occurs only as a child object to a mesh and always has attribute sets as child objects of its own.

PARENT HIERARCHY

Data.

PARENT OBJECTS

Mesh (always).

CHILD OBJECTS

Attribute sets (always). The number of child objects is equal to the value of the `numCorners` field. Child objects are correlated with elements of the array `corners[]` in the order of their occurrence in the specification of the mesh corners object and its child objects; that is, the *i*th child object is correlated with the *i*th element of the array `corners[]`.

EXAMPLE

```
Container (
    Mesh (...)                  # parent mesh
    Container(
        MeshCorners (
            2                   # numCorners
            # Corner 0
            5 # vertexIndex
            2 # faces
            25 26 # face indices
            # Corner 1
            5 # vertexIndex
            2 # faces
            23 24 # face indices
        )
        Container (
            AttributeSet ( )
            Normal ( -0.2 0.8 0.3 )
        )
        Container (
            AttributeSet ( )
            Normal ( -0.7 -0.1 0.4 )
        )
    )
)
```

Mesh Edges

LABELS

ASCII	`MeshEdges`
Binary	`edge ( = 0x65646765 )`

MESH EDGE DATA TYPE

```
Uns32       vertexIndex1
Uns32       vertexIndex2
```

Field descriptions

`vertexIndex1` The smaller of the indices of the two vertices of the mesh edge. The indices are taken from the vertex index of the parent mesh.

`vertexIndex2` The larger of the indices of the two vertices of the mesh edge.

IMPORTANT

The edge defined by a mesh edge data type must be an edge of a face (not merely a contour) of the parent mesh. ▲

DATA FORMAT

```
Uns32       nEdges
MeshEdge    edges[nEdges]
```

Field descriptions

`nEdges` The number of edges of the parent mesh treated by this mesh edge object. The value in this field must be greater than 0 and less than or equal to the number of edges of faces of the parent mesh.

`edges[]` An array of mesh edge data types. The elements of this array are correlated with attribute sets that occur as child objects of the mesh edges object. The number of fields of this array must equal the value of `nEdges`.

DATA SIZE

```
4 + sizeof(edges[0...nEdges-1])
```

DESCRIPTION

The mesh edges object is used to attach attribute sets separately and selectively to one or more edges of faces of a mesh.

PARENT HIERARCHY

Data.

PARENT OBJECTS

Mesh (always).

CHILD OBJECTS

Attribute sets (always). The number of child objects is equal to the value of the `nEdges` field. Child objects are correlated with elements of the array `edges[]` in the order of their occurrence in the specification of the mesh edges object and its child objects; that is, the *i*th child object is correlated with the *i*th element of the array `edges[]`.

EXAMPLE

```
Container (
    Mesh ( ... )
    Container (
        MeshEdges (
        2                               # numEdges
        0 1                             # first edge
        1 3                             # second edge
    )
        Container (                     # first edge attribute set
            AttributeSet ( )
            DiffuseColor ( 0.2 0.8 0.3 )
        )
        Container (                     # second edge attribute set
            AttributeSet ( )
            DiffuseColor ( 0.8 0.2 0.3 )
        )
    )
)
```

Trimeshes

This section gives the 3DMF specification of the Trimesh object. Trimesh binary metafiles implement a simple type of compression, using a scheme described below.

Note
Normally a Trimesh also has one or more `AttributeArray` subobjects. For details, see "Attribute Arrays," beginning on page 1350. ◆

LABELS

ASCII	`TriMesh`
Binary	`tmsh` (`= 0x746D7368`)

AUXILIARY DATA STRUCTURES

The following auxiliary data structures are used to specify trimeshes. They are mirror images of the similarly-named structures in the file QD3DGeometry.h. See *3D Graphics Programming With QuickDraw 3D 1.5* for a detailed description.

```
typedef struct TriMeshTriangleData {
    Uns32  pointIndices[3];
} TriMeshTriangleData;

typedef struct TriMeshEdgeData {
    Uns32  pointIndices[2];
    Uns32  triangleIndices[2];
} TriMeshEdgeData;

typedef struct BoundingBox {
    Point3D    min;
    Point3D    max;
    Boolean    isEmpty;
} BoundingBox;
```

DATA FORMAT

```
Uns32                 numTriangles
Uns32                 numTriangleAttributeTypes
Uns32                 numEdges
Uns32                 numEdgeAttributeTypes
Uns32                 numPoints
Uns32                 numVertexAttributeTypes
TriMeshTriangleData       triangles[numTriangles]
TriMeshEdgeData           edges[numEdges]
Point3D                   points[numPoints]
BoundingBox               bBox
```

DATA COMPRESSION

The `triangles` and `edges` fields in a trimesh are compressed. These fields are arrays of indices into arrays of elements; the maximum value of the indices depends on the size of the array of elements. Hence, the maximum value of an index in the `TriMeshTriangleData` field `pointIndices` is equal to `numPoints`. The same is true for the `pointIndices` field of `TriMeshEdgeData`; the maximum value of an index in the `triangleIndices` field is `numTriangles`. Compression is performed as follows:

- If maximum value of `index` <= 0xFE, write `index` as `Uns8`
- If maximum value of `index` > 0xFE and <= 0xFFFE, write `index` as `Uns16`
- If maximum value of `index` > 0xFFFE, write `index` as `Uns32`

The `triangleIndices` field of `TriMeshEdgeData` has, in addition, the following special case. If a side of an edge does not have a triangle (as is the case if the edge is on a boundary), this is indicated in the trimesh data structure by the constant `kQ3ArrayIndexNULL`. Since this is a 32-bit quantity, you may need to compress it. Do this by writing `kQ3ArrayIndexNULL` as follows:

- If maximum value of `index` <= 0xFE, write 0xFF
- If maximum value of `index` > 0xFE and <= 0xFFFE, write 0xFFFF
- If maximum value of `index` > 0xFFFE, write `kQ3ArrayIndexNULL`

EXAMPLE

For an example of a trimesh text metafile, see "Attribute Arrays," beginning on page 1350.

Attribute Arrays

An attribute array contains all of the information contained in a single `TQ3TriMeshAttribute` (described below). It also contains information that identifies the location of this particular attribute array with respect to all of the other attribute arrays contained in the trimesh.

LABELS

ASCII	`AttributeArray`
Binary	`atar` (= `0x61746172`)

HEADER AND DATA

An attribute array has a header consisting of five numbers. The first number is type `TQ3Int32` and the last four are type `TQ3Uns32`.

Field descriptions

`TQ3Int32`	`AttributeType` field of `TQ3TriMeshAttribute`
`TQ3Uns32`	Reserved and currently unused; should always be 0.
`TQ3Uns32`	Call this field `positionOfArray`. 0 means this `TQ3TriMeshAttribute` is an element in the array of `TQ3TriMeshAttributes` pointed to by the `triangleAttributeTypes` field of `TQ3TriMeshData`. Similarly, 1 means it's an element in the `edgeAttributeTypes` array, and 2 means it's in `vertexAttributeTypes`.
`TQ3Uns32`	Call this field `positionInArray`. It specifies the element in the array singled out by field 3 above (`positionOfArray`). It's 0-based, so 0 means the first `TQ3TriMeshAttribute` in the array, 1 the second, etc.
`TQ3Uns32`	A flag; 0 if the `attributeUseArray` field of `TQ3TriMeshAttribute` is `NULL`, 1 otherwise. If it's 1, the `attributeUseArray` data follows immediately. It consists of one `TQ3Uns8` for each attribute in the attribute array.

The header is followed by the data. For built-in attributes, if the attribute type is anything other than `kQ3AttributeTypeSurfaceShader`, the data is part of the attribute array's root object, so it immediately follows the header in a continuous stream. The size of the data is determined by the `AttributeType` field. If the attribute type is `kQ3AttributeTypeSurfaceShader`, then the texture shaders appear as subobjects, and the attribute array root contains only the header. If the attribute is a custom attribute, then the attributes also appear as subobjects. This is the case even if the data for the custom attribute is simple, such as one `float` value.

EXAMPLE

The following is an example of a metafile that describes a trimesh with a moderately complex set of attribute arrays. This example represents a complete metafile written by QD3D 1.5.

```
3DMetafile ( 1 5 Normal tableofcontents0> )
    trimesh2:
    Container (
        TriMesh (
            4 2 3 1 6 1       # numTriangles
                              # numTriangleAttributeTypes
                              # numEdges
                              # numEdgeAttributeTypes
                              # numPoints
                              # numVertexAttributeTypes
            0 3 1
            0 2 3
            2 4 3
            4 5 3
            0 3 0 1
            2 3 1 2
            3 4 2 3
            -20 -20 0 -20 10 0 0 0 0
            0 30 0 20 -20 0 20 10 0
            -20 -20 -1 20 30 1 False )

        AttributeArray (
            6 0 0 0 0
```

```
            0.2 0.6 0.8
            1 0 0
            0 1 0.5
            0 0.2 1
        )
        Container (
            AttributeArray ( 11 0 0 1 0
            )
            textureshader3:
            Container (
                TextureShader ( )
                pixmaptexture4:
                PixmapTexture (
                    8 8 36 32         # RGB32 BigEndian BigEndian
                    0x000000FF000000FF0000FF000000FF00
                    0x0000FF000000FF000000FF000000FF00
                    0x01909860000000FF000000FF0000FF00
                    0x0000FF000000FF000000FF000000FF00
                    0x0000FF00047F00000000FF000000FF00
                    0x0000FF000000FF000000FF000000FF00
                    0x0000FF000000FF00000000000000FF00
                    0x0000FF000000FF000000FF000000FF00
                    0x0000FF000000FF000000FF00B7A401E6
                    0x0000FF000000FF000000FF000000FF00
                    0x0000FF000000FF000000FF000000FF00
                    0x0000005000000FF000000FF000000FF00
                    0x0000FF000000FF000000FF000000FF00
                    0x0000FF00000000000000FF000000FF00
                    0x0000FF000000FF000000FF000000FF00
                    0x0000FF000000FF0001E856400000FF00
                    0x0000FF000000FF000000FF000000FF00
                    0x0000FF000000FF000000FF0000000000
                )
            )


            textureshader5:
            Container (
```

```
            TextureShader ( )
            pixmaptexture6:
            PixmapTexture (
                8 8 36 32         # RGB32 BigEndian BigEndian
                0x00FFFF0000FFFF0000FFFF0000FFFF00
                0x00FFFF0000FFFF0000FF000000FF0000
                0x0012602000FFFF0000FFFF0000FFFF00
                0x00FFFF0000FFFF0000FFFF0000FF0000
                0x00FF000001E857B000FFFF0000FFFF00
                0x00FFFF0000FFFF0000FFFF0000FFFF00
                0x00FF000000FF0000AB00001400FFFF00
                0x00FFFF0000FFFF0000FFFF0000FFFF00
                0x00FFFF0000FF000000FF000000000030
                0x00FFFF0000FFFF0000FFFF0000FFFF00
                0x00FFFF0000FFFF0000FF000000FF0000
                0x0000000000FFFF0000FFFF0000FFFF00
                0x00FFFF0000FFFF0000FFFF0000FF0000
                0x00FF000001E6B75C00FF000000FF0000
                0x00FF000000FF000000FF000000FF0000
                0x00FF000000FF00000001000000FF0000
                0x00FF000000FF000000FF000000FF0000
                0x00FF000000FF000000FF0000000001E6
            )
        )
        Reference ( 1 )
        Reference ( 2 )
    )
    AttributeArray (
        7 0 1 0 0
        0.3
        0.4
        0.6
    )



    AttributeArray (
        2 0 2 0 0
```

```
            0 0
            0 1
            1 0
            1 1
            0 0
            0 1
        )
        attributeset7:
        Container (
            AttributeSet ( )
            AmbientCoefficient ( 0.5 )
            DiffuseColor ( 1 0 0 )
        )
    )
    tableofcontents0:
    TableOfContents (
        tableofcontents1>        # next TOC
        3                        # reference seed
        -1                       # typeSeed
        1                        # tocEntryType
        16                       # tocEntrySize
        2                        # nEntries
        1 textureshader3>
        TextureShader
        2 textureshader5>
        TextureShader
    )
```

Offset	Hexadecimal code	ASCII
0000	3344 4D46 2020 2010 2001 2005 2020 2020	3DMF·····
0010	2020 2020 2020 0490 636E 7472 2020 0470	·····cntr···p
0020	746D 7368 2020 2094 2020 2004 2020 2002	tmsh·····
0030	2020 2003 2020 2001 2020 2006 2020 2001	·····
0040	2003 0120 0203 0204 0304 0503 2003 2001	·····
0050	0203 0102 0304 0203 C1A0 2020 C1A0 2020	·····
0060	2020 2020 C1A0 2020 4120 2020 2020 2020	·····A·····
0070	2020 2020 2020 2020 2020 2020 2020 2020	·····

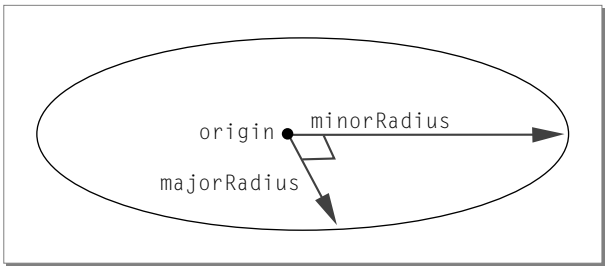
```
0080    41F0 2020 2020 2020 41A0 2020 C1A0 2020    A·····A·····
0090    2020 2020 41A0 2020 4120 2020 2020 2020    ····A···A·····
00A0    C1A0 2020 C1A0 2020 BF80 2020 41A0 2020    ··········A···
00B0    41F0 2020 3F80 2020 2020 2020 6174 6172    A···?·····atar
00C0    2020 2044 2020 2006 2020 2020 2020 2020    ···D··········
00D0    2020 2020 2020 2020 3E4C CCCD 3F19 999A    ·····>L··?···
00E0    3F4C CCCD 3F80 2020 2020 2020 2020 2020    ?L··?·········
00F0    2020 2020 3F80 2020 3F20 2020 2020 2020    ····?···?·····
0100    3E4C CCCD 3F80 2020 636E 7472 2020 02DC    >L··?···cntr····
0110    6174 6172 2020 2014 2020 200B 2020 2020    atar··········
0120    2020 2020 2020 2001 2020 2020 636E 7472    ··········cntr
0130    2020 014C 7478 7375 2020 2020 7478 706D    ···Ltxsu····txpm
0140    2020 013C 2020 2008 2020 2008 2020 2024    ···<·········$
0150    2020 2020 2020 2020 2020 2020 2020 2020    ··············
0160    2020 20FF 2020 20FF 2020 FF20 2020 FF20    ··············
0170    2020 FF20 2020 FF20 2020 FF20 2020 FF20    ··············
0180    0190 9860 2020 20FF 2020 20FF 2020 FF20    ···`··········
0190    2020 FF20 2020 FF20 2020 FF20 2020 FF20    ··············
01A0    2020 FF20 047F 2020 2020 FF20 2020 FF20    ··············
01B0    2020 FF20 2020 FF20 2020 FF20 2020 FF20    ··············
01C0    2020 FF20 2020 FF20 2020 2020 2020 FF20    ··············
01D0    2020 FF20 2020 FF20 2020 FF20 2020 FF20    ··············
01E0    2020 FF20 2020 FF20 2020 FF20 B7A4 01E6    ··············
01F0    2020 FF20 2020 FF20 2020 FF20 2020 FF20    ··············
0200    2020 FF20 2020 FF20 2020 FF20 2020 FF20    ··············
0210    2020 2050 2020 FF20 2020 FF20 2020 FF20    ···P··········
0220    2020 FF20 2020 FF20 2020 FF20 2020 FF20    ··············
0230    2020 FF20 2020 2020 2020 FF20 2020 FF20    ··············
0240    2020 FF20 2020 FF20 2020 FF20 2020 FF20    ··············
0250    2020 FF20 2020 FF20 01E8 5640 2020 FF20    ··········V@····
0260    2020 FF20 2020 FF20 2020 FF20 2020 FF20    ··············
0270    2020 FF20 2020 FF20 2020 FF20 2020 2020    ··············
0280    636E 7472 2020 014C 7478 7375 2020 2020    cntr···Ltxsu····
0290    7478 706D 2020 013C 2020 2008 2020 2008    txpm···<·····
02A0    2020 2024 2020 2020 2020 2020 2020 2020    ···$··········
```

```
02B0     2020 2020 20FF FF20 20FF FF20 20FF FF20     ·····················
02C0     20FF FF20 20FF FF20 20FF FF20 20FF 2020     ·····················
02D0     20FF 2020 2012 6020 20FF FF20 20FF FF20     ·····`···············
02E0     20FF FF20 20FF FF20 20FF FF20 20FF FF20     ·····················
02F0     20FF 2020 20FF 2020 01E8 57B0 20FF FF20     ··········W·····
0300     20FF FF20 20FF FF20 20FF FF20 20FF FF20     ·····················
0310     20FF FF20 20FF 2020 20FF 2020 AB20 2014     ·····················
0320     20FF FF20 20FF FF20 20FF FF20 20FF FF20     ·····················
0330     20FF FF20 20FF FF20 20FF 2020 20FF 2020     ·····················
0340     2020 2030 20FF FF20 20FF FF20 20FF FF20     ···0·····················
0350     20FF FF20 20FF FF20 20FF FF20 20FF 2020     ·····················
0360     20FF 2020 2020 2020 20FF FF20 20FF FF20     ·····················
0370     20FF FF20 20FF FF20 20FF FF20 20FF FF20     ·····················
0380     20FF 2020 20FF 2020 01E6 B75C 20FF 2020     ··········\····
0390     20FF 2020 20FF 2020 20FF 2020 20FF 2020     ·····················
03A0     20FF 2020 20FF 2020 20FF 2020 2001 2020     ·····················
03B0     20FF 2020 20FF 2020 20FF 2020 20FF 2020     ·····················
03C0     20FF 2020 20FF 2020 20FF 2020 20FF 2020     ·····················
03D0     2020 01E6 7266 726E 2020 2004 2020 2001     ····rfrn·····
03E0     7266 726E 2020 2004 2020 2002 6174 6172     rfrn·····atar
03F0     2020 2020 2020 2007 2020 2020 2020 2001     ·····················
0400     2020 2020 2020 2020 3E99 999A 3ECC CCCD     ·····>···>···
0410     3F19 999A 6174 6172 2020 2044 2020 2002     ?···atar···D····
0420     2020 2020 2020 2002 2020 2020 2020 2020     ·····················
0430     2020 2020 2020 2020 2020 2020 3F80 2020     ··········?···
0440     3F80 2020 2020 2020 3F80 2020 3F80 2020     ?·····?···?···
0450     2020 2020 2020 2020 2020 2020 3F80 2020     ··········?···
0460     636E 7472 2020 2028 6174 7472 2020 2020     cntr···(attr····
0470     6361 6D62 2020 2004 3F20 2020 6B64 6966     camb····?···kdif
0480     2020 200C 3F80 2020 2020 2020 2020 2020     ····?···········
0490     746F 6320 2020 203C 2020 2020 2020 2020     toc····<·····
04A0     2020 2003 FFFF FFFF 2020 2001 2020 2010     ·····················
04B0     2020 2002 2020 2001 2020 2020 2020 012C     ·····················,
04C0     7478 7375 2020 2002 2020 2020 2020 0280     txsu·····
04D0     7478 7375                                   txsu
```

Ellipses

Figure 22-12 shows an ellipse.

Figure 22-12 An ellipse



LABELS

ASCII	`Ellipse`
Binary	`elps` (`= 0x656C7073`)

DATA FORMAT

`Vector3D`	`majorAxis`
`Vector3D`	`minorAxis`
`Point3D`	`origin`
`Float32`	`uMin`
`Float32`	`uMax`

Field descriptions

`majorAxis`	The (semi-) major axis of the ellipse.
`minorAxis`	The (semi-) minor axis of the ellipse.
`origin`	The center of the ellipse.
`uMin`	Minimum parametric limit value, assuming parametrization of the angle between the major axis and the vector from origin to the circumference. A value of u = 0 corresponds to 0 radians, and u = 1 corresponds to 2 π radians. This is used to create partial ellipses. The basic

	idea is that only the part of the ellipse between uMin and uMax is drawn. For details, see the QD3D documentation or *develop* magazine, Dec. 96. Must be 0 in Version 1.5.
`uMax`	Maximum parametric limit value; see uMin above. Must be 1 in Version 1.5.

DATA SIZE

0 or 44

DESCRIPTION

An ellipse is a two-dimensional object defined by an origin (that is, the center of the ellipse) and two orthogonal vectors that define the major and minor radii of the ellipse. The origin and the two endpoints of the major and minor radii define the plane in which the ellipse lies. Attributes may be assigned only to the entire ellipse.

DEFAULT SURFACE PARAMETERIZATION

None.

PARENT HIERARCHY

Shared, shape, geometry.

PARENT OBJECTS

None.

CHILD OBJECTS

Attribute set (optional).

EXAMPLE

```
Ellipse (
    2   0   0          #majorRadius
    0   1   0          #minorRadius
    0   0   0          #origin
    0                  #uMin
    1                  #uMax
)
```

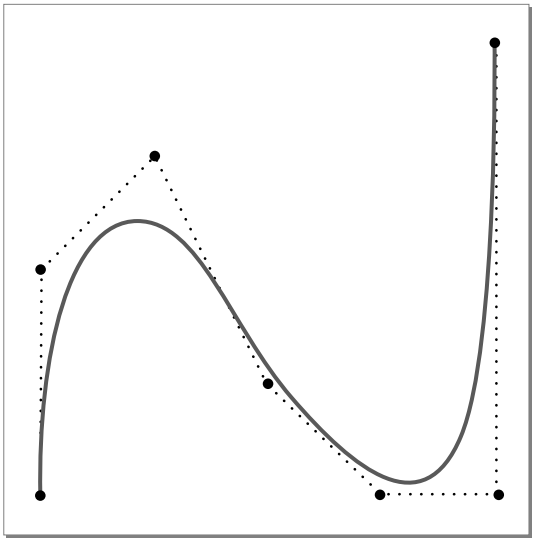
DEFAULT SIZE

For objects of size 0, the default is shown in the example above.

NURB Curves

Figure 22-13 shows a NURB curve.

Figure 22-13 A NURB curve



LABELS

ASCII	`NURBCurve`
Binary	`nrbc (= 0x6E726263)`

DATA FORMAT

```
Uns32                   order
Uns32                   nPoints
RationalPoint4D         points[nPoints]
Float32                 knots[order + nPoints]
```

Field descriptions

`order`	The order of the NURB curve. For NURB curves defined by ratios of cubic B-spline polynomials, the order is 4. In general, the order of a NURB curve defined by polynomial equations of degree *n* is *n*+1. The value of this field must be greater than 1.
`nPoints`	The number of control points that define the NURB curve. The value of this field must be greater than 1.
`points[]`	An array of rational four-dimensional control points that define the NURB curve. The *w* coordinate of each control point must be greater than 0.
`knots[]`	An array of knots that define the NURB curve. The number of knots in a NURB curve is the sum of the values in the `order` and `nPoints` fields. The values in this array must be nondecreasing. Successive values may be equal, up to a multiplicity equivalent to the order of the curve; that is, if the order of a NURB curve is *n*, then at most *n* successive values may be equal.

DATA SIZE

```
8 + (nPoints * 16) + ((nPoints + order) * 4)
```

DESCRIPTION

A nonuniform rational B-spline (NURB) curve is a three-dimensional projection of a four-dimensional curve. A NURB curve is specified by its order, the

number of control points used to define it, the control points themselves, and the knots used to define it. Attributes may be applied only to the entire NURB curve.

DEFAULT SURFACE PARAMETERIZATION

None.

PARENT HIERARCHY

Shared, shape, geometry.

PARENT OBJECTS

None.

CHILD OBJECTS

Attribute set (optional).

EXAMPLE

```
NURBCurve (
   4                                        # order
   7                                        # nPoints
   0 0 0 1                                  # points
   1 1 0 1
   2 0 0 1
   3 1 0 1
   4 0 0 1
   5 1 0 1
   6 0 0 1
   0 0 0 0 0.25 0.5 0.75 1 1 1 1            # knots
)
```

DEFAULT SIZE

None.

2D NURB Curves

LABELS

ASCII	`NURBCurve2D`
Binary	`nb2c` (`= 0x6E623263`)

DATA FORMAT

`Uns32`	`order`
`Uns32`	`nPoints`
`RationalPoint3D`	`points[nPoints]`
`Float32`	`knots[order + nPoints]`

Field descriptions

`order`	The order of the NURB curve. In general, the order of a NURB curve defined by polynomial equations of degree *n* is *n*+1. The value of this field must be greater than 1.
`nPoints`	The number of control points that define the 2D NURB curve. The value of this field must be greater than 1.
`points[]`	An array of three-dimensional control points that define the 2D NURB curve. The *z* coordinate of each point in this array must be greater than 0.
`knots[]`	An array of knots that define the 2D NURB curve. The number of knots in a NURB curve is the sum of the values in the `order` and `nPoints` fields. The values in this array must be nondecreasing, but successive values may be equal.

DATA SIZE

```
8 + 12 * nPoints + 4 * (order + nPoints)
```

DESCRIPTION

See "NURB Curves," beginning on page 1359 for a general description of NURB curves. 2D NURB curves occur only as child objects to trim loop objects, and

trim loop objects occur only as child objects to NURB patches. This object is the only two-dimensional curve permitted by 3D metafile Version 1.0.

DEFAULT SURFACE PARAMETERIZATION

None.

PARENT HIERARCHY

Data.

PARENT OBJECTS

Trim loop object (always).

CHILD OBJECTS

None.

DEFAULT SIZE

None.

Trim Loops

LABELS

ASCII	`TrimLoop`
Binary	`trml ( = 0x74726D6C )`

DATA FORMAT

None.

DATA SIZE

0

DESCRIPTION

A trim loop object is used to bind two-dimensional curves to a NURB patch for the purpose of trimming that patch. As of this release, only 2D NURB curves may be used for trimming.

Trimming curves are attached to a NURB patch by placing them in a container the root object of which is a trim loop object and placing that container in a further container together with the relevant NURB patch.

The two-dimensional curves governed by a trim loop object must form a sequence such that the last control point of the *i*th curve is also the first control point of the *i*+1st curve, and the last control point of the last curve is also the first control point of the first curve.

DEFAULT SURFACE PARAMETERIZATION

None.

PARENT HIERARCHY

Data.

PARENT OBJECTS

NURB patch (always).

CHILD OBJECTS

2D NURB curves (required). A trim loop object may have several child objects.

EXAMPLE

```
Container (
    NURBPatch (...)
    Container (
    TrimLoop ( )
        NURBCurve2D (...)
        .
        .
```

```
        .
        NURBCurve2D (...)
    )
```

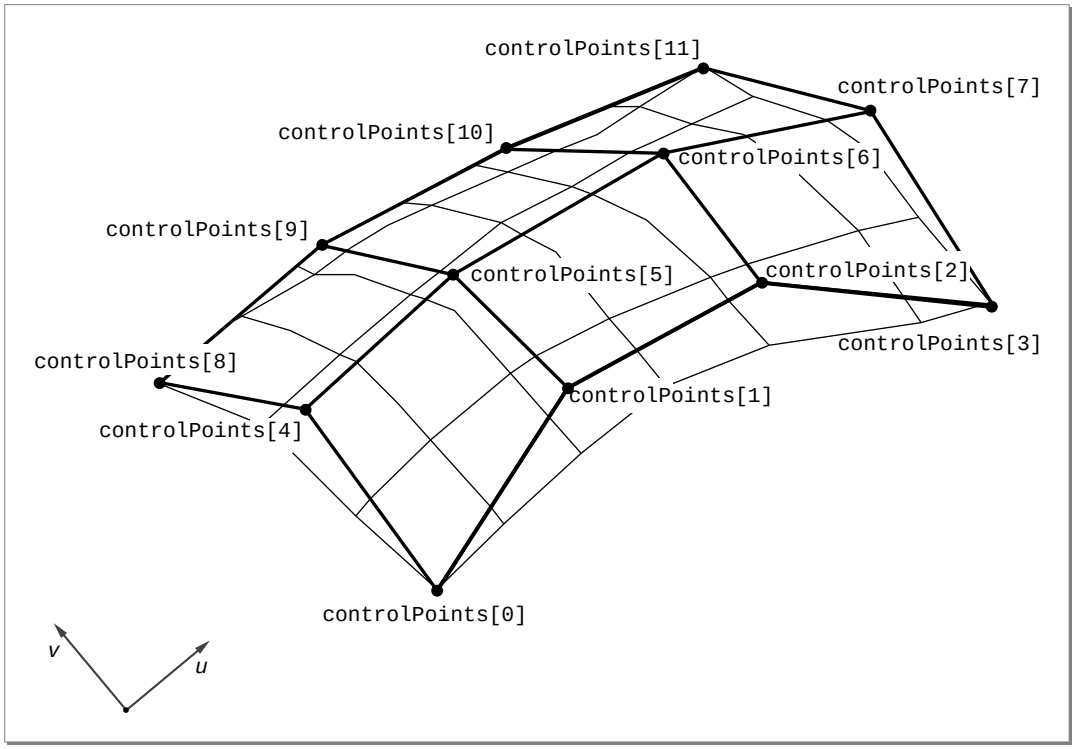
DEFAULT SIZE

None.

NURB Patches

Figure 22-14 shows a NURB patch.

Figure 22-14 A NURB patch



LABELS

ASCII	`NURBPatch`
Binary	`nrbp` (= `0x6E726270`)

DATA FORMAT

`Uns32`	`uOrder`
`Uns32`	`vOrder`
`Uns32`	`numMPoints`
`Uns32`	`numNPoints`
`RationalPoint4D`	`points[numMPoints * numNPoints]`
`Float32`	`uKnots[uOrder + numMPoints]`
`Float32`	`vKnots[vOrder + numNPoints]`

Field descriptions

`uOrder`	The order of a NURB patch in the *u* parametric direction. For NURB patches defined by ratios of B-spline polynomials that are cubic in *u*, the order is 4. In general, the order of a NURB patch defined by polynomial equations in which *u* is of degree *n* is *n*+1.
`vOrder`	The order of a NURB patch in the *v* parametric direction. For NURB patches defined by ratios of B-spline polynomials that are cubic in *v*, the order is 4. In general, the order of a NURB patch defined by polynomial equations in which *v* is of degree *n* is *n*+1.
`numMPoints`	The number of control points in the *u* parametric direction. The value of this field must be greater than 1.
`numNPoints`	The number of control points in the *v* parametric direction. The value of this field must be greater than 1.
`points[]`	An array of rational four-dimensional control points that define the NURB patch. The size of this array is as indicated in the data format.
`uKnots[]`	An array of knots in the *u* parametric direction that define the NURB patch. The values in this array must be nondecreasing, but successive values may be equal. The size of this array is as indicated in the data format.

`vKnots[]` An array of knots in the *v* parametric direction that define the NURB patch. The values in this array must be nondecreasing, but successive values may be equal. The size of this array is as indicated in the data format.

DATA SIZE

```
16 + (16 * numMPoints * numNPoints) + (uOrder + numNPoints + vOrder + numMPoints) * 4
```

DESCRIPTION

A NURB patch is a three-dimensional surface defined by ratios of B-spline surfaces, which are three-dimensional analogs of B-spline curves.

DEFAULT SURFACE PARAMETERIZATION

The default surface parameterization of a NURB patch is as shown in Figure 22-14.

PARENT HIERARCHY

Shared, shape, geometry.

PARENT OBJECTS

None.

CHILD OBJECTS

Trim curves (optional). A trim curves object is a collection of two-dimensional NURB curves that are used to trim a NURB surface. See “Trim Loops,” beginning on page 1363, and “2D NURB Curves,” beginning on page 1362, for descriptions of these objects.

EXAMPLE

```
NURBPatch (
   4                                                    #uOrder
   4                                                    #vOrder
   4                                                    #numMPoints
   4                                                    #numNPoints

   -2 2 0 1   -1 2 0 1   1 2 0 1   2 2 0 1             #points
   -2 2 0 1   -1 2 0 1   1 0 5 1   2 2 0 1
   -2 -2 0 1  -1 -2 0 1  1 -2 0 1  2 -2 0 1
   -2 -2 0 1  -1 -2 0 1  1 -2 0 1  2 -2 0 1

   0 0 0 0 1 1 1 1                                      #uKnots
   0 0 0 0 1 1 1 1                                      #vKnots
)
```

Note
The control points of a NURB patch are listed in a rectangular order, first in order of increasing *v*, then in order of increasing *u*. ◆

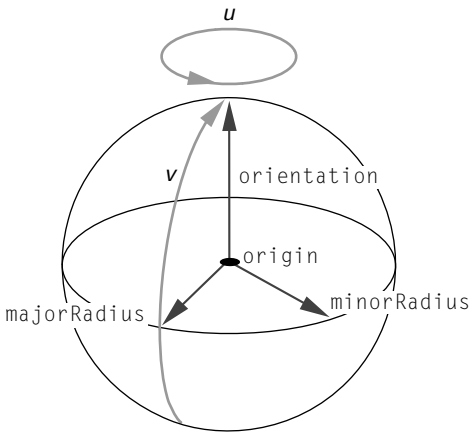
DEFAULT SIZE

None.

Ellipsoids

Figure 22-15 shows an ellipsoid.

Figure 22-15 An ellipsoid



LABELS

ASCII	`Ellipsoid`
Binary	`elpd` (= `0x656C7064`)

DATA FORMAT

`Vector3D`	`orientation`
`Vector3D`	`majorRadius`
`Vector3D`	`minorRadius`
`Point3D`	`origin`
`Float32`	`uMin`
`Float32`	`uMax`
`Float32`	`vMin`
`Float32`	`vMax`

Field descriptions

`orientation`	The orientation of the ellipsoid.
`majorRadius`	The major radius of the ellipsoid.
`minorRadius`	The minor radius of the ellipsoid.

`origin`	The origin (that is, the center) of the ellipsoid.
`uMin`	Minimum parametric limit value for u. To understand u, first consider u for the ellipse determined by `majorRadius` and `minorRadius`.The value of u on this sub-ellipse is parametrized by the angle between the major axis and the vector from the origin to the circumference of the ellipse. The value u = 0 corresponds to 0 radians, and u = 1 corresponds to 2 π radians.The values `uMin` and `uMax` are used to create partial ellipses. The basic idea is that only the part of the ellipse between `uMin` and `uMax` is drawn. For such a particular partial sub-ellipse, the partial ellipsoid can be thought of as the result of keeping `uMin` and `uMax` fixed but letting v vary through its admissible range. For details, see the QD3D documentation or *develop* magazine, Dec. 96. Must be 0 in Version 1.5.
`uMax`	Maximum parametric limit value in u direction; see `uMin` above. Must be 1 in Version 1.5.
`vMin`	Minimum parametric limit value for v. To understand v, first consider v for the ellipse determined by `majorRadius` and orientation. The value of v on this sub-ellipse is parametrized by the angle between the major axis and the vector from the origin to the circumference of the ellipse. The value v = 0 corresponds to 0 radians, and v = 1 corresponds to π (not 2π) radians. The values `vMin` and `vMax` are used to create partial ellipses of this sub-ellipse. The basic idea is that only the part of the ellipse between `vMin` and `vMax` is drawn. Let *pMin* be the endpoint of the partial sub-ellipse that corresponds to `vMin`, and *pMax* be the endpoint that corresponds to `vMax`. Then the partial ellipsoid is the result of truncating the whole ellipsoid by a two planes parallel to the plane specified by `majorAxis` and `minorAxis`: one of these planes passes through `vMin` and the other plane passes through `vMax`. For details, see the QD3D documentation or *develop* magazine, Dec. 96. Must be 0 in Version 1.5
`vMax`	Maximum parametric limit value in v direction; see `vMin` above. Must be 1 in Version 1.5.

DATA SIZE

0 or 64

DESCRIPTION

An ellipsoid is a three-dimensional object defined by an origin (that is, the center of the ellipsoid) and three pairwise orthogonal vectors that define the orientation and the major and minor radii of the ellipsoid.

DEFAULT SURFACE PARAMETERIZATION

The default surface parameterization for an ellipsoid is as shown in Figure 22-15. To the left of the major radius, $v = 0$; to the right of the major radius, $v = 1$. At the (top of the) orientation vector, and at the bottom of the ellipsoid, $u = 0$.

PARENT HIERARCHY

Shared, shape, geometry.

PARENT OBJECTS

None.

CHILD OBJECTS

Attribute set (optional).

EXAMPLE

```
Ellipsoid ( )
Ellipsoid (
2 0 0
0 1 0
0 0 1
0 0 0
0      # uMin
1      # uMax
0      # vMin
```

```
1      # vMax
)
Container (
    Ellipsoid ( )
    Container (
        AttributeSet ( )
        DiffuseColor ( 1 1 0 )
    )
)
```

DEFAULT SIZE

For objects of size 0, the default is:

```
1 0 0
0 1 0
0 0 1
0 0 0
0
1
0
1
```

Caps

LABELS

ASCII	`Caps`
Binary	`caps ( = 0x63617073 )`

CAPS FLAGS

`None`	`0x00000000`
`Top`	`0x00000001`
`Bottom`	`0x00000002`

Constant descriptions

`None`	The parent cone or cylinder shall not have any caps.

`Top`	The parent cylinder shall have a cap at the end opposite to its base.
`Bottom`	The parent cone or cylinder shall have a cap at its base.

DATA FORMAT

`CapsFlags`	`caps`

Field descriptions

`caps`	A bitfield expression specifying one or more flags.

DATA SIZE

4

DESCRIPTION

A cap is a plane figure having the shape of an oval that closes the base of a cone or one end of a cylinder. A cone and a cylinder may each be supplied with a bottom cap. Only a cylinder may be supplied with a top cap. The length of the semimajor axis of a cap is equal to the length of the major radius of its parent object; the length of the semiminor axis of a cap is equal to the length of the minor radius of its parent object. A cap lies in a plane perpendicular to the orientation vector of its parent object. The center of a top cap is at the end of the orientation vector of its parent object; the center of a bottom cap is at the origin of its parent object. A separate attribute set may be assigned to each cap of an object having one or more caps.

PARENT HIERARCHY

Data, cap data.

PARENT OBJECTS

Cone, cylinder (always).

CHILD OBJECTS

None.

EXAMPLE

```
Container (
    Cone ( ... )
    Caps ( Top | Bottom )
    Container (
        BottomCapAttributeSet ( )
        DiffuseColor ( 0 1 0 )
    )
)
```

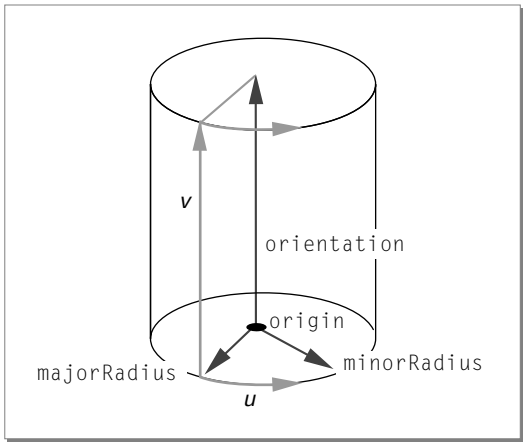
DEFAULT VALUE

None.

Cylinders

Figure 22-16 shows a cylinder.

Figure 22-16 A cylinder



LABELS

ASCII	`Cylinder`
Binary	`cyln` (`= 0x63796C6E`)

DATA FORMAT

`Vector3D`	`orientation`
`Vector3D`	`majorRadius`
`Vector3D`	`minorRadius`
`Point3D`	`origin`
`Float32`	`uMin`
`Float32`	`uMax`
`Float32`	`vMin`
`Float32`	`vMax`

Field descriptions

`orientation`	The orientation of the cylinder.
`majorRadius`	The major radius of the cylinder.
`minorRadius`	The minor radius of the cylinder.
`origin`	The origin (that is, the center of the base) of the cylinder.
`uMin`	Minimum parametric limit value for u. To understand u, first consider u for the ellipse determined by `majorRadius` and `minorRadius`. The value of u on this ellipse is parametrized by the angle between the major axis and the vector from the origin to the circumference of the ellipse. The value u = 0 corresponds to 0 radians, and u = 1 corresponds to 2 π radians. The values `uMin` and `uMax` are used to create partial ellipses. The basic idea is that only the part of the ellipse between `uMin` and `uMax` is drawn. For such a particular partial ellipse, the partial cylinder can be thought of as the result of keeping `uMin` and `uMax` fixed but letting v vary through its admissible range. The result looks like a cylinder with a wedge taken out. For details, see the QD3D documentation or *develop* magazine, Dec. 96. Must be 0 in Version 1.5.
`uMax`	Maximum parametric limit value in u direction; see `uMin` above. Must be 1 in Version 1.5.

`vMin`	Minimum parametric limit value in v direction. The value of v can be viewed as arc length parametrization of the orientation vector, ranging from 0 at the origin to 1 at the tip. If `vMin` is not 0, then a cylinder-shaped slice will be removed from the bottom of the cylinder. Must be 0 in Version 1.5.
`vMax`	Maximum parametric limit value in v direction; see `vMin` above. Must be 1 in Version 1.5.

DATA SIZE

0 or 64

DESCRIPTION

A cylinder is a three-dimensional object defined by an origin (that is, the center of the cylinder) and three mutually perpendicular vectors that define the orientation and the major and minor radii of the cylinder. A cylinder may include a top cap, a bottom cap, or both. Attributes may be assigned to each included cap, to the face of the cylinder, and to the entire cylinder.

DEFAULT SURFACE PARAMETERIZATION

The default surface parameterization for a cylinder is as shown in Figure 22-16.

PARENT HIERARCHY

Shared, shape, geometry.

PARENT OBJECTS

None.

CHILD OBJECTS

Caps (top), top cap attribute set, caps (bottom), bottom cap attribute set, face cap attribute set, attribute set. All child objects are optional.

EXAMPLE

```
Cylinder ( )

Cylinder (
    0 2 0
    0 1 0
    0 0 1
    0 0 0
    0       # uMin
    1       # uMax
    0       # vMin
    1       # vMax
)
Container (
    Cylinder ( )
    Caps ( Bottom | Top )
    Container (
        BottomCapAttributeSet ( )
        Container (
        AttributeSet ( )
        DiffuseColor ( 0 1 0 )
        )
    )
    Container (
        FaceCapAttributeSet ( )
        Container (
            AttributeSet ( )
            DiffuseColor ( 1 0 1 )
        )
    )
    Container (
        TopCapAttributeSet ( )
        Container (
            AttributeSet ( )
            DiffuseColor ( 1 1 0 )
```

```
        )
    )
)
```

Note
In the above example, color attributes are attached to the surface of the cylinder very indirectly. As you see, color objects are elements of ordinary attribute sets rather than of cap attribute sets. Those attribute sets are elements of containers, which, in turn, are elements of cap attribute sets. The cap attribute sets serve to bind the ordinary attribute sets to the caps of the cylinder. ◆

DEFAULT SIZE

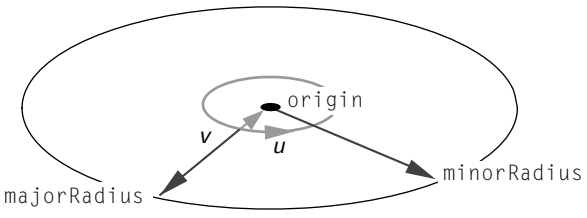
For objects of size 0, the default is:

```
1 0 0
0 1 0
0 0 1
0 0 0
0
1
0
1
```

Disks

Figure 22-17 shows a disk.

Figure 22-17 A disk



LABELS

ASCII	`Disk`
Binary	`disk ( = 0x6469736B )`

DATA FORMAT

`Vector3D`	`majorRadius`
`Vector3D`	`minorRadius`
`Point3D`	`origin`
`Float32`	`uMin`
`Float32`	`uMax`

Field descriptions

`majorRadius`	The major radius of the disk.
`minorRadius`	The minor radius of the disk.
`origin`	The center of the disk.
`uMin`	Minimum parametric limit value, assuming parametrization of the angle between `majorRadius` and the vector from origin to the circumference. The value u = 0 corresponds to 0 radians, and u = 1 corresponds to 2 π radians. This is used to create partial ellipses. Let *pMin* be the point on the boundary of the partial ellipse that corresponds to `uMin`, and *pMax* the point corresponding to `uMax`. Then only the following part of the disk is drawn: the part bounded by the partial ellipse from *pMin* to *pMax*, the vector from the origin to *pMin*, and the vector from the origin to *pMax*. For details, see the QD3D documentation or *develop* magazine, Dec. 96. Must be 0 in Version 1.5.
`uMax`	Maximum parametric limit value; see `uMin` above. Must be 1 in Version 1.5.
`vMin`	Minimum parametric limit value in v direction. v can be viewed as the parametrization of the vector from origin to the circumference, ranging from 0 at the origin to 1 at the edge. Must be 0 in version 1.5.
`vMax`	Maximum parametric limit value in v direction; see vMin above. Must be 1 in version 1.5.

DATA SIZE

0 or 52

DESCRIPTION

A disk is a two-dimensional object defined by an origin (that is, the center of the disk) and two vectors that define the major and minor radii of the disk. A disk may have the shape of a circle, ellipse, or other oval. Attributes may be assigned to the entire disk only.

DEFAULT SURFACE PARAMETERIZATION

The default surface parameterization for a disk is as shown in Figure 22-17.

PARENT HIERARCHY

Shared, shape, geometry.

PARENT OBJECTS

None.

CHILD OBJECTS

Attribute set (optional).

EXAMPLE

```
Disk (
   1  0  0          # majorRadius
   0  1  0          # minorRadius
   0  0  0          # origin
   0                #uMin
   1                #uMax
   0                #vMin
   1                #vMax
)
```

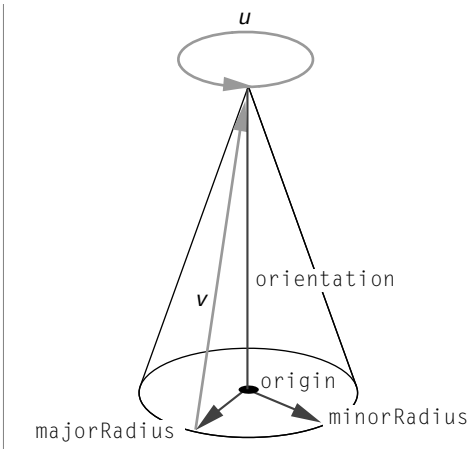
DEFAULT SIZE

For objects of size 0, the default is as in the previous example.

Cones

Figure 22-18 shows a cone.

Figure 22-18 A cone



LABELS

ASCII	`Cone`
Binary	`cone (= 0x636F6E65)`

DATA FORMAT

`Vector3D`	`orientation`
`Vector3D`	`majorRadius`
`Vector3D`	`minorRadius`
`Point3D`	`origin`
`Float32`	`uMin`

```
Float32          uMax
Float32          vMin
Float32          vMax
```

Field descriptions

`orientation` The orientation of the cone. This vector also specifies the height of the cone.

`majorRadius` The major radius of the cone.

`minorRadius` The minor radius of the cone.

`origin` The origin (that is, the center of the base) of the cone.

`uMin` Minimum parametric limit value for u. To understand u, first consider u for the ellipse determined by `majorRadius` and `minorRadius`. The value of u on this ellipse is parametrized by the angle between the major axis and the vector from the origin to the circumference of the ellipse. The value u = 0 corresponds to 0 radians, and u = 1 corresponds to 2 π radians. The values `uMin` and `uMax` are used to create partial ellipses. The basic idea is that only the part of the ellipse between `uMin` and `uMax` is drawn. For such a particular partial ellipse, the partial cone can be thought of as the result of keeping `uMin` and `uMax` fixed but letting v vary through its admissible range. The result looks like a cone with a wedge taken out. For details, see the QD3D documentation or *develop* magazine, Dec. 96. Must be 0 in Version 1.5.

`uMax` Maximum parametric limit value in u direction; see `uMin` above. Must be 1 in Version 1.5.

`vMin` Minimum parametric limit value in v direction. v can be viewed as arc length parametrization of the orientation vector, ranging from 0 at the origin to 1 at the tip. If `vMin` is not 0, then a truncated cone shaped slice will be removed from the bottom of the cylinder. Must be 0 in Version 1.5.

`vMax` Maximum parametric limit value in v direction; see `vMin` above. If `vMax` is less than 1, then a small cone will be chopped off the top of the original cone, resulting in a truncated cone. Must be 1 in Version 1.5.

DATA SIZE

0 or 64

DESCRIPTION

A cone is a three-dimensional object defined by an origin (that is, the center of the base) and three vectors that define the orientation and major and minor radii of the cone. A cap may be attached to the base of a cone. Attributes may be assigned to the cap and face of a cone, and also to the entire cone.

DEFAULT SURFACE PARAMETERIZATION

The default surface parameterization for a cone is as shown in Figure 22-18.

PARENT HIERARCHY

Shared, shape, geometry.

PARENT OBJECTS

None.

CHILD OBJECTS

Caps (optional), bottom cap attribute set (optional), face cap attribute set (optional), attribute set (optional). A cone must have a bottom cap in order to have a bottom cap attribute set. Use `Caps ( Bottom )` to set a cap on the base of a cone.

EXAMPLE

```
Container (
   Cone (
      0 1 0 # orientation
      0 0 1 # major axis
      1 0 0 # minor axis
      0 0 0 # origin
      0     # uMin
```

```
        1     # uMax
        0     # vMin
        1     # vMax
    )
    Caps ( Bottom )
    Container (
        BottomCapAttributeSet ( )
        Container (
            AttributeSet ( )
            DiffuseColor ( 1 0 0 )
        )
    )
    Container (
        FaceCapAttributeSet ( )
        Container (
            AttributeSet ( )
            DiffuseColor ( 0 0 1 )
        )
    )
)
```

Note
See the note in "Cylinders," beginning on page 1374, for an explanation of cap attribute sets. ◆

DEFAULT SIZE

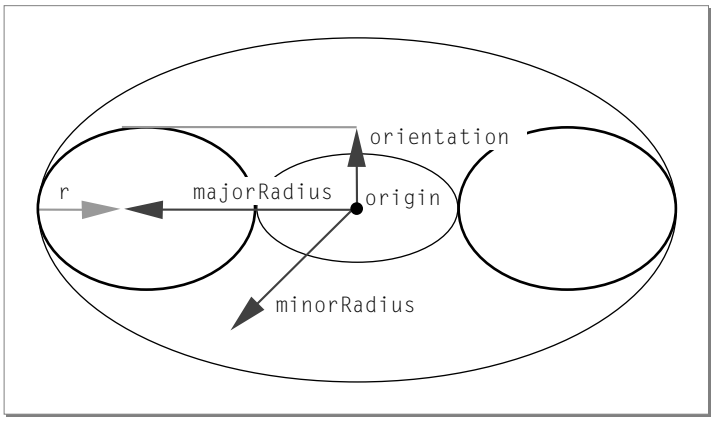
For objects of size 0, the default is:

```
1 0 0
0 1 0
0 0 1
0 0 0
0
1
0
1
```

Tori

Figure 22-19 shows a torus.

Figure 22-19 A torus



LABELS

ASCII	`Torus`
Binary	`tors (= 0x746F7273)`

DATA FORMAT

`Vector3D`	`orientation`
`Vector3D`	`majorRadius`
`Vector3D`	`minorRadius`
`Point3D`	`origin`
`Float32`	`ratio`
`Float32`	`uMin`
`Float32`	`uMax`
`Float32`	`vMin`
`Float32`	`vMax`

Field descriptions

`orientation`	The orientation of the torus. This field specifies the axis of rotation and half-thickness of the torus. The orientation must be orthogonal to both the major and minor radii.
`majorRadius`	The major radius of the torus.
`minorRadius`	The minor radius of the torus.
`origin`	The center of the torus.
`ratio`	The ratio of the length of the major radius of the rotated ellipse to the length of the orientation vector of the torus. (In Figure 22-19, this is ρ ÷ length(`orientation`.) This field indicates the eccentricity of a vertical cross-section through the torus (wide if ratio > 1, narrow if ratio < 1).
`uMin`	Minimum parametric limit value for u. To understand u, first consider u for the ellipse determined by `majorRadius` and `minorRadius`. If the torus is thought of as a doughnut, one can think of the doughnut as created by starting with this ellipse and making it thicker. The value of u on this ellipse is parametrized by the angle between the major axis and the vector from the origin to the circumference of the ellipse. The value u = 0 corresponds to 0 radians, and u = 1 corresponds to 2 π radians. The values `uMin` and `uMax` are used to create partial ellipses. The basic idea is that only the part of the ellipse between `uMin` and `uMax` is drawn. For such a partial ellipse, the partial torus can be thought of as the part of the doughnut between `uMin` and `uMax`. For details, see the QD3D documentation or *develop* magazine, Dec. 96. Must be 0 in Version 1.5.
`uMax`	Maximum parametric limit value in u direction; see `uMin` above. Must be 1 in Version 1.5.
`vMin`	Minimum parametric limit value in v direction. To understand v, we start with the ellipse described above for the `uMin` field. Pick any point p on this ellipse, and consider the following second ellipse: One axis is the orientation vector rooted at point *p*. The other axis (call it *M*) has the same direction as a vector from point *p* to the torus' origin, and its length is (ratio) times (length of orientation vector). This second ellipse describes the profile of the tube of the doughnut. The parametric value v of a point *q* on this ellipse is given by the angle between axis *M* and a vector

from point *p* to point *q*. The value v = 0 corresponds to 0 radians, and v = 1 corresponds to 2 π radians. The values `vMin` and `vMax` are used to create partial ellipses. The basic idea is that only the part of the ellipse between `vMin` and `vMax` is drawn. Assuming that `uMin` = 0 and `uMax` = 1, if `vMin` != 0 and `vMax` != 0, the resulting partial torus can be thought of as a whole doughnut with a wedge-shaped groove cut out of it. For example, if `uMin` = 0 and `uMax` = .5, the result looks like a bagel sliced in half, ready to have cream cheese spread on it. The value of `vMin` must be 0 in Version 1.5.

`vMax` Maximum parametric limit value in v direction; see `vMin` above. Must be 1 in Version 1.5.

DATA SIZE

0 or 68

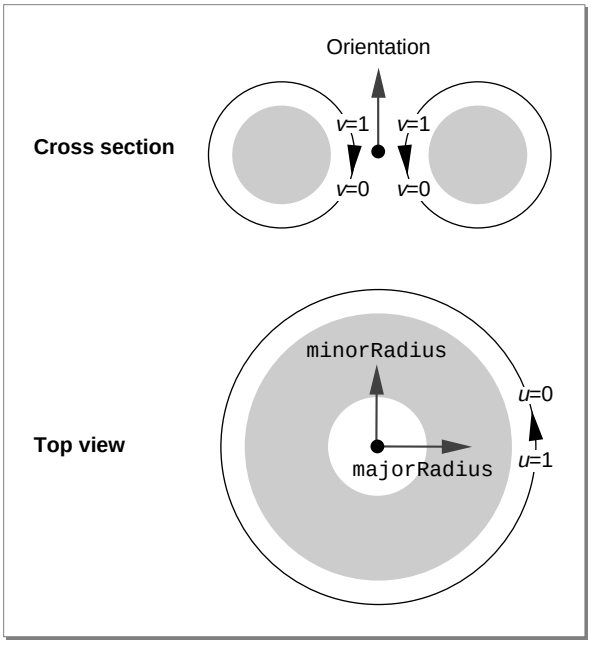
DESCRIPTION

A torus is a three-dinensional object formed by the rotation of an ellipse about an axis in the plane of the ellipse that does not cut the ellipse. The major and minor radii of the torus are the distance of the center of the ellipse from that axis.

DEFAULT SURFACE PARAMETERIZATION

The default surface parameterization for a torus is as shown in Figure 22-20.

Figure 22-20 The defalt surface parameterization of a torus



PARENT HIERARCHY

Shared, shape, geometry.

PARENT OBJECTS

None.

CHILD OBJECTS

Attribute set (optional).

EXAMPLES

```
Container (
    Torus (
        0   .2  0       #orientation
        1   0   0       #majorRadius
        0   0   1       #minorRadius
        0   0   0       #origin
        .5              #ratio
        0               #uMin
        1               #uMax
        0               #vMin
        1               #vMax
    )
    Container (
        AttributeSet ( )
        DiffuseColor (1 1 0)
    )
)
```

DEFAULT SIZE

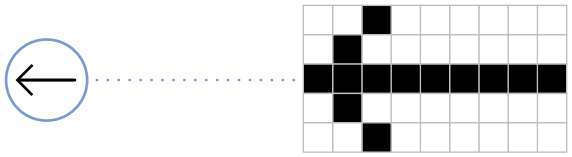
For objects of size 0, the default is:

```
1 0 0
0 1 0
0 0 1
0 0 0
1
0
1
0
1
```

Markers

Figure 22-21 shows a marker.

Figure 22-21 A marker



LABELS

ASCII	`Marker`
Binary	`mrkr` (`= 0x6D726B72`)

DATA FORMAT

`Point3D`	`location`
`Int32`	`xOffset`
`Int32`	`yOffset`
`Uns32`	`width`
`Uns32`	`height`
`Uns32`	`rowBytes`
`EndianEnum`	`bitOrder`
`RawData`	`data[height * rowBytes]`

Field descriptions

`location`	The origin of the marker.
`xOffset`	The number of pixels, in the horizontal direction, to offset the upper-left corner of the marker from the origin specified in the `location` field.
`yOffset`	The number of pixels, in the vertical direction, to offset the upper-left corner of the marker from the origin specified in the `location` field.

`width`	The width of the marker, in pixels. The value of this field must be greater than 0.
`height`	The height of the marker, in pixels. The value of this field must be greater than 0.
`rowBytes`	The number of bytes in a row of the marker.
`bitOrder`	The order in which the bits in a byte are addressed. This field must contain one of the constants `BigEndian` or `LittleEndian`.
`data[]`	This field defines a bitmap that specifies the image to be drawn.

DATA SIZE

```
36 + (rowBytes * height) + padding
```

DESCRIPTION

A marker is a two-dimensional object typically used to indicate the position of an object (or part of an object) in a window. The marker is drawn perpendicular to the viewing vector, aligned with the window, with its origin at the specified location. A marker is always drawn with the same size, shape, and orientation, no matter what transformations are active. However, a transformation may move the origin and thereby affect the position of the marker in the window. Attributes may be assigned only to the entire marker; these attributes apply to those bits in the bitmap that are set to 1.

DEFAULT SURFACE PARAMETERIZATION

None.

PARENT HIERARCHY

Shared, shape, geometry.

PARENT OBJECTS

None.

CHILD OBJECTS

Attribute set (optional).

EXAMPLE

```
Container (
       Marker (
           0.5 0.5 0.5                                  # location
           -28                                          # xOffset
           -3                                           # yOffset
           56                                           # width
           6                                            # height
           7                                            # rowBytes
           BigEndian                                    # bitOrder
           0x7E3C3C667E7C18606066666066187C3C
           0x607E7C661860066066607C1860066666
           0x6066007E3C3C667E6618
       )
       Container (
           AttributeSet ( )
           DiffuseColor ( 0.8 0.2 0.6 )
       )
   )

Marker (
   0 0 0                        # location
   -16                          # xOffset
   -16                          # yOffset
   32                           # width
   32                           # height
   4                            # rowBytes
   BigEndian                    # bitOrder

   0x001000402167E02010981810113 00C08
   0x1E60C6860D403A461880274CB0C041FC
   0x60A0811C608301193080119E30908B38
   0x18604E300CC1CA3037B23C7043181870
   0x0387E82001A01DC000502B4000502A80
```

```
    0x00506A80005DD3000076220000484C00
    0x0050180000606000004180000042000 0
    0x0042000000FF000000FF000000FF0000
)
```

DEFAULT SIZE

None.

Attributes

Diffuse Color

LABELS

ASCII	`DiffuseColor`
Binary	`kdif (= 0x6B646966)`

DATA FORMAT

`ColorRGB`	`diffuseColor`

Field descriptions

`diffuseColor`	A structure having three fields: red, green, blue. The permitted values of these fields are 32-bit floating-point numbers in the closed interval [0, 1], where 0 is the minimum value and 1 is the maximum value.

DATA SIZE

12

DESCRIPTION

Diffuse color is the color of the light of a diffuse reflection (the type of reflection that is characteristic of light reflected from a dull, non-shiny surface). A diffuse color attribute specifies the color of the light diffusely reflected by the objects to which it is assigned.

PARENT HIERARCHY

Element, attribute.

PARENT OBJECTS

Attribute sets. A diffuse color object always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
    AttributeSet ( )
    DiffuseColor ( 0 1 0 )
)
```

Specular Color

LABELS

ASCII	`SpecularColor`
Binary	`kspc` (`= 0x6b737063`)

DATA FORMAT

`ColorRGB`	`specularColor`

Field descriptions

`specularColor`	A structure having three fields: red, green, blue. The permitted values of these fields are 32-bit floating-point numbers in the closed interval [0, 1], where 0 is the minimum value and 1 is the maximum value.

DATA SIZE

12

DESCRIPTION

Specular color is the color of the light of a specular reflection (specular reflection is the type of reflection that is characteristic of light reflected from a shiny surface). A specular color attribute specifies the color of the light specularly reflected by the objects to which it is assigned. Note that the diffuse color and specular color assigned to the same object can differ.

PARENT HIERARCHY

Element, attribute.

PARENT OBJECTS

Attribute sets. A specular color object always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
    AttributeSet ( )
    DiffuseColor ( .1 .1 .1)        # near-black
    SpecularColor ( 1 1 1 )         # white
)
```

Specular Control

LABELS

ASCII	`SpecularControl`
Binary	`cspc` (`= 0x63737063`)

DATA FORMAT

`Float32` `specularControl`

Field descriptions

`specularControl` The exponent to be used in computing the intensity of the specular color of one or more objects. The value of this field must be greater than or equal to 0, and is normally an integer greater than or equal to 1.

DATA SIZE

4

DESCRIPTION

A specular control object specifies the specular reflection exponent used in the Phong and related illumination models.

PARENT HIERARCHY

Element, attribute.

PARENT OBJECTS

Attribute sets. A specular control object always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
    AttributeSet ( )
    DiffuseColor ( 1 0 0 )                # red
    SpecularColor ( 1 1 1 )               # white highlights
    SpecularControl ( 60 )                # sharp fall-off
    )
Ellipsoid( )
)
```

Transparency Color

LABELS

ASCII	`TransparencyColor`
Binary	`kxpr ( = 0x6B787072 )`

DATA FORMAT

`ColorRGB`	`transparency`

Field descriptions

`transparency`	A structure having three fields: red, green, blue. The permitted values of these fields are 32-bit floating-point numbers in the closed interval [0, 1], where 0 is the minimum value and 1 is the maximum value.

DATA SIZE

12

DESCRIPTION

A transparency color attribute affects the amount of color allowed to pass through an object that is not opaque. The transparency color values are multiplied by the color values of obscured objects during pixel color

computations. Thus, the transparency color values (1 1 1) indicate complete transparency and the values (0 0 0) indicate complete opacity. The values (0 1 0) indicate that all light in the green color channel is allowed to pass through the foreground object, and no light in the red and blue channels is allowed to pass through the foreground object.

PARENT HIERARCHY

Element, attribute.

PARENT OBJECTS

Attribute sets. A transparency color object always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
    AttributeSet ( )
    TransparencyColor ( .5 .5 .5 )
)
```

Surface UV

LABELS

ASCII	`SurfaceUV`
Binary	`sruv` (`= 0x73727576`)

DATA FORMAT

`Param2D`	`surfaceUV`

Field descriptions

`surfaceUV`	The values in the two fields of this structure specify a surface *uv* parameterization for one or more objects. Both of these values must be floating-point numbers greater than or equal to 0 and less than or equal to 1.

DATA SIZE

8

DESCRIPTION

A surface UV object is used to specify a surface *uv* parameterization for one or more objects. A surface UV object is normally used in conjunction with a trim shader.

PARENT HIERARCHY

Element, attribute.

PARENT OBJECTS

Attribute set. A surface UV object always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
    Mesh ( ... )
    Container (
        VertexAttributeSetList (
            200 Include 4 10 21 22 11
        )
```

```
        Container (
            AttributeSet ( )
            SurfaceUV ( 0 0 )
        )
        Container (
            AttributeSet ( )
            SurfaceUV ( 0 1 )
        )
        Container (
            AttributeSet ( )
            SurfaceUV ( 1 1 )
        )
        Container (
            AttributeSet ( )
            SurfaceUV ( 1 0 )
        )
    )
)
```

Shading UV

LABELS

ASCII	`ShadingUV`
Binary	`shuv` (= `0x73687576`)

DATA FORMAT

`Param2D` `shadingUV`

Field descriptions

`shadingUV` The values in the two fields of this structure specify parameters in *u* and *v* for the purpose of shading. Both of these values must be floating-point numbers greater than or equal to 0 and less than or equal to 1.

DATA SIZE

8

DESCRIPTION

A shading UV object is used to specify *uv* parameters for the purpose of shading. A shading UV object is normally used in conjunction with a texture shader.

PARENT HIERARCHY

Element, attribute.

PARENT OBJECTS

Attribute set. A shading UV object always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
    AttributeSet ( )
    ShadingUV ( 0 0 )
)
```

Surface Tangents

LABELS

ASCII	`SurfaceTangent`
Binary	`srtn(= 0x7372746E)`

DATA FORMAT

```
Vector3D        paramU
Vector3D        paramV
```

Field descriptions

`paramU` The tangent in the *u* parametric direction.

`paramV` The tangent in the *v* parametric direction.

DATA SIZE

24

DESCRIPTION

A surface tangent object is used to specify three-dimensional tangents to the surface of a geometric object. These tangents serve to indicate the direction of increasing *u* and *v* in the surface parameterization of that object.

PARENT HIERARCHY

Element, attribute.

PARENT OBJECTS

Attribute set. A surface tangent always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
    AttributeSet ( )
    SurfaceUV ( 0 0 )
    SurfaceTangent (
        1 0 0
```

```
        0 1 0
    )
)
```

Normals

LABELS

ASCII	`Normal`
Binary	`nrml (= 0x6E726D6C )`

DATA FORMAT

`Vector3D`	`normal`

Field descriptions

`normal`	The surface normal at a vertex. This vector should be normalized.

DATA SIZE

12

DESCRIPTION

The surface normal at a vertex of a verticed object is the average of the normals to the faces of that object sharing that vertex. This normal is obtained by normalizing the relevant face normal vectors, adding those vectors together, and normalizing the result. The surface normal vector is used in Gouraud shading calculations.

PARENT HIERARCHY

Element, attribute.

PARENT OBJECTS

Attribute sets. A normal always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
    AttributeSet ( )
    Normal ( -1 0 0 )
)
```

Ambient Coefficients

LABELS

ASCII	`AmbientCoefficient`
Binary	`camb (= 0x63616D62)`

DATA FORMAT

```
Float32       ambientCoefficient
```

Field descriptions

`ambientCoefficient`

The value of this field must lie in the closed interval [0, 1]. 0 is the minimum value, 1 is the maximum value.

DATA SIZE

4

DESCRIPTION

The ambient coefficient is a measure of the level of an object's reflection of ambient light. Ambient coefficients may be assigned separately and selectively to the facets and vertices of faceted and verticed objects, and the same ambient coefficient may be assigned to several objects by placing the coefficient in a suitably located attribute set.

PARENT HIERARCHY

Element, attribute.

PARENT OBJECTS

Attribute sets. An ambient coefficient always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
    AttributeSet ( )
    AmbientCoefficient ( 0.5 )
    DiffuseColor ( 1 1 1 )
)
```

Highlight State

LABELS

ASCII	`HighlightState`
Binary	`hlst` (`= 0x686C7374`)

DATA FORMAT

```
Boolean          highlighted
```

Field descriptions

`highlighted` A value of `True` indicates that affected geometric objects are to receive the highlighting effects specified by an associated highlight style object during rendering. A value of `False` indicates that the affected objects are not to receive those effects.

DATA SIZE

4

DESCRIPTION

A highlight state object is used to specify whether affected geometric objects are to receive highlighting effects during rendering. The relevant highlighting effects are specified by an associated highlight style object. If a geometric object's highlight state is set to `True` (and an associated highlight style object has been defined), then any renderer that supports highlighting will apply the attributes specified by the highlight style object to that geometric object when rendering; these attributes will override incompatible attributes assigned to that geometric object by other means. A highlight state object is idle if no associated highlight style object exists. See "Highlight Styles," beginning on page 1428, for complete details on highlight style objects.

PARENT HIERARCHY

Element, attribute.

PARENT OBJECTS

Attribute sets. A highlight state object always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
    Container (
        HighlightStyle ( )                  # highlight style object
        Container (
            AttributeSet ( )
            DiffuseColor ( 1 0 0 )          # highlighting: red color
        )
    )
    Container (
        Polygon ( ... )
        Container (
            AttributeSet ( )
            DiffuseColor ( 0 0 1 )    # polygon's normal color: blue
            HighlightState ( True )   # polygon is to be highlighted
        )                             # and will appear red when
    )                                 # rendered
```

Attribute Sets

Attribute Sets

LABELS

ASCII	`AttributeSet`
Binary	`attr` (`= 0x61747472`)

DATA FORMAT

No data.

DATA SIZE

0

DESCRIPTION

An attribute set is a collection of attributes to be applied to an object, a facet of an object, or a vertex of an object. An attribute set may include attribute objects of as many types as desired, but may include only one attribute object of any particular type. Thus, an attribute set may contain both a diffuse color attribute and a specular color attribute, but may not contain two diffuse color attributes.

Though any attribute object may be included in any attribute set, some attributes cannot sensibly be applied to objects of certain types. For example, a normal cannot sensibly be applied to an entire view, as encapsulated in a view hints object. An application should disregard such attribute specifications.

Attributes may be assigned to other objects only indirectly, through the use of attribute sets. Attributes are included in an attribute set by placing the attribute objects and the attribute set object together in a container. The attributes in that set may be assigned to a geometric object by placing the relevant container and the geometric object together in a further container.

An attribute set may also be placed in a cap attribute set of any type; in this way, attributes may be assigned separately and selectively to the caps and face of a cone or cylinder. Attribute sets may also be placed in face, geometry, and vertex attribute set lists; in this way, attributes may be assigned separately and selectively to the facets, segments, and vertices of geometric objects having those features.

An attribute set may also be placed in a group. Unless overridden, the attributes in an attribute set placed in a hierarchically structured group are inherited by objects at lower levels in the hierarchy of that group. (An application should not permit an attribute to be inherited by an object to which that attribute cannot sensibly be applied.) See the sections on cap attribute sets, attribute set lists, and groups for complete details on the composition of these objects.

PARENT HIERARCHY

Shared, set.

PARENT OBJECTS

Any geometric object, cap attribute set, attribute set list, or group. An attribute set always has a parent object.

CHILD OBJECTS

Attributes: ambient coefficient, diffuse color, specular color, specular control, transparency color, highlight state, shading UV, surface UV (all optional).

EXAMPLE

```
Container (
    Polygon (...)        # all attributes in set applied to polygon
    Container (          # container puts attributes in set
        AttributeSet ( )
        AmbientCoefficient (...)
        DiffuseColor (...)
        SpecularColor (...)
        SpecularControl (...)
        Normal (...)
    )
)
```

Top Cap Attribute Sets

LABELS

ASCII	`TopCapAttributeSet`
Binary	`tcas (= 0x74636173)`

DATA FORMAT

No data.

DATA SIZE

0

DESCRIPTION

A top cap attribute set is used to attach attributes to the top cap of a cylinder that has an optional top cap. The attributes to be assigned to the cap are placed in a regular attribute set in the usual manner. Then the container holding the regular attribute set and the attributes is placed in the cap attribute set by including that container and the cap attribute set in a further container.

The attributes associated with a top cap attribute set are not drawn if the parent object lacks a top cap.

PARENT HIERARCHY

Data, cap data.

PARENT OBJECTS

Cylinder (always).

CHILD OBJECTS

Attribute set (optional). An empty top cap attribute set has no effect.

EXAMPLE

```
Container (
    Cylindner ( ... )
    Caps ( Top )
    Container (
        TopCapAttributeSet ( )
        Container (
            AttributeSet ( )
            DiffuseColor ( 0.2 0.9 0.4 )
        )
    )
)
```

Bottom Cap Attribute Sets

LABELS

ASCII	`BottomCapAttributeSet`
Binary	`bcas ( = 0x62636173 )`

DATA FORMAT

No data.

DATA SIZE

0

DESCRIPTION

A bottom cap attribute set is used to attach attributes to the bottom cap of a cone or cylinder that has an optional bottom cap. The attributes to be assigned to the cap are placed in a regular attribute set in the usual manner. Then the container holding the regular attribute set and the attributes is placed in the cap attribute set by including that container and the cap attribute set in a further container.

The attributes associated with a bottom cap attribute set are not drawn if the parent object lacks a bottom cap.

PARENT HIERARCHY

Data, cap data.

PARENT OBJECTS

Cone, cylinder (always).

CHILD OBJECTS

Attribute set (optional). An empty bottom cap attribute set has no effect.

EXAMPLE

```
Container (
    Cylinder ( )
    Caps ( Bottom )
    Container (
        BottomCapAttributeSet ( )
        Container (
            AttributeSet ( )
            DiffuseColor ( 0.2 0.9 0.4 )
        )
    )
)
```

Face Cap Attribute Sets

LABELS

ASCII	`FaceCapAttributeSet`
Binary	`fcas (= 0x66636173)`

DATA FORMAT

No data.

DATA SIZE

0

DESCRIPTION

A face cap attribute set is used to attach an attribute set to the surface of a cone or cylinder but not to its caps. This object is used to apply attributes to a cone or cylinder in a way that does not cause them to be inherited by its caps.

PARENT HIERARCHY

Data, cap data.

PARENT OBJECTS

Cone, cylinder (always).

CHILD OBJECTS

Attribute set (optional). An empty face cap attribute set has no effect.

EXAMPLE

```
Container (
    Cylinder ( )
    Caps ( Top )
    Container (
        AttributeSet ( )
        SurfaceShader (...)
    )
    Container (
        FaceCapAttributeSet ( )
        Container (
            AttributeSet ( )
            DiffuseColor ( 1 0 0 )
        )
    )
)
```

Attribute Set Lists

Geometry Attribute Set Lists

LABELS

ASCII	`GeometryAttributeSetList`
Binary	`gasl ( = 0x6761736C )`

DATA FORMAT

`Uns32`	`nObjects`
`PackingEnum`	`packing`
`Uns32`	`nIndices`
`Uns`	`indices[nIndices]`

Field descriptions

`nObjects`	The total number of instances of the relevant feature of the parent geometric object possessed by that object. If the parent object is a polyline, the relevant feature is polyline segment, so the value of this field is the total number of segments of the polyline.
`packing`	See “Face Attribute Set Lists,” beginning on page 1416, for a complete explanation of this field.
`nIndices`	The size of the following array. See “Face Attribute Set Lists,” beginning on page 1416, for a complete explanation of this field.
`indices[]`	An array of indices. A standard method of indexing instances of the relevant feature of the parent object is assumed to have been established, as with the segments of a polyline. The values of this field are the indices of such instances and are to be specified in increasing order. See “Face Attribute Set Lists,” beginning on page 1416, for a complete explanation of this field.

DATA SIZE

16 + `nIndices` * `sizeof(Uns)` + padding

DESCRIPTION

A geometry attribute set list is used to assign sets of attributes separately and selectively to distinct instances of a tractable feature of geometric objects. A standard method of indexing the instances of such a feature is presupposed by a geometry attribute set list.

At present, the polyline is the only primitive geometric object to which a geometry attribute set list may be attached. The attribute sets appearing in a geometry attribute set list are assigned to the line segments of which the polyline is composed, not to the vertices of the polyline. (To attach attributes to the vertices, use a vertex attribute set list.)

The standard index of the segments of a polyline is described in "Polylines," beginning on page 1311. To recapitulate, the segment having index *i* is the segment having as its endpoints `vertices[i]` and `vertices[i+1]`.

PARENT HIERARCHY

Data, attribute set list.

PARENT OBJECTS

Polyline (always).

CHILD OBJECTS

Attribute sets (required). See "Face Attribute Set Lists," beginning on page 1416, for a complete explanation of how child objects are correlated with instances of the relevant features of the parent geometric object.

EXAMPLE

```
Container (
    PolyLine (...)                          #parent geometric object

    Container (
        GeometryAttributeSetList ( )
        6 exclude 4       # there are 6 segments; exclude 4 of them
        0   2   3   5     # indices of the segments to be excluded
        #child objects
        Container (
            AttributeSet          ‘         #applied to segment 1
            DiffuseColor (...)
        )
        Container (
            AttributeSet                    #applied to segment 4
            DiffuseColor (...)
        )
    )
)
```

Face Attribute Set Lists

LABELS

ASCII	`FaceAttributeSetList`
Binary	`fasl (= 0x6661736C)`

PACKING ENUM DATA TYPE

`PackingEnum`

The permitted values are `include ( = 0x00000000)` and `exclude ( = 0x00000001)`.

DATA FORMAT

```
Uns32          nObjects
PackingEnum    packing
Uns32          nIndices
Uns32          indices[nIndices]
```

Field descriptions

`nObjects` The total number of faces or facets of the parent object. If the parent object is a box, the value of this field is 6. If the parent object is a trigrid, the value of this field is the number of vertices used to define that trigrid, which is also the number of facets of the trigrid. If the parent object is a mesh, the value of this field is the number of faces of that mesh.

`packing` The value of this field determines whether the facets of the parent object of the set list to receive attributes are those whose facet indices appear in the array `indices[]` or are those whose indices do not appear in that array. A value of `include` indicates the former; `exclude` indicates the latter. You may wish to select `include` if most facets of the parent object are not to receive any attributes. Should any other value appear in this field, the entire set list and all of its child objects should be ignored.

`nIndices` The number of facets of the parent object to which the action specified in the packing field is to be applied; that is, the number of facets to be included in (or excluded from) the group of facets to receive attributes. The value of this field may not exceed that of the `nObjects` field.

`indices[]` An array of facet indices. The values in the fields of this array are the indices of those facets of the parent object to be subject to the action of the value of the packing field, in the event that the number of facets to receive attributes is less than the value in the `nObjects` field. The size of this array must equal the value in the `nIndices` field. Indices are to be entered in fields of this array in increasing order; no index may appear more than once. If the value in the `packing` field is `include`, then the field values represent those facets which are to receive attributes in consequence of the set list. If the value in the `packing` field is `exclude`,

then the field values represent those facets that are not to receive attributes in consequence of the set list. If the value in the `packing` field is `exclude` and the value in the `nIndices` field is 0, then this field may be left unspecified; similarly, if the value in the `packing` field is `include` and the value in the `nIndices` field is equal to the value in the `nObjects` field, then this field may be left unspecified.

DATA SIZE

```
16 + nIndices * sizeof(Uns) + padding
```

DESCRIPTION

A face attribute set list is used to assign sets of attributes separately and selectively to one or more facets of a multi-faceted geometric object (that is, to the faces of a box or mesh, or to the triangular facets of a trigrid). A face attribute set list may not be assigned to a general polygon. The listed attribute sets themselves occur as child objects of the set list object and are correlated with facets of the parent object of the set list as described later in this section. You may think of the child objects as the items in the set list; officially, the set list is the object defined in this section.

For convenience, the `packing` field allows you to choose whether to specify (by inclusion) the facets to receive attributes or to specify (by exclusion) the facets not to receive attributes. The number of child objects you must specify is equal to the number of facets actually to receive attributes, whichever option you select for the `packing` field. You may wish to specify by inclusion rather than by exclusion if most facets are not to receive any attributes. This option can reduce the size of the `indices[]` array, save work, and save disk space.

If the value of the `packing` field of a set list is `include`, then the number of child objects of that set list must equal the value of the `nIndices` field of that set list. If the value of the `packing` field is `exclude`, then the number of child objects must equal the (absolute value of) the difference between the values of the `nObjects` and `nIndices` fields.

Child objects are correlated with facets of the parent object of the set list as follows. Let the child objects of the set list be enumerated in the order of their occurrence in the metafile. If the value of the `packing` field is `include`, then the *i*th child object is correlated with the facet whose index is the value of the *i*th field of the array `indices[]`, or `indices[i-1]`. If the value of the `packing` field is

`exclude`, then the *i*th child object is correlated with the facet whose facet index is the *i*th element of the sequence (in increasing order) of facets whose indices do *not* appear in the array `indices[]`. For example, suppose that the parent object is a mesh having 17 faces, `packing` is set to `exclude`, `nIndices` is 11, and the elements of `indices[]` are 1, 2, 4, 6, 7, 8, 11, 12, 13, 14, 16. Then six facets are to receive attributes: facets 0, 3, 5, 9, 10, 15, so the set list will have six child objects c_0,..., c_5. The third child object (that is, c_2) is correlated with facet 5, and, in general, the *i*th element of the sequence <c_0,..., c_5> is correlated with the *i*th element of the sequence <0, 3, 5, 9, 10, 15>.

The index used to enumerate the facets of a multifaceted geometric object is described in the section pertaining to that object. Indices begin with zero, so that the index of the *i*+1st facet of a multifaceted object is *i*. The index used to construct an attribute set list must be standard.

PARENT HIERARCHY

Data, attribute set list.

PARENT OBJECTS

Box, mesh, trigrid.

CHILD OBJECTS

Attribute sets (required). The number of child objects is determined in the manner indicated in the description of a face attribute set list.

EXAMPLE

```
Container (
    TriGrid (...)                           #parent object
    Container (
        FaceAttributeSetList ( )
        6                       #nObjects          (parent has six facets;
        exclude                 #packing           (exclude
        4                       #nIndices          (four of them:
        0   2   3   5           #indices[]         (these four.)
```

```
        #begin list
        Container (
            AttributeSet                            #apply to facet 1
            DiffuseColor (...)
        )
        Container (
            AttributeSet                            #apply to facet 4
            DiffuseColor (...)
        #end list
        )
    )
)
```

Vertex Attribute Set Lists

LABELS

ASCII	`VertexAttributeSetList`
Binary	`vasl (= 0x7661736C)`

DATA FORMAT

`Uns32`	`nObjects`
`PackingEnum`	`packing`
`Uns32`	`nIndices`
`Uns`	`indices[nIndices]`

Field descriptions

`nObjects`	The number of vertices of the parent geometric object.
`packing`	See "Face Attribute Set Lists," beginning on page 1416, for a complete explanation of this field.
`nIndices`	Size of the following array. See "Face Attribute Set Lists," beginning on page 1416, for a complete explanation of this field.

`indices[]` An array of vertex indices. See "Face Attribute Set Lists," beginning on page 1416, for a complete explanation of this field.

DATA SIZE

```
16 + nIndices * sizeof(Uns) + padding
```

DESCRIPTION

A vertex attribute set list is used to assign sets of attributes separately and selectively to the vertices of a verticed geometric object. Among the primitive metafile geometric objects, the following have vertices: general polygons, lines, meshes, polygons, polylines, triangles, and trigrids.

The index used to enumerate the vertices of an object of one of these types is described in the section on objects of that type. To recapitulate, in all cases the vertices are enumerated in the order of their occurrence in the specification of the parent geometric object. In the case of a general polygon, the index does not distinguish between contours.

PARENT HIERARCHY

Data, attribute set list.

PARENT OBJECTS

General polygon, line, mesh, polygon, polyline, triangle, trigrid. A vertex attribute set list always has a parent object.

CHILD OBJECTS

Attribute sets (required). See "Face Attribute Set Lists," beginning on page 1416, for a complete explanation of how child objects are correlated with aspects of the parent geometric object.

EXAMPLE

```
Container (
    GeneralPolygon (            # parent geometric object
        2                       # nContours
        #contour 0
        3                       # nVertices, contour 0
        -1 0 0                  # vertex 0
        1 0 0                   # vertex 1
        0 1.7 0                 # vertex 2
        #contour 1
        3                       # nVertices, contour 1
        -1 0.4 0                # vertex 3
        1 0.4 0                 # vertex 4
        0 2.1 0                 # vertex 5
    )
    Container (
        VertexAttributeSetList ( 6 Exclude 2 0 4 )     # set list
        Container (                                    # child objects
            AttributeSet ( )               # vertex 1 (contour 0)
            DiffuseColor ( 0 0 1 )
        )
        Container (
            AttributeSet ( )               # vertex 2 (contour 0)
            DiffuseColor ( 0 1 1 )
        )
        Container (
            AttributeSet ( )               # vertex 3 (contour 1)
            DiffuseColor ( 1 0 1 )
        )
        Container (
            AttributeSet ( )               # vertex 5 (contour 1)
            DiffuseColor ( 1 1 0 )
        )
    )
    Container (
        AttributeSet ( )
        DiffuseColor ( 1 1 1 )
    )
)
```

Styles

Back-facing Styles

LABELS

ASCII	`BackfacingStyle`
Binary	`bckf` (`= 0x62636B66`)

BACK-FACING STYLES

`Both`	`0x00000000`
`Culled`	`0x00000001`
`Flipped`	`0x00000002`

Constant descriptions

`Both`	A renderer should draw shapes that face toward and away from the camera. If a shape has only front-facing attributes, those attributes are used for both sides of the shape.
`Culled`	A renderer should not draw shapes that face away from the camera (this is not the same as hidden surface removal).
`Flipped`	A renderer should draw shapes that face toward and away from the camera. If a shape has only front-facing attributes, those attributes are used for both sides of the shape, but the normals of back-facing shapes are inverted, so that they face toward the camera.

DATA FORMAT

`BackfacingEnum`	`backfacing`
`backfacing`	The value in this field must be one of the three constants defined above.

DESCRIPTION

A scene's back-facing style determines whether or not a renderer draws shapes that face away from a scene's camera. This style object defines some of the characteristics of a renderer and generally applies to all of the objects in a model.

PARENT HIERARCHY

Shared, shape, style.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
BeginGroup ( OrderedDisplayGroup ( ) )
   Matrix ( ... )
   BackfacingStyle ( Both )
   Mesh ( ... )
   Mesh ( ... )
EndGroup ( )
```

Interpolation Styles

LABELS

ASCII	`InterpolationStyle`
Binary	`intp (= 0x696E7470)`

INTERPOLATION STYLES

`None`	`0x00000000`
`Vertex`	`0x00000001`
`Pixel`	`0x00000002`

Constant descriptions

`None`	No interpolation is to occur. The renderer is to apply each effect uniformly across a surface.
`Vertex`	The renderer is to interpolate values linearly across a verticed surface, using the values at the vertices.
`Pixel`	The renderer is to calculate a value of each effect for every pixel in the image.

DATA FORMAT

```
InterpolationStyleEnum          interpolationStyle
```

Field descriptions

`interpolationStyle`
The value in this field must be one of these constants: `None`, `Vertex`, or `Pixel`.

DATA SIZE

4

DESCRIPTION

A scene's interpolation style determines the method of interpolation a renderer uses when applying lighting or other shading effects to a surface. A value of `None` causes the surfaces of a model to have a faceted appearance; the other two values cause its surfaces to be rendered smoothly.

PARENT HIERARCHY

Shared, shape, style.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
BeginGroup ( DisplayGroup ( ) )
    InterpolationStyle ( Vertex )
    Container (
        Triangle ( ... )
        VertexAttributeSetList ( ... )
        .
        .
        .
    )
EndGroup ( )
```

Fill Styles

LABELS

ASCII	`FillStyle`
Binary	`fist (= 0x66697374)`

FILL STYLES

`Filled`	`0x00000000`
`Edges`	`0x00000001`
`Points`	`0x00000002`

Constant descriptions

`Filled`	The renderer should draw shapes as solid filled objects.

`Edges`	The renderer should draw shapes as the sets of lines that define the edges of surfaces.
`Points`	The renderer should draw shapes as the sets of points that define the vertices of surfaces.

DATA FORMAT

`FillStyleEnum` `fillStyle`

Field descriptions

`fillStyle`	The value of this field must be one of these constants: `Filled`, `Edges`, `Points`.

DATA SIZE

4

DESCRIPTION

A scene's fill style determines whether an object is drawn as a solid filled object or is decomposed into a set of edges or points.

PARENT HIERARCHY

Shared, shape, style.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
BeginGroup ( DisplayGroup ( ) )
    FillStyle ( Edges )
    Container (
        Mesh ( ... )
        VertexAttributeSetList ( ... )
    )
    Torus ( ... )
EndGroup( )
```

Highlight Styles

LABELS

ASCII	`HighlightStyle`
Binary	`high` (`= 0x68696768`)

DATA FORMAT

No data.

DATA SIZE

0

DESCRIPTION

A highlight style object is used to specify attributes to be applied to selected geometric objects during rendering. Any renderer that supports highlighting will use the attributes specified by a highlight style object to override incompatible attributes assigned to affected geometric objects in other ways. The attributes specified by a highlight style object are applied to a geometric object only if that geometric object also has a highlight state attribute that is set to `True`. See “Highlight State,” beginning on page 1405, for complete details on highlight state attributes.

PARENT HIERARCHY

Shared, shape, style.

PARENT OBJECTS

None.

CHILD OBJECTS

Attribute set (required).

EXAMPLE

```
BeginGroup ( DisplayGroup ( ) )
    Container (
        HighlightStyle ( )                  # highlight style object
        Container (
            AttributeSet ( )
            DiffuseColor ( 0 0 1 )          # highlight attribute
        )
    Container (
        Polygon ( ... )
        Container (
            AttributeSet ( )
            DiffuseColor ( 1 0 0 )
            HighlightState ( True )  # polygon will be highlighted
        )
    )
    Container (
        Box
        Container (
            AttributeSet ( )
            DiffuseColor ( 0 1 0 )
            HighlightState ( False )# box will not be highlighted
        )
    )
```

```
    Container (
        Line ( ... )                # line will not be highlighted
        Container (
            AttributeSet ( )
            DiffuseColor ( 1 1 1 )
        )
    )
)
EndGroup ( )
```

Subdivision Styles

LABELS

ASCII	`SubdivisionStyle`
Binary	`sbdv (= 0x7364636C )`

SUBDIVISION METHOD ENUM DATA TYPE

`Constant`	`0x00000000`
`WorldSpace`	`0x00000001`
`ScreenSpace`	`0x00000002`

Note
There are two data formats. ◆

FIRST DATA FORMAT

`SubdivisionMethodEnum`	`subdivisionMethod`
`Float32`	`value1`

Field descriptions

`subdivisionMethod`
The value in this field must be one of the specifiers `WorldSpace` or `ScreenSpace`. A value of `WorldSpace` indicates that the renderer subdivides a curve (or surface)

into polylines (or polygons) whose sides have a world-space length that is at most as large as the value specified in the `value1` field. A value of `ScreenSpace` indicates that the renderer subdivides a curve (or surface) into polylines (or polygons) whose sides have a length that is at most as large as the number of pixels specified in the `value1` field.

`value1` For world-space subdivision, the maximum length of a polyline segment (or polygon side) into which a curve (or surface) is subdivided. For screen-space subdivision, the maximum number of pixels in a polyline segment (or polygon side) into which a curve (or surface) is subdivided. The value in this field should be greater than 0.

DATA SIZE

8

SECOND DATA FORMAT

`SubdivisionMethodEnum`	`subdivisionMethod`
`Uns32`	`value1`
`Uns32`	`value2`

Field descriptions

`subdivisionMethod`

The value in this field must be the specifier `Constant`. This value indicates that the renderer subdivides a curve into a number of polyline segments and a surface into a mesh of polygons.

`value1` The number of polylines into which a curve should be subdivided, or the number of vertices in the *u* parametric direction of the polygonal mesh into which a surface is divided. The value in this field should be greater than 0.

`value2` The number of vertices in the *v* parametric direction of the polygonal mesh into which a surface is divided. The value in this field should be greater than 0.

DATA SIZE

12

DESCRIPTION

A scene's subdivision style determines how a renderer decomposes smooth curves and surfaces into polylines and polygonal meshes for display purposes. Different specifiers and numerical values determine different degrees of fineness of approximation.

PARENT HIERARCHY

Shared, shape, style.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
BeginGroup ( DisplayGroup ( )
    SubdivisionStyle ( Constant 32 32 )
    Ellipsoid ( ... )
    )
    Container (
        SubdivisionStyle ( WorldSpace 12 )
        Box ( ... )
    )
EndGroup ( )
```

Orientation Styles

LABELS

ASCII	`OrientationStyle`
Binary	`ornt (= 0x6F726E74)`

ORIENTATION STYLES

`CounterClockwise`	`0x00000000`
`Clockwise`	`0x00000001`

Constant descriptions

`CounterClockwise`
: The front face of a polygonal shape is defined using the righthand rule.

`Clockwise`
: The front face of a polygonal shape is defined using the lefthand rule.

DATA FORMAT

`OrientationEnum`	`orientation`

Field descriptions

`orientation`
: The value in this field must be one of these constants: `CounterClockwise`, `Clockwise`.

DATA SIZE

4

DESCRIPTION

A scene's orientation style determines which side of a planar surface is considered (by the renderer) to be the "front" side. This style may be changed in order to change the orientation of a polygonal shape.

PARENT HIERARCHY

Shared, shape, style.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
BeginGroup( DisplayGroup ( ) )
    OrientationStyle ( Clockwise )
    .
    .
    .
EndGroup ( )
```

Receive Shadows Styles

LABELS

ASCII	`ReceiveShadowsStyle`
Binary	`rcsh` (`= 0x72637368`)

DATA FORMAT

`Boolean`	`receiveShadows`

Field descriptions

`receiveShadows`	A value of `True` indicates that objects are to receive shadows; a value of `False` indicates that objects are not to receive shadows.

DATA SIZE

4

DESCRIPTION

A scene's receive shadows style specifies whether or not obscured objects shall receive shadows in rendering.

Note
Some lights also specify whether or not the objects they illuminate shall cast shadows. However, objects in the scope of a receive shadows style set to `False` do not receive shadows, regardless of whether they are also appropriately situated to receive shadows from a light set to cast shadows. ◆

PARENT HIERARCHY

Shared, shape, style.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
BeginGroup ( DisplayGroup ( ) )
    PointLight ( ... )
    ReceiveShadows ( True )
    Mesh ( ... )
    Mesh ( ... )
    Mesh ( ... )
EndGroup ( )
```

Pick ID Styles

LABELS

ASCII	`PickIDStyle`
Binary	`pkid (= 0x706B6964)`

DATA FORMAT

```
Uns32          id
```

Field descriptions

`id`	An integer, supplied by your application.

DATA SIZE

4

DESCRIPTION

A pick ID style object is used to correlate the class of objects within its scope with an integer. This integer may be included in the specification of a picking operation to restrict that operation to the objects in that class. A pick ID style object must be placed in a group or container to have effect; the scope of a pick ID style object placed in a group (or container) is the class of objects located between that style object and the end of that group (or container).

PARENT HIERARCHY

Shared, shape, style.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
PickIDStyle ( 8 )
```

Pick Parts Styles

LABELS

ASCII	`PickPartsStyle`
Binary	`pkpt (= 0x706B7074)`

PICK PARTS STYLES

`Object`	`0x00000000`
`Face`	`0x00000001`
`Edge`	`0x00000002`
`Vertex`	`0x00000003`

Constant descriptions

`Object`	The hit list for picking contains only whole objects.
`Face`	The hit list for picking contains faces of objects.
`Edge`	The hit list for picking contains edges of objects.
`Vertex`	The hit list for picking contains vertices of objects.

DATA FORMAT

`PickPartsFlags`	`pickParts`

Field descriptions

`pickParts`	The value in this field must be one of the four flags specified in the `PickPartsFlags` data enumeration.

DATA SIZE

4

DESCRIPTION

A pick parts style object is used to specify the sort of object to be picked during the operation of picking. The flags `Face`, `Edge`, and `Vertex` are used to pick meshes; the flag `Object` is used to pick all other objects. The default flag is `Object`.

PARENT HIERARCHY

Shared, shape, style.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
PickPartsStyle ( Object )
```

Transforms

Translate Transforms

LABELS

ASCII	`Translate`
Binary	`trns (= 0x74726E73)`

DATA FORMAT

`Vector3D`	`translate`

Field descriptions

`translate`	A translation in three dimensions, specified by a vector.

DATA SIZE

12

DESCRIPTION

A translate transform moves an object along the *x*, *y*, and *z* axes by the values specified by its translation vector. Thus, the transform `Translate ( i j k )` moves each point P = <P_x, P_y, P_z> in its scope to the point P′ = <P_{x+i}, P_{y+j}, P_{z+k}>.

PARENT HIERARCHY

Shared, shape, transform.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
Translate ( -1 1 0 )
```

Scale Transforms

LABELS

ASCII	`Scale`
Binary	`scal( = 0x7363616C )`

DATA FORMAT

`Vector3D`	`scale`

Field descriptions

`scale`	A scaling vector.

DATA SIZE

12

DESCRIPTION

A scale transform scales an object along the *x*, *y*, and *z* axes by the values specified by its scaling vector.

PARENT HIERARCHY

Shared, shape, transform.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
Scale ( 2 2 2 )
```

Matrix Transforms

LABELS

ASCII	`Matrix`
Binary	`mtrx ( = 0x6D747278 )`

DATA FORMAT

`Matrix4x4` `matrix`

Field descriptions

`matrix` A 4-by-4 array specifying a custom matrix transformation. The specified matrix should be invertible.

DATA SIZE

64

DESCRIPTION

A matrix transform is a transform by an arbitrary invertible 4-by-4 matrix.

PARENT HIERARCHY

Shared, shape, transform.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
Matrix (
   1 0 0 0
   0 1 0 0
   0 0 1 0
   0 0 0 1
)
```

Rotate Transforms

LABELS

ASCII	`Rotate`
Binary	`rott (= 0x726F7474)`

AXIS ENUM DATA TYPE

`X`	`0x00000000`
`Y`	`0x00000001`
`Z`	`0x00000002`

DATA FORMAT

`AxisEnum`	`axis`
`Float32`	`radians`

Field descriptions

`axis`	The axis of rotation. The value in this field must be one of these constants: `X`, `Y`, or `Z`.
`radians`	The number of radians to rotate around the axis of rotation.

DATA SIZE

8

DESCRIPTION

A rotate transform rotates an object about the *x*, *y*, or *z* axis by a specified number of radians.

PARENT HIERARCHY

Shared, shape, transform.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
Rotate (              # rotate about the z axis by -1.57 radians
   Z
   -1.57
   )
```

Rotate-About-Point Transforms

LABELS

ASCII	`RotateAboutPoint`
Binary	`rtap ( = 0x72746170 )`

DATA FORMAT

`AxisEnum`	`axis`
`Float32`	`radians`
`Point3D`	`origin`

Field descriptions

`axis`	The axis of rotation.
`radians`	The number of radians to rotate about the axis of rotation.
`origin`	The point at which the rotation is to occur.

DATA SIZE

20

DESCRIPTION

A rotate-about-point transform rotates an affected object by the specified number of radians about the line parallel to the value in the `axis` field and passing through the point specified in the `origin` field.

PARENT HIERARCHY

Shared, shape, transform.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
Rotate (
   Y                    # axis
   1.0                  # radians
   2 3 4                # origin
)
```

Rotate-About-Axis Transforms

LABELS

ASCII	`RotateAboutAxis`
Binary	`rtaa ( = 0x72746161 )`

DATA FORMAT

```
Point3D          origin
Vector3D         orientation
Float32          radians
```

Field descriptions

`origin`	The origin of the axis of rotation.
`orientation`	The orientation of the axis of rotation. This vector should be normalized.
`radians`	The number of radians by which an affected object is rotated.

DATA SIZE

28

DESCRIPTION

A rotate-about-axis transform rotates an object about an arbitrary axis in space by a specified number of radians. The value in the `origin` field and the orientation vector are used to define the axis of rotation.

PARENT HIERARCHY

Shared, shape, transform.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
RotateAboutAxis (
    20 0 0              # origin
    .33 .33 .34         # orientation
    1.57                # radians
)
```

Quaternion Transforms

LABELS

ASCII	`Quaternion`
Binary	`qtrn ( = 0x7174726E )`

DATA FORMAT

`Float32`	`w`
`Float32`	`x`
`Float32`	`y`
`Float32`	`z`

Field descriptions

`w`	The *w* component of the quaternion transform.
`x`	The *x* component of the quaternion transform.
`y`	The *y* component of the quaternion transform.
`z`	The *z* component of the quaternion transform.

DATA SIZE

16

DESCRIPTION

A quaternion transform rotates and twists an object in a manner determined by the mathematical properties of quaternions.

PARENT HIERARCHY

Shared, shape, transform.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
Quaternion ( 0.2 0.7 0.2 1.57 )
```

Shader Transforms

LABELS

ASCII	`ShaderTransform`
Binary	`sdxf ( = 0x73647866)`

DATA FORMAT

`Matrix4x4`	`shaderTransform`

Field descriptions

`shaderTransform`	A 4-by-4 matrix.

DATA SIZE

64

DESCRIPTION

A shader transform transforms a shaded object into a distinct world-space coordinate system. A shader transform does not affect the current transformation hierarchy and does not affect the manner in which the object to which it is applied is drawn.

PARENT HIERARCHY

Data.

PARENT OBJECTS

Any shader. A shader transform always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
    CustomShader ( )
    ShaderTransform (
        1 0 0 0
        0 1 0 0
        0 0 1 0
        2 3 4 1
    )
)
```

Shader UV Transforms

LABELS

ASCII	`ShaderUVTransform`
Binary	`sduv ( = 0x73647576 )`

DATA FORMAT

`Matrix3x3` `matrix`

Field descriptions

`matrix` A 3-by-3 matrix.

DATA SIZE

36

DESCRIPTION

A shader *uv* transform may be used to transform the surface *uv* parameterization of a geometric object prior to shading. A shader *uv* transform may be used to rotate a texture map.

PARENT HIERARCHY

Data.

PARENT OBJECTS

Any shader. A shader *uv* transform always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
    TextureShader ( )
    ShaderUVTransform (
        1 0 0
        0 1 0
        0.2 0.3 1
    )
    PixmapTexture ( ... )
)
```

Lights

Attenuation and Fall-Off Values

Some lights suffer attenuation; that is, a loss of intensity over distance. The application determines the degree of attenuation of a light by specifying substituends for three distinct variables in a complex term that occurs in whatever formula it uses to compute the intensity of that light at a given distance from its source. The choice of constants determines whether the light suffers attenuation and, if so, the degree to which its intensity diminishes as a function of distance. These constants are specified in a data structure of the type described here.

ATTENUATION DATATYPE

```
Float32          c0
Float32          c1
Float32          c2
```

DESCRIPTION

The attenuation factor determined by an attenuation data type is expressed by the result of replacing the variables c_0, c_1, c_2 by the values of the fields `c0`, `c1`, `c2` in the complex term

$$\left(\frac{1}{c_0 + c_1 d_{l,\,p} + c_2 d_{l,\,p}{}^2} \right)$$

(Here l is the location of the light source, p is the illuminated point, and $d_{l,\,p}$ is the distance from l to p.)

The initial intensity of a light is multiplied by its attenuation factor when the intensity of the light at a point is computed. Thus, if $c_0 = 1$ and $c_1 = c_2 = 0$, then the light does not suffer attenuation over distance. If $c_1 = 1$ and $c_0 = c_2 = 0$, then the intensity of the light at a point p diminishes in proportion to the distance between p and the light source, provided that that distance is at least one unit. If

$c_2 = 1$ and $c_0 = c_1 = 0$, then the intensity of the light at p diminishes in proportion to the square of the distance between p and the light source, again provided that that distance is at least one unit. If $c_0 = c_2 = 1$ and $c_1 = 0$, then the intensity of the light at p diminishes in proportion to the sum of 1 and the square of the distance between p and the light source.

The attenuation factor is not clamped to a maximum value. Thus, for some choices of c_0, c_1, c_2, the intensity of a light may exceed its source intensity at distances of less than one unit, driving the RGB color values of the light toward the maximum of (1, 1, 1), or pure white.

The amount of illumination that a point illuminated by a light receives from that light also depends on several other factors. Among these factors are the diffuse and specular reflection characteristics of the surface that contains that point and the relative positions of the light source, the illuminated point, and the viewer (the camera).

LIGHT FALL-OFF VALUES

A spot light specifies a cone of light emanating from a source location. Within the inner cone defined by the hot angle of a spot light, the light may suffer attenuation over distance from the light source. Within the outer section of the cone between the hot angle and the outer angle of a spot light, the light may suffer further attenuation.

Spot lights have a fall-off value that determines the manner of attenuation of the light from the edge of the cone defined by the hot angle to the edge of the cone defined by the outer angle. The direction of fall off is perpendicular to the ray from the source location through the center of the cone. The amount of additional attenuation determined by any fall-off value is the same along all rays from the location of the light source forming the same angle with the axis of the cone.

The following constants specify four fall-off values a spot light may have.

FALLOFF VALUES

`None`	`0x00000000`
`Linear`	`0x00000001`
`Exponential`	`0x00000002`
`Cosine`	`0x00000003`

Constant descriptions

`None`	The intensity of the light is not affected by the distance from the center of the cone to the edge of the cone.
`Linear`	The intensity of the light at the edge of the cone falls off at a constant rate from the intensity of the light at the center of the cone.
`Exponential`	The intensity of the light at the edge of the cone falls off exponentially from the intensity of the light at the center of the cone.
`Cosine`	The intensity of the light at the edge of the cone falls off as the cosine of the outer angle from the intensity of the light at the center of the cone.

Light Data

LABELS

ASCII	`LightData`
Binary	`lght ( = 0x6C676874 )`

DATA FORMAT

`Boolean`	`isOn`
`Float32`	`intensity`
`ColorRGB`	`color`

Field descriptions

`isOn`	A value of `True` indicates that the parent light is active (is on). A value of `False` indicates that the parent light is inactive (is off).
`intensity`	The intensity of the parent light at its source. The value in this field must be in the closed interval [0, 1]. 0 is the minimum value; 1 is the maximum value.
`color`	The RGB color of the parent light.

DATA SIZE

20

DESCRIPTION

A light data object specifies the color and source intensity of a parent light, and whether that light is currently active or inactive. A light object that does not have a light data object as a child object should be given the default values indicated below.

Note
A value of less than 1.0 in the intensity field of a light data object affects the color of the parent light. ◆

PARENT HIERARCHY

Data.

PARENT OBJECTS

A light data object always has a parent object; the parent object is always a light object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
    AmbientLight ( )
    LightData (
        True                    # is on
        0.75                    # intensity
        0.7 0.3 0.4             # color
    )
)
```

DEFAULT SETTING

```
True          # is on
1.0           # intensity (full)
1 1 1         # color (white)
```

Ambient Light

LABELS

ASCII	`AmbientLight`
Binary	`ambn` (`= 0x616D626E`)

DATA FORMAT

No data.

DATA SIZE

0

DESCRIPTION

Ambient light is a base amount of light that is added to the illumination of all surfaces in a scene. Ambient light has no apparent source or location; its intensity is constant, and it does not cast shadows.

PARENT HIERARCHY

Shared, shape, light.

PARENT OBJECTS

None.

CHILD OBJECTS

Light data (optional). If no child object is specified, the light should have the properties specified in the default setting of a light data object.

EXAMPLE

```
Container (
    ViewHints ( )
    .
    .
    .
    BeginGroup ( DisplayGroup ( ) )
        Container (
            AmbientLight ( )
            LightData ( ... )
        )
        Container (
            DirectionalLight( )
            LightData ( ... )
        )
    EndGroup ( )
)
```

Directional Lights

LABELS

ASCII	`DirectionalLight`
Binary	`drct ( = 0x64726374 )`

DATA FORMAT

`Vector3D`	`direction`
`Boolean`	`castsShadows`

Field descriptions

`direction`	The direction of the directional light. This vector should be normalized.
`castsShadows`	A value of `True` indicates that the light casts shadows; a value of `False` indicates that the light does not cast shadows.

DATA SIZE

16

DESCRIPTION

A directional light is a light that emits parallel rays in a specific direction. A directional light may be set to cast shadows.

Note
Some style objects also specify whether or not objects in a scene shall receive shadows. However, objects in the scope of a receive shadows style set to `False` do not receive shadows, regardless of whether they are also appropriately situated to receive shadows from a light set to cast shadows. ◆

PARENT HIERARCHY

Shared, shape, light.

PARENT OBJECTS

None.

CHILD OBJECTS

Light data (optional). If no child object is specified, the light should have the properties specified in the default setting of a light data object.

EXAMPLE

```
Container (
    DirectionalLight ( 1 0 0 True )
    LightData ( ... )
)
```

Point Lights

LABELS

ASCII	`PointLight`
Binary	`pntl( = 0x706E746C )`

DATA FORMAT

`Point3D`	`location`
`Attenuation`	`attenuation`
`Boolean`	`castsShadows`

Field descriptions

`location`	The location of the source of the point light.
`attenuation`	This structure determines the amount that the intensity of the light diminishes over distance. See the section "Attenuation and Fall-Off Values" (page 1450) for a description of this structure.
`castsShadows`	A value of `True` specifies that objects illuminated by the light are to cast shadows; a value of `False` specifies that objects illuminated by the light are not to cast shadows.

DATA SIZE

20

DESCRIPTION

A point light is a light that emits rays in all directions from a specific point source. A point light may suffer attenuation over distance and may cast shadows.

Note
Some style objects also specify whether or not objects in a scene shall receive shadows. However, objects in the scope of a receive shadows style set to `False` do not receive shadows, regardless of whether they are also appropriately situated to receive shadows from a light set to cast shadows. ◆

PARENT HIERARCHY

Shared, shape, light.

PARENT OBJECTS

None.

CHILD OBJECTS

Light data (optional). If no child object is specified, the light should have the properties specified in the default setting of a light data object.

EXAMPLE

```
BeginGroup ( DisplayGroup ( ) )
    Triangle ( ... )
    Box ( ... )
    Container (
        PointLight (
            -10, 1, -1              # location
            1 0 1                   # attenuation
            True                    # casts shadows
        )
        LightData ( ... )
    )
EndGroup ( )
```

Spot Lights

LABELS

ASCII	`SpotLight`
Binary	`spot ( = 0x73706F74 )`

DATA FORMAT

`Point3D`	`location`
`Vector3D`	`orientation`
`Boolean`	`castsShadows`
`Attenuation`	`attenuation`
`Float32`	`hotAngle`
`Float32`	`outerAngle`
`FallOffEnum`	`fallOff`

Field descriptions

`location`	The location of the source of the spot light.
`orientation`	The orientation of the cone of light emitted by the spot light. The direction of this vector is toward the light source. This vector should be normalized.
`castsShadows`	A value of `True` specifies that objects illuminated by the light are to cast shadows; a value of `False` indicates that objects illuminated by the light are not to cast shadows.
`attenuation`	This structure determines the amount that the intensity of the light diminishes over distance. See "Attenuation and Fall-Off Values," beginning on page 1450, for a description of this structure.
`hotAngle`	The half-angle (specified in radians) from the center of the cone of light within which the light remains at constant full intensity. The value in this field should be in the half-open interval $[0, \pi/2)$.
`outerAngle`	The half-angle (specified in radians) from the center to the edge of the cone of the spot light. The value in this field should be in the half-open interval $[0, \pi/2)$, and should not be less than the value in the `hotAngle` field.

`fallOff`	The fall-off value for the spot light. The value in this field determines the manner of attenuation of the light from the edge of the hot angle to the edge of the outer angle. See "Attenuation and Fall-Off Values," beginning on page 1450, for a description of the constants that can be used in this field.

DATA SIZE

44

DESCRIPTION

A spot light is a light source that emits a circular cone of light in a specific direction from a specific location. Every spot light has a hot angle and an outer angle that together define the shape of the cone of light and the amount of attenuation that occurs from the center of the cone to the edge of the cone. The attenuation of the light's intensity from the edge of the hot angle to the edge of the outer angle is determined by the light's fall-off value.

Note
Some style objects also specify whether or not objects in a scene shall receive shadows. Thus, conflicting shadowing instructions can be sent to a renderer. The outcome in such a case is renderer-specific, application-specific, or both. ◆

PARENT HIERARCHY

Shared, shape, light.

PARENT OBJECTS

None.

CHILD OBJECTS

Light data. If no child object is specified, the light should have the properties specified in the default setting of a light data object.

EXAMPLE

```
Container (
    SpotLight (
        0 9 0                   # location
        0 1 0                   # orientation
        True                    # castsShadows
        0 0 1                   # attenuation
        0.3                     # hotAngle
        0.5                     # outerAngle
        Linear                  # fallOff
    )
    LightData ( ... )
)
```

Cameras

Camera Placement

LABELS

ASCII	`CameraPlacement`
Binary	`cmpl (= 0x636D706C)`

DATA FORMAT

`Point3D`	`location`
`Point3D`	`pointOfInterest`
`Vector3D`	`upVector`

Field descriptions

`location`	The location (in world-space coordinates) of the eye point of the parent camera.

`pointOfInterest`	The point at which the parent camera is aimed, in world-space coordinates.
`upVector`	The up-vector of the parent camera. This vector should be perpendicular to the viewing direction defined by the values in the `location` and `pointOfInterest` fields. This vector should be normalized.

DATA SIZE

36

DESCRIPTION

A camera placement object defines the location, point of interest, and orientation of its parent camera, in world-space coordinates. The camera vector (also called the view vector) is defined to be the vector `pointOfInterest - location`. This vector is normal to the projection plane and to the clipping planes, and the distances from the camera to those planes are measured along this vector.

A camera placement object determines the coordinate system of the projection plane as follows. The origin of the projection plane is the point at the intersection of the projection plane and the line through the location and point of interest. The *y* axis of the projection plane coincides with the projection onto the projection plane of the up vector, and the *x* axis of the projection plane is the axis such that it, the *y* axis of the projection plane, and the inverse of the camera vector form a righthanded coordinate system.

If no camera placement object is specified for a camera, that camera should receive the default camera placement values specified below.

PARENT HIERARCHY

Data.

PARENT OBJECTS

View angle aspect camera, view plane camera, orthographic camera. A camera placement object always has a camera as a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
    ViewAngleAspectCamera ( ... )
    CameraPlacement (
        0 0 30                  # location on z axis
        0 0 0                   # point of interest is the origin
        0 1 0                   # up vector aligned with yaxis
    )
)
```

DEFAULT VALUES

```
0 0 1         # location
0 0 0         # pointOfInterest
0 1 0         # upVector
```

Camera Range

LABELS

ASCII	`CameraRange`
Binary	`cmrg ( = 0x636D7267 )`

DATA FORMAT

`Float32`	`hither`
`Float32`	`yon`

Field descriptions

`hither`	The distance from the location of the parent camera to the near clipping plane. The value in this field should be greater than 0.
`yon`	The distance from the location of the parent camera to the far clipping plane. The value in this field should be greater than the value in the `hither` field.

DATA SIZE

8

DESCRIPTION

A camera range object is used to set the near and far clipping planes of its parent camera. Distances are measured in the direction defined by the camera vector, which is normal to both clipping planes.

PARENT HIERARCHY

Data.

PARENT OBJECTS

View angle aspect camera, view plane camera, orthographic camera. A camera placement object always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
    ViewPlaneCamera ( ... )
    CameraPlacement ( ... )
    CameraRange (
        .01                             # hither
        75                              # yon
    )
)
```

Camera Viewport

LABELS

ASCII	`CameraViewPort`
Binary	`cmvp ( = 0x636D7670 )`

DATA FORMAT

`Point2D`	`origin`
`Float32`	`width`
`Float32`	`height`

Field descriptions

`origin`	The origin of the view port of the parent camera. The abscissa and ordinate of this point should lie in the closed interval [–1, 1]. The value in this field is the upper-left corner of the view port.
`width`	The width of the view port of the parent camera. The value in this field should lie in the half-open interval (0, 2], and should not be greater than the absolute value of the difference between 1 and the abscissa of the origin.
`height`	The height of the view port of the parent camera. The value in this field should lie in the half-open interval (0, 2], and should not be greater than the difference between –1 and the ordinate of the origin.

DATA SIZE

16

DESCRIPTION

Every camera specifies the dimensions of the largest (rectangular) image that that camera can produce (called the *parent image*), either explicitly or implicitly. The parent image may be specified by giving the coordinates of its vertices, by giving the height to width ratio of its sides, or in some other fashion. The camera view port object specifies the subregion of the parent image that is actually to be drawn. The value in the `origin` field defines the upper left corner of the view port; the values in the other two fields determine the lengths of the sides of the view port.

The default setting specified below sets the view port equal to the parent image. Other settings may be used to clip the parent image to desired specifications.

Camera view port specifications are made in a coordinate system in which the height-to-width ratio of the parent image is one to one, and the coordinates of the upper-left and lower-right corners of that image are (–1, 1) and (1, –1), respectively. The actual height-to-width ratio of the parent image may not be one to one. If not, then view port specifications should be made under the assumption that the view port will be rescaled by the inverse of the height-to-width ratio of the parent image after the view port specifications have been made. Thus, if the height-to-width ratio of the parent image is *i/j*, and the height-to-width ratio of the image actually to be drawn is *i′/j′*, then the height-to-width ratio of the rectangle specified in the view port should be *i′j/ij′*. Any view port having a different height-to-width ratio will result in a distorted image.

PARENT HIERARCHY

Data.

PARENT OBJECTS

View angle aspect camera, view plane camera, orthographic camera. A camera viewport object always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
CameraViewPort (
    -0.5 0.5
    1.0
    1.0
)
```

DEFAULT VALUES

```
-1 1          # origin at upper left corner of the parent image
2             # width is the entire width of the parent image
2             # height is the entire height of the parent image
```

Orthographic Cameras

LABELS

ASCII	`OrthographicCamera`
Binary	`orth ( = 0x6F727468 )`

DATA FORMAT

`Float32`	`left`
`Float32`	`top`
`Float32`	`right`
`Float32`	`bottom`

Field descriptions

`left` The *x* coordinate (in the camera's coordinate system) of the upper left corner of the front face of the view volume; or,

	the distance from the center of the camera lens (that is, the view rectangle) to the left side of the lens.
`top`	The *y* coordinate (in the camera's coordinate system) of the upper left corner of the front face of the view volume; or, the distance from the center of the camera lens (that is, the view rectangle) to the top side of the lens.
`right`	The *x* coordinate (in the camera's coordinate system) of the lower right corner of the front face of the view volume; or, the distance from the center of the camera lens (that is, the view rectangle) to the right side of the lens.
`bottom`	The *y* coordinate (in the camera's coordinate system) of the lower right corner of the front face of the view volume; or, the distance from the center of the camera lens (that is, the view rectangle) to the left side of the lens.

DATA SIZE

16

DESCRIPTION

An orthographic camera is a parallel projection camera that employs an orthographic projection to obtain its image. The direction of projection is the opposite of the camera vector (that is, `location - pointOfInterest`), the projection plane is the near clipping plane, and the projection is thus along a normal to the projection plane. The origin of the projection plane is the point at `hither` (camera vector); if the absolute values of the fields `top` and `bottom` are equal, and the absolute values of the fields `left` and `right` are equal, then the origin of the projection plane is at the center of the front face of the view volume.

PARENT HIERARCHY

Shared, shape, camera.

PARENT OBJECTS

View hints (sometimes).

CHILD OBJECTS

Camera placement, camera range, camera view port (optional). If a camera does not have one of these child objects, then it should be assigned the default values specified in the section on that child object.

EXAMPLE

```
OrthographicCamera (
    -10
    -10
    10
    10
)
```

View Plane Cameras

LABELS

ASCII	`ViewPlaneCamera`
Binary	`vwpl (= 0x7677706C)`

DATA FORMAT

```
Float32         viewPlane
Float32         halfWidthAtViewPlane
Float32         halfHeightatViewPlane
Float32         centerXOnViewPlane
Float32         centerYOnViewPlane
```

Field descriptions

`viewPlane` The distance from the camera location to the view plane.

`halfWidthAtViewPlane`
One half the width of the view plane window.

`halfHeightAtViewPlane`
: The value in the `halfWidthAtViewPlane` field divided by the horizontal-to-vertical aspect ratio of the view port. The value in this field determines the half-height of the view plane window.

`centerXOnViewPlane`
: The *x* coordinate of the center of the view plane window, specified in the view plane coordinate system.

`centerYOnViewPlane`
: The *y* coordinate of the center of the view plane window, specified in the view plane coordinate system.

DATA SIZE

20

DESCRIPTION

A view plane camera is a type of perspective camera defined in terms of an arbitrary view plane. The camera vector is normal to the view plane, and the distance from the camera location to the view plane is measured in the direction defined by the camera vector. The window on the view plane and its center are defined in the projection plane coordinate system determined by the camera's camera placement object. The view volume of a view plane camera is determined by the four rays through the camera location and through the four corners of the rectangular window on the view plane, together with the two clipping planes. The view volume is the frustum whose top is the rectangle having as its vertices the intersections of these four rays with the near clipping plane and whose base is the rectangle having as its vertices the intersections of these rays with the far clipping plane.

The center of projection of a view plane camera is the camera location point. If the center of the window defined by a view plane camera is not at the origin of the view plane, then the camera yields an off-axis view. The projection determined by a view plane camera may have one, two, or three principal vanishing points.

A view plane camera may be used to obtain a close-up image of a single object by using the approximate center and dimensions of that object to specify the size and location of the window on the view plane.

PARENT HIERARCHY

Shared, shape, camera.

PARENT OBJECTS

View hints (sometimes).

CHILD OBJECTS

Camera placement, camera view port, camera range (optional). If a camera does not have one of these child objects, then it should be assigned the default values specified in the section on that child object.

EXAMPLE

```
Container (
    ViewPlaneCamera (
        20
        15.0
        15.0
        18
        29
    )
    CameraPlacement ( ... )
    CameraRange ( ... )
    CameraViewPort ( ... )
)
```

View Angle Aspect Cameras

LABELS

ASCII	`ViewAngleAspectCamera`
Binary	`vana ( = 0x76616E61 )`

DATA FORMAT

```
Float32          fieldOfView
Float32          aspectRatioXtoY
```

Field descriptions

`fieldOfView` An angle, specified in radians, that defines the maximum field of view of the camera. The value in this field should lie in the open interval $(0, \pi)$.

`aspectRatioXtoY` The horizontal-to-vertical aspect ratio of the camera. If the value in this field is less than 1.0, the camera's field of view is vertical; otherwise, the camera's field of view is horizontal.

DATA SIZE

8

DESCRIPTION

A view angle aspect camera is a type of perspective camera defined in terms of a field of view angle and a horizontal-to-vertical aspect ratio. The aspect ratio determines the ratio of the base to the height of the rectangles that define the top and base of the camera's view volume. These rectangles lie in the near and far clipping planes, respectively, are upright in the camera's coordinate system, and are centered at the points of intersection of the line along the camera vector and the clipping planes.

If the aspect ratio is less than 1.0, then the field of view angle is in the $x = 0$ plane of the camera's coordinate system. Otherwise, the field of view angle is in the $y = 0$ plane of the camera's coordinate system. In both cases the rays that define the angle intersect in the camera location point, and the field of view angle is bisected by the ray from the camera location defined by the camera vector. The center of projection is the camera location point. The view volume of a view angle aspect camera is symmetrical about its center line. The method of projection determined by a view angle aspect camera has one principal vanishing point, located at the origin of the projection plane.

PARENT HIERARCHY

Shared, shape, camera.

PARENT OBJECTS

View hints (sometimes).

CHILD OBJECTS

Camera placement, camera view port, camera range (optional). If a camera does not have one of these child objects, then it should be assigned the default values specified in the section on that child object.

EXAMPLE

```
Container (
    ViewAngleAspectCamera (
        1.7
        1.0
    )
    CameraPlacement ( ... )
    CameraRange ( ... )
    CameraViewPort ( ... )
)
```

Groups

Display Groups

LABELS

ASCII	`DisplayGroup`
Binary	`dspg ( = 0x6C697374 )`

DATA FORMAT

No data.

DATA SIZE

0

DESCRIPTION

A display group is a list of drawable objects and containers the root objects of which are drawable objects. Types of drawable objects include geometric objects, attribute sets, styles, transforms, and other display groups. A display group is delimited by begin group and end group objects.

PARENT HIERARCHY

Shared, shape, group.

PARENT OBJECTS

None.

CHILD OBJECTS

Display group state (optional). If no child object is specified, group state flags should be set to the default values specified in "Display Group States," beginning on page 1483.

EXAMPLE

```
BeginGroup ( Display Group( ) )
    SubdivisionStyle ( Constant 32 32 )
    Container (
        Mesh ( ... )
        VertexAttributeSetList ( ... )
        FaceAttributeSetList ( ... )
    )
```

```
    Container (
        Mesh ( ... )
        VertexAttributeSetList ( ... )
        FaceAttributeSetList ( ... )
    )
    .
    .
    .
EndGroup ( )
```

Ordered Display Groups

LABELS

ASCII	`OrderedDisplayGroup`
Binary	`ordg` (`= 0x6F72646C`)

DATA FORMAT

No data.

DATA SIZE

0

DESCRIPTION

An ordered display group is a display group in which the objects listed are sorted by type. The elements of an ordered display group are listed in the following order: transforms, styles, attribute sets, shaders, geometric objects, other groups. An ordered display group is delimited by `BeginGroup` and `EndGroup` objects.

PARENT HIERARCHY

Shared, shape, group, display group.

PARENT OBJECTS

None.

CHILD OBJECTS

Display group state (optional). If no child object is specified, group state flags should be set to the default values specified in "Display Group States," beginning on page 1483.

EXAMPLE

```
BeginGroup ( OrderedDisplayGroup ( ) )
    RotateTransform ( ... )
    ScaleTransform ( ... )
    SubdivisionStyle ( ... )
    BackfacingStyle ( ... )
    BeginGroup ( DisplayGroup ( ) )
        .
        .
        .
    EndGroup ( )
EndGroup ( )
```

Light Groups

LABELS

ASCII	`LightGroup`
Binary	`lghg ( = 0x676C6768 )`

DATA FORMAT

No data.

DATA SIZE

0

DESCRIPTION

A light group is simply a list of light objects. A light group is delimited by begin group and end group objects.

PARENT HIERARCHY

Shared, shape, group.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
BeginGroup ( LightGroup ( ) )
    AmbientLight ( )
    DirectionalLight ( ... )
    SpotLight ( ... )
EndGroup ( )
```

I/O Proxy Display Groups

LABELS

ASCII	`IOProxyDisplayGroup`
Binary	`iopx (= 0x70727879)`

DATA FORMAT

No data.

DATA SIZE

0

DESCRIPTION

An I/O proxy display group is used to place distinct specifications of the same model together in a group. The purpose of an I/O proxy display group is to permit a reading application that does not recognize all specifications of a model to pass over those that it does not recognize until it encounters one that it does recognize and can use to recover the model. For example, a pentagon may be represented by either a mesh or a polygon. If both representations are placed together in an I/O proxy display group, then a reading application that recognizes meshes but does not recognize polygons can recover the pentagon from its mesh representation.

Representations of a model in an I/O proxy display group should appear in preferential order: any representation of a model is to be preferred to any other representation of that model occurring later in the group. While drawing, bounding, or picking, the reading application should use the first representation of the model that it recognizes and should ignore all other representations.

PARENT HIERARCHY

Shared, shape, group.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
BeginGroup ( IOProxyDisplayGroup ( ) )
    Polygon ( ... )                         # first preference
    GeneralPolygon ( ... )                  # second preference
    Mesh                                    # third preference
EndGroup ( )
```

Info Groups

LABELS

ASCII	`InfoGroup`
Binary	`info ( = 0x696E666F )`

DATA FORMAT

No data.

DATA SIZE

0

DESCRIPTION

An info group is a list of string objects delimited by begin group and end group objects. An info group allows objects containing information in text form to be placed together in a group.

PARENT HIERARCHY

Shared, shape, group.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
BeginGroup ( InfoGroup ( ) )
    CString ( ... )
    .
    .
    .
    CString ( ... )
EndGroup ( )
```

Groups (Generic)

LABELS

ASCII	`Group`
Binary	`grup ( = 0x67727570 )`

DATA FORMAT

No data.

DATA SIZE

0

DESCRIPTION

A group (generic) is simply a list of drawable objects, delimited by begin group and end group objects.

PARENT HIERARCHY

Shared, shape, group.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
BeginGroup ( Group ( ) )
    .
    .
    .
EndGroup ( )
```

Begin Group Objects

LABELS

ASCII	`BeginGroup`
Binary	`bgng ( = 0x62676E67)`

DESCRIPTION

A begin group object is used to declare a group and to delimit the start of that group. Every group must begin with a begin group object.

PARENT HIERARCHY

3DMF.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
BeginGroup(
    DisplayGroup ( )                # empty group
)
EndGroup ( )
```

End Group Objects

LABELS

ASCII	`EndGroup`
Binary	`endg ( = 0x656E6467 )`

DATA FORMAT

No data.

DATA SIZE

0

DESCRIPTION

An end group object is placed immediately after the last object in a group and is used to delimit that group.

PARENT HIERARCHY

3DMF.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
BeginGroup ( DisplayGroup ( ) )        # empty group
EndGroup ( )
```

Display Group States

LABELS

ASCII	`DisplayGroupState`
Binary	`dgst (= 0x64677374)`

DISPLAY GROUP STATE FLAGS

`None`	`0x00000000`
`IsInline`	`0x00000001`
`DoNotDraw`	`0x00000002`
`NoBoundingBox`	`0x00000004`
`NoBoundingSphere`	`0x00000008`
`DoNotPick`	`0x00000010`

Constant descriptions

`None`	No flags are specified.
`IsInline`	The parent group is to be executed inline (that is, without pushing the graphics state on a stack before execution and popping it after execution). This flag is used to prevent the objects in the parent group from inheriting properties specified at a higher level in a hierarchical model containing the parent group. If this flag is set, then objects

	in the parent group receive only those properties specified in that group.
`DoNotDraw`	The parent group is not to be drawn when rendering or picking. If this flag is set, then the parent group is not to be traversed when it is encountered in a hierarchical model.
`NoBoundingBox`	The bounding box of the parent group is not to be used for rendering.
`NoBoundingSphere`	The bounding sphere of the parent group is not to be used for rendering.
`DoNotPick`	The parent group is not eligible for inclusion in the hit list of a pick object.

DATA FORMAT

`DisplayGroupStateFlags` `traversalFlags`

Field descriptions

`traversalFlags`	A bitfield expression specifying one or more display group state flags.

DATA SIZE

4

DESCRIPTION

A display group state object is used to specify a set of flags that determines how its parent display group is to be traversed during rendering or picking and whether a bounding box or bounding sphere is to be used during rendering. If a display group does not have a display group state object as a child object, that group's state flags should be set to the default state specified below.

In a text file, a display group state object should be placed together with a group object in the begin group object that immediately precedes that group.

PARENT HIERARCHY

Data.

PARENT OBJECTS

Display group, ordered display group. A display group state object always has a parent object.

CHILD OBJECTS

None.

DEFAULT DISPLAY GROUP STATE FLAGS

```
None (= 0x00000000)
```

EXAMPLE

```
BeginGroup (
    DisplayGroup ( )
    DisplayGroupState ( DoNotPick )
)
.
.
.
EndGroup ( )
```

Renderers

Wireframe Renderers

LABELS

ASCII	`WireFrame`
Binary	`wrfr (= 0x77726672)`

DATA FORMAT

No data.

DATA SIZE

0

DESCRIPTION

A wireframe renderer creates line drawings of models. Such a renderer does not decompose polylines or polygons during rendering. It can render all back-facing, point, and edge drawing styles.

PARENT HIERARCHY

Shared, renderer.

PARENT OBJECTS

View hints (sometimes).

CHILD OBJECTS

None.

EXAMPLE

```
Container (
    ViewHints ( )
    Wireframe ( )
    ViewPlaneCamera ( ... )
    PointLight ( ... )
    BeginGroup ( DisplayGroup ( ) )
        .
        .
        .
    EndGroup ( )
)
```

Interactive Renderers

LABELS

ASCII	`InteractiveRenderer`
Binary	`ctwn ( = 0x6374776E )`

DATA FORMAT

No data.

DATA SIZE

0

DESCRIPTION

The interactive renderer uses a fast and accurate depth-sorting algorithm for drawing solid, shaded surfaces as well as vectors. The interactive renderer is also capable of rendering highly detailed, complex models with very realistic surface illumination and shading, but at the expense of time and memory.

PARENT HIERARCHY

Shared, renderer.

PARENT OBJECTS

View hints (sometimes).

CHILD OBJECTS

None.

EXAMPLE

```
Container (
    ViewHints ( )
    InteractiveRenderer ( )
    ViewPlaneCamera ( ... )
    PointLight ( ... )
    BeginGroup ( DisplayGroup ( ) )
        .
        .
        .
    EndGroup ( )
)
```

Generic Renderers

LABELS

ASCII	`GenericRenderer`
Binary	`gnrr (= 0x676E7272)`

DATA FORMAT

No data.

DATA SIZE

0

DESCRIPTION

A generic renderer performs no rendering functions, but may be used to pick or to accumulate state.

PARENT HIERARCHY

Shared, renderer.

PARENT OBJECTS

View hints (sometimes).

CHILD OBJECTS

None.

EXAMPLE

```
Container (
    ViewHints ( )
    GenericRenderer ( )
    ViewPlaneCamera ( ... )
    PointLight ( ... )
    BeginGroup ( DisplayGroup ( ) )
        .
        .
        .
    EndGroup ( )
)
```

Shaders

Shader Data Objects

LABELS

ASCII	`Shader`
Binary	`shdr` (= `0x73686472`)

SHADER UV BOUNDARY TYPES

`Wrap`	`0x00000000`
`Clamp`	`0x00000001`

Constant descriptions

`Wrap`	Values outside the valid range of *uv* values are to be wrapped. To wrap a shader effect is to replicate the entire effect across the mapped area.
`Clamp`	Values outside the valid range of *uv* values are to be clamped. To clamp a shader effect is to replicate the boundaries of the effect across the portion of the mapped area that lies outside the valid range.

DATA FORMAT

`ShaderUVBoundaryEnum`	`uBounds`
`ShaderUVBoundaryEnum`	`vBounds`

Field descriptions

`uBounds`	The value in this field determines whether values in the *u* parametric direction that lie outside the valid range are wrapped or clamped by the parent shader.
`vBounds`	The value in this field determines whether values in the *v* parametric direction that lie outside the valid range are wrapped or clamped by the parent shader.

DATA SIZE

8

DESCRIPTION

A shader data object is a boundary-handling method specifier that determines how a parent shader handles parametric *uv* values that are outside the valid range (namely, 0 to 1).

PARENT HIERARCHY

Data.

PARENT OBJECTS

Any shader. A shader data object always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
    CustomShader ( ... )
    ShaderData ( Wrap Clamp )
)
```

DEFAULT VALUES

```
Wrap Wrap
```

Texture Shaders

LABELS

ASCII	`TextureShader`
Binary	`txsu (= 0x74787375)`

DATA FORMAT

No data.

DATA SIZE

0

DESCRIPTION

A texture shader is used to apply a texture to a surface in shading.

PARENT HIERARCHY

Shared, shape, shader, surface shader.

PARENT OBJECTS

None.

CHILD OBJECTS

Pixmap texture object. A texture shader always has one child object.

EXAMPLE

```
Container (
    TextureShader ( )
    PixmapTexture ( ... )
)
```

Pixmap Texture Objects

LABELS

ASCII	`PixmapTexture`
Binary	`txpm (= 0x7478706D)`

ENDIAN TYPES

`BigEndian`	`0x00000000`
`LittleEndian`	`0x00000001`

Constant descriptions

`BigEndian`	Packing is to be done in a big-endian manner.
`LittleEndian`	Packing is to be done in a little-endian manner.

PIXEL TYPES

RGB8	0x00000000
RGB16	0x00000001
RGB24	0x00000002
RGB32	0x00000003

Constant descriptions

RGB8	8 bits are devoted to each pixel in the pixmap.
RGB16	16 bits are devoted to each pixel in the pixmap.
RGB24	24 bits are devoted to each pixel in the pixmap.
RGB32	32 bits are devoted to each pixel in the pixmap.

DATA FORMAT

Uns32	width
Uns32	height
Uns32	rowBytes
Uns32	pixelSize
PixelTypeEnum	pixelType
EndianEnum	bitOrder
EndianEnum	byteOrder
RawData	image[rowBytes * height]

Field descriptions

width	The width of the pixmap. The value in this field must be greater than 0.
height	The height of the pixmap. The value in this field must be greater than 0.
rowBytes	The number of bytes in a row of the pixmap. The value in this field cannot be less than the product of the values in the width and pixelSize fields.
pixelSize	The size of each pixel in the pixmap. The value in this field must be greater than 0 and less than 32.
pixelType	The type of the pixels of the pixmap: 0x00000000 = RGB32 0x00000001 = ARGB32

	0x00000010 = RGB16 0x00000011 = ARGB16
`bitOrder`	The order in which the bits in a byte are addressed. This field must contain one of the constants `BigEndian` or `LittleEndian`.
`byteOrder`	The order in which the bytes in a word are addressed. This field must contain one of the constants `BigEndian` or `LittleEndian`.
`image[]`	The array that defines the pixmap.

DATA SIZE

```
28 + rowBytes * height + padding
```

DESCRIPTION

A pixmap texture object is a generic method of transferring pixmap data that is used in conjunction with a texture shader.

PARENT HIERARCHY

Shared, texture.

PARENT OBJECTS

Texture shader. A pixmap texture object sometimes, but not always, has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
PixmapTexture (
    256 256                         # width/height
    128                             # rowBytes
    32                              # pixelSize
```

```
    RGB24
    BigEndian BigEndian
    0x00123232...
    0x...
)
```

View Objects

View Hints

LABELS

ASCII	`ViewHints`
Binary	`vwhn ( = 0x7677686E )`

DATA FORMAT

No data.

DATA SIZE

0

DESCRIPTION

The view hints object is used to group together all of the objects needed to render an image from a model (that is, a renderer, a camera, lights, and any additional information to be supplied to the renderer). These other objects occur as child objects to the view hints object; a container may be used to group them together. The container holding a view hints object and its associated rendering specifications should be placed immediately before the models to be rendered according to those specifications.

A metafile may contain more than one view hints object. If a metafile contains more than one view hints object, the specifications associated with each view hints object are inherited by all subsequent view hints objects, unless

overridden by contrary specifications. Accordingly, a subsequent view hints object need have as child objects only those specifications that differ from those of its predecessors. For example, you may wish to render the same model using different cameras, while keeping the lights and other specifications intact. Once the initial specifications have been made, you need only specify a different camera together with a new view hints object. The model may be placed in the scope of a subsequent view hints object through the use of a reference object; the specification of the model need not be repeated.

PARENT HIERARCHY

Shared.

PARENT OBJECTS

None.

CHILD OBJECTS

Renderer, camera, lights (as many as desired), attribute set, image dimensions, image mask, image clear color (all optional).

EXAMPLE

```
3DMetafile ( 1 0 Normal toc> )
Container (
    ViewHints ( )
    Container (
        ViewAngleAspectCamera ( 0.73 1.0 )
        CameraPlacement (
            0 0 30
            0 0 0
            0 1 0
        )
    )
    DirectionalLight ( -0.7 -0.7 -0.65 )
    Container (
        AttributeSet ( )
        DiffuseColor ( 0.2 0.2 0.2 )
```

```
        SpecularControl ( 3 )
    )
    ImageDimensions ( 200 200 )
)
ref1:
BeginGroup ( DisplayGroup ( ) )
.
.
.
EndGroup ( )
Container (
    ViewHints ( )
    Container (
        ViewAngleAspectCamera ( 0.73 1.0 )
        CameraPlacement (
            0 10 0
            0 0 0
            0 1 0
        )
    )
)
Reference ( 1 )
```

Image Masks

LABELS

ASCII	`ImageMask`
Binary	`immk ( = 0x696D6D6B )`

DATA FORMAT

`Uns32`	`width`
`Uns32`	`height`
`Uns32`	`rowBytes`
`RawData`	`image[rowBytes * height]`

Field descriptions

`width`	The width, in bits, of the bitmap whose bits are listed in the array `image[]`. The value in this field should be greater than 0.
`height`	The height, in bits, of the bitmap whose bits are listed in the array `image[]`. The value in this field should be greater than 0.
`rowBytes`	The number of bytes in a row of the bitmap.
`image[]`	An array of bit specifications.

DATA SIZE

```
12 + (rowBytes * height) + padding
```

DESCRIPTION

An image mask is a bitmap that is used to mask out certain portions of an image. The values in the `width` and `height` fields of an image mask specify the boundaries of the rectangular subregion of an image that is actually to be drawn. (Width and height are measured from the upper-left corner of the image to which a mask is applied.) Each bit listed in the array `images[]` corresponds to 1 pixel in the rectangle defined by the width and height of the mask. If a bit is set, then the corresponding pixel is drawn with the color determined by the underlying image. If a bit is clear, then the corresponding pixel is drawn black. Normally, an image mask is applied to an image after that image has been rasterized.

An image dimensions object may be used together with an image mask: the former may be used to clip an image, and the latter may be used to filter the clipped image.

PARENT HIERARCHY

Data, view hints data.

PARENT OBJECTS

View hints. An image mask always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
3DMetafile ( 1 0 Normal toc> )
Container (
    ViewHints ( )
    ImageDimensions ( 32 32 )
    ImageClearColor ( 1 1 1 )
    ImageMask (
        32 32                                           # width, height
        4                                               # rowBytes
        BigEndian                                       # bitOrder
        0x000000000FFFF8000FFFF8000FFFF800
        0x0FFFF8000FFFF8000FFFF8000FFFFFE0
        0x0FFFFFE00FFFFFE00FFFFFE00FFFFFE0
        0x0FFFFFE00FFFFFE00FFFFFE00FFFFFE0
        0x0FFFFFE00FFFFFE00FFFFFE00FFFFFE0
        0x0FFFFFE00FFFFFE00FFFFFE00FFFFFE0
        0x0C61FFE00F24FFE00E64FFE00F24FFE0
        0x0F24FFE00C61FFE00FFFFFE000000000
    )
)
Rotate ( X 0.25 )
Rotate ( Y 0.23 )
Container (
    Torus ( 0 0.7 0 0 0 1 1 0 0 0 0 0 0.7 )
    Container (
        AttributeSet ( )
        DiffuseColor ( 0.2 0.9 0.9 )
    )
)
```

Image Dimensions Objects

LABELS

ASCII	`ImageDimensions`
Binary	`imdm (= 0x696D646D)`

DATA FORMAT

`Uns32`	`width`
`Uns32`	`height`

Field descriptions

`width`	The preferred width, in pixels, of the displayed portion of an image.
`height`	The preferred height, in pixels, of the displayed portion of an image.

DATA SIZE

8

DESCRIPTION

An image dimensions object is used to specify the height and width of the rectangular portion of an image that is to be displayed. The height and width of an image dimensions object are measured from the upper-left corner of the image to which that image dimensions object is applied. Normally, an image is rasterized before an image dimensions object is applied to it. An image dimensions object may be used together with an image mask: the former may be used to clip an image, and the latter may be used to filter the clipped image.

PARENT HIERARCHY

Data, view hints data.

PARENT OBJECTS

View hints. An image dimensions object always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
    ViewHints ( )
    ImageDimensions ( 32 32 )
    ImageMask ( ... )
)
```

Image Clear Color Objects

LABELS

ASCII	`ImageClearColor`
Binary	`imcc ( = 0x696D6363 )`

DATA FORMAT

`ColorRGB`	`clearColor`

Field descriptions

`clearColor`	The RGB color to be given to the visible background of a model when an image is rendered from that model.

DATA SIZE

4

DESCRIPTION

An image clear color object is used to assign color to the background of a model in a rendered image when the model does not completely fill that image.

PARENT HIERARCHY

Data, view hints data.

PARENT OBJECTS

View hints. An image clear color object always has a parent object.

CHILD OBJECTS

None.

EXAMPLE

```
Container (
    ViewHints ( )
    ImageDimensions ( ... )
    ImageClearColor ( 1 1 1 )
    .
    .
    .
)
```

Unknown Objects

Unknown Text

LABELS

ASCII `UnknownText`

Binary	`uktx` (`= 0x756B7478`)

DATA FORMAT

`String`	`asciiName`
`String`	`contents`

Field descriptions

`asciiName`	The object type of the unknown object, enclosed in double quotation marks.
`contents`	The specification (without encapsulation) of the unknown object, enclosed in double quotation marks. Blank space and comments in the original object specification of the unknown object may be omitted when this field is written.

DATA SIZE

```
sizeof(asciiName) + sizeof(contents)
```

DESCRIPTION

An unknown text object is used to transport unknown data found in a text file. It is an encapsulated replica of that unknown data. In the usual case, an unknown text object contains an ill-formed object specification. Your file reading program may be designed to transport the data contained in an unknown text object, to validate and convert the data to a specification of a known object, or to discard the data.

An unknown text object may occur in a binary file as well as in a text file.

PARENT HIERARCHY

Shared, shape.

PARENT OBJECTS

Any object that may have a child object may be a parent object to an unknown text object.

CHILD OBJECTS

None.

EXAMPLE

```
UnknownText (
   "Sphere"                                  # unknown object type
   "1 0 0    0 1 0     0 0 1     0 0 a"  # illegal specification
)
```

Unknown Binary

LABELS

ASCII	`UnknownBinary`
Binary	`ukbn` (`= 0x756B626E`)

DATA FORMAT

`Int32`	`objectType`
`Uns32`	`objectSize`
`EndianEnum`	`byteOrder`
`RawData`	`objectData[objectSize]`

Field descriptions

`objectType`	The binary representation of the type of the unknown object.
`objectSize`	The size of the unknown object.
`byteOrder`	The byte order of the unknown object. The information in this field is needed to transport unknown data between processors and permits parsing endian-specific primitives contained in the object data: 0x00000000 = BigEndian 0x00000001 = LittleEndian

`objectData[]` The specification of the unknown object in the form of raw data.

DATA SIZE

```
12 + sizeof(objectData)
```

DESCRIPTION

An unknown binary object is used to transport unknown data found in a binary file. It is an encapsulated replica of that unknown data. In the usual case, an unknown binary object contains an ill-formed object specification. Your file reading program may be designed to transport the data contained in an unknown text object, to validate and convert the data to a specification of a known object, or to discard the data.

An unknown binary object may occur in a text file as well as in a binary file.

PARENT HIERARCHY

Shared, shape.

PARENT OBJECTS

None.

CHILD OBJECTS

None.

EXAMPLE

```
UnknownBinary (
    1701605476
    4
    BigEndian
    0x0AB2
)
```

QuickDraw 3D RAVE

This chapter describes the QuickDraw 3D Renderer Acceleration Virtual Engine (RAVE), the part of the Macintosh system software that controls 3D drawing engines (also known as 3D drivers). As explained more fully below, a drawing engine is software that supports the low-level rasterization operations required for interactive 3D rendering. To achieve interactive performance, a drawing engine is often associated with some hardware device designed specifically to accelerate the 3D rasterization process.

QuickDraw 3D RAVE is used internally by QuickDraw 3D, the 3D graphics library from Apple Computer, Inc. that you can use to create, configure, render, and interact with models of three-dimensional objects. For most 3D drawing and interaction, you should use the high-level application programming interfaces provided by QuickDraw 3D. In some cases, however, you might need to use the low-level services provided by QuickDraw 3D RAVE. You can use QuickDraw 3D RAVE if

- you are writing a specialized application (such as a game-development framework) that needs to take advantage of Apple's optimized software rasterizers and any available 3D acceleration hardware
- you are writing interactive software (such as a game or other entertainment software) that requires the extremely fast 3D rendering that can be achieved with a very low-level, lightweight graphics library
- you are developing 3D acceleration hardware or software that is to be accessed by any applications rendering 3D images

To use this document, you should already be familiar with QuickDraw 3D, described in *3D Graphics Programming With QuickDraw 3D*. You should also be familiar with low-level 3D rendering algorithms. The bibliography lists a number of standard 3D reference books that document those algorithms.

This document begins by describing the basic capabilities of QuickDraw 3D RAVE. Then it shows how to use some of those capabilities to find the available

drawing engines, select and configure a drawing engine, and use that drawing engine to draw 3D images. The section "Writing a Drawing Engine," beginning on page 1524, shows how to add a new drawing engine to those already available for use by QuickDraw 3D RAVE. You need to read this section only if you are developing custom 3D acceleration hardware or software.

The section "QuickDraw 3D RAVE Reference," beginning on page 1535, provides a complete reference to the constants, data structures, and functions provided by QuickDraw 3D RAVE.

IMPORTANT

QuickDraw 3D RAVE is used by the interactive renderer supplied as part of QuickDraw 3D version 1.0. However, the features described here that provide compatibility with **OpenGL**™ are not supported by that renderer and are subject to change in future versions of QuickDraw 3D RAVE. ▲

About QuickDraw 3D RAVE

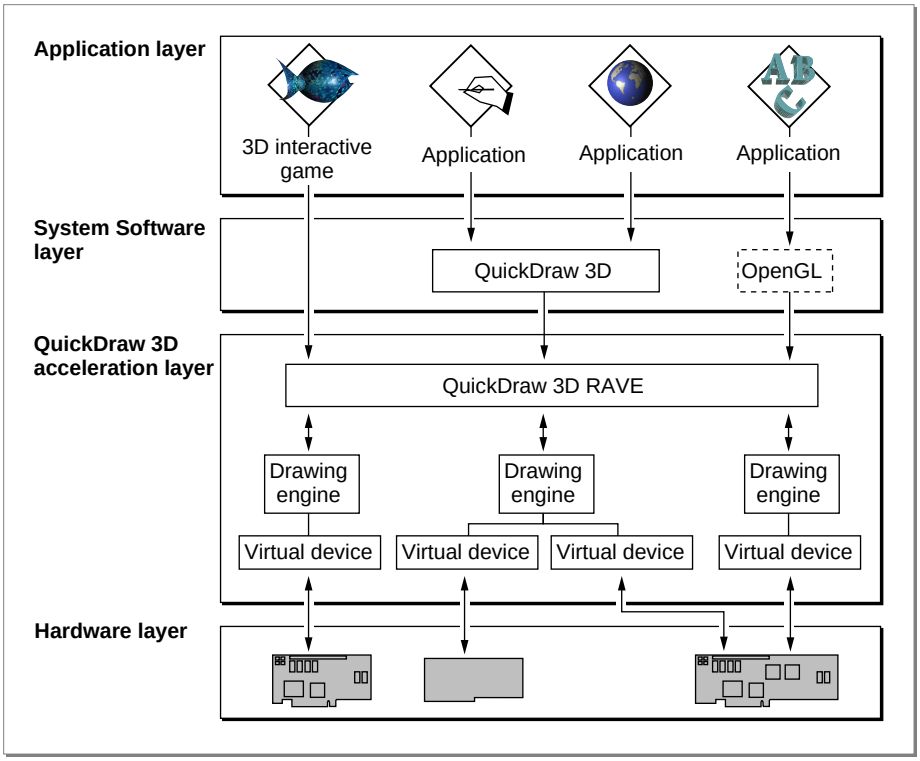
The **QuickDraw 3D Renderer Acceleration Virtual Engine** (or, more briefly, **QuickDraw 3D RAVE**) is the part of the Macintosh system software that controls 3D drawing engines. A **drawing engine** is software that supports low-level rasterization operations—that is, the process of determining values for pixels in an image on the screen or some other medium. You are probably already familiar with QuickDraw, which is a 2D drawing engine. The 3D drawing engines managed by QuickDraw 3D RAVE differ from 2D drawing engines in several important respects:

- A 3D drawing engine must support a *z* (or depth) value for **hidden surface removal** (removing any surfaces in a model that are hidden by opaque surfaces of objects).
- A 3D drawing engine typically supports **double buffering,** the use of two different buffers to store pixel images. Double buffering helps reduce the flashing caused by redrawing an image. Double buffering can also be used to avoid tearing artifacts caused by updating a window at high speed.
- A 3D drawing engine typically supports special rasterization modes, such as texture mapping or constructive solid geometry.

In almost all other respects, a 3D drawing engine operates just like a 2D drawing engine. You draw objects by sending drawing commands to the drawing engine, which interprets the commands and constructs a rasterized image. A 3D drawing engine is often associated with hardware designed specifically to accelerate the 3D rasterization process.

Figure 23-1 illustrates the position of QuickDraw 3D RAVE in relation to drawing engines, the clients that call it, and the devices driven by the drawing engines.

Figure 23-1 The position of QuickDraw 3D RAVE



QuickDraw 3D RAVE and all registered drawing engines with their associated devices comprise the **QuickDraw 3D Acceleration Layer**. As you can see, this

layer operates as a hardware abstraction layer that insulates the system software (for instance, QuickDraw 3D) or other clients from the actual video display hardware and graphics acceleration hardware available on a particular Macintosh computer.

Most applications creating 3D images should use QuickDraw 3D, which determines the best drawing engine and associated output device to use to display an image. QuickDraw 3D calls that drawing engine, using functions provided by QuickDraw 3D RAVE. As a result, most applications do not need to know about the QuickDraw 3D Acceleration Layer. Instead, they should use high-level 3D system software (such as QuickDraw 3D or OpenGL) to create and render 3D models.

Occasionally, however, some software (like the 3D game shown in Figure 23-1) needs interactive performance but only limited 3D rendering capabilities. In these cases, the software can call QuickDraw 3D RAVE functions directly, to find and configure a drawing engine and to issue drawing commands.

The QuickDraw 3D Acceleration Layer is intended to provide a basis for 3D rendering at interactive speeds. Accordingly, QuickDraw 3D RAVE is implemented in such a way as to minimize the overhead incurred by communication between an application and a drawing engine. In particular, a function call from an application to QuickDraw 3D RAVE does not require a context change. In addition, a function call from an application to a drawing engine does not require intermediate processing by QuickDraw 3D RAVE. Instead, drawing calls are implemented as C language macros that call directly into the code of a drawing engine. (See “Manipulating Draw Contexts” (page 1598) for details.)

IMPORTANT

As a result of these two features, calling a drawing engine through QuickDraw 3D RAVE provides the same level of performance as linking the engine directly with the calling application. ▲

Drawing Engines

A drawing engine is a plug-in software module that accepts drawing commands and produces a rasterized image. QuickDraw 3D RAVE is designed to make it easy for you to add drawing engines to those already available. When you register a drawing engine, it thereby becomes available for use by any application or system software running on a Macintosh computer.

QuickDraw 3D RAVE expects that a drawing engine will have a certain minimum set of required features and possibly one or more optional features. Every drawing engine must provide these features:

- hidden surface removal (usually accomplished using z buffering with at least 16 bits per pixel)
- point and line drawing, with application-specifiable point and line widths
- drawing of Gouraud-shaded triangles
- drawing of bitmaps having depths of 1, 16, or 32 bits per pixel
- support for double buffering

In addition to the required features, a drawing engine may support one or more of these optional features:

- high-precision hidden surface removal (using z buffering with at least 24 bits per pixel)
- perspective-corrected hidden surface removal
- texture mapping
- triangle meshes (a memory and time optimization that allows rendered triangles to share vertices)
- transparency blending, with or without an alpha channel
- antialiasing
- z-sorted rendering of non-opaque objects
- support for OpenGL features (such as scissoring, multiple blending modes, area and line stipple patterns, and so forth)

The interactive renderer supplied as part of QuickDraw 3D uses a software-only drawing engine that can draw to any available device. In addition to the required features listed earlier, the drawing engine supplied with the interactive renderer supports these optional features:

- z buffering with 16 or 32 bits per pixel
- direct rendering at 16 or 32 bits per pixel (rendering at fewer than 16 bits per pixel is also supported, but with lower performance)
- perspective-corrected texture mapping

It's important to keep in mind that a drawing engine is a low-level 3D driver and hence does not support some features found in higher-level interfaces. The current programming interfaces to drawing engines do not support any of these features:

- transformations, shading, or clipping
- I/O support (such as reading and writing 3D metafiles)
- high-level primitives (such as curved surfaces)
- support for drawing to windows that straddle two or more devices

IMPORTANT

Because of these limitations, most applications should not use QuickDraw 3D RAVE directly. Instead, you should use the high-level programming interfaces provided by QuickDraw 3D or other system software that provides 3D capabilities. ▲

QuickDraw 3D RAVE does not require that a drawing engine be capable of drawing to all devices available on a particular computer. Rather, a particular drawing engine may support only a single output device. For example, a drawing engine that uses a frame buffer's built-in 3D acceleration hardware may be incapable of rendering to any other device. As a result, QuickDraw 3D RAVE won't allow some other device to be associated with that drawing engine. This means that QuickDraw 3D RAVE does not provide automatic support for drawing into windows that cross multiple devices. Instead, it is the application's responsibility to determine when a window does straddle devices and to construct multiple draw contexts (described next) for the output image.

Draw Contexts

Although a drawing engine may be capable of supporting more than one device, it cannot divide a raster across multiple devices. Instead, every drawing command sent to a drawing engine must be destined for a single device. QuickDraw 3D RAVE guarantees this by requiring a calling application to specify a draw context as a parameter for every drawing command. A **draw context** is a structure (of type `TQADrawContext`) that maintains state information and other data associated with a particular drawing engine and device.

As mentioned at the end of the previous section, you need to create several draw contexts if you want to draw into a window that spans several devices.

Similarly, you need to create several draw contexts if you want to draw into several different windows on the same device. Each draw context maintains its own state information image buffers and is unaffected by any functions that operate on another draw context.

The state information associated with a draw context is maintained using a large number of **state variables.** For example, the background color of a draw context is specified by four state variables, designated by the four identifiers (or **tags**) `kQATag_ColorBG_a`, `kQATag_ColorBG_r`, `kQATag_ColorBG_g`, and `kQATag_ColorBG_b`. See "Creating and Configuring a Draw Context," beginning on page 1517, for some sample code that reads and sets state variables, and "Tags for State Variables," beginning on page 1539, for a complete list of the available state variables.

A hardware device (such as a frame buffer or a video interface) is represented in QuickDraw 3D RAVE by a **virtual device,** a structure of type `TQADevice` that determines which one of a variety of types of hardware devices a draw context draws into. On Macintosh computers, QuickDraw 3D RAVE supports two kinds of virtual devices: memory devices and graphics devices. A **memory device** represents an area of memory, and a **graphics device** represents a video device (such as a plug-in video card or built-in video interface) that controls a screen, or an offscreen graphics world (which allows your application to build complex images off the screen before displaying them). In effect, a virtual device specifies the buffers into which all drawing commands associated with a draw context write pixels.

Using QuickDraw 3D RAVE

This section illustrates how to use QuickDraw 3D RAVE. In particular, it provides source code examples that show how you can

- specify a virtual device
- determine which drawing engines are available and what features they have
- create and configure a draw context
- draw objects in a draw context
- use a draw context as an image cache
- use a texture map's alpha channel for transparency or as a blend matte

- render with antialiasing

These are examples of operations that an application might need to perform. To learn how to write and register a new drawing engine, see the section "Writing a Drawing Engine," beginning on page 1524.

Note
The code examples shown in this section provide only very rudimentary error handling. ◆

Specifying a Virtual Device

You send all drawing commands to a draw context. To create a draw context, you need to specify a virtual device and a drawing engine. This section shows how to initialize a virtual device. See the next section for information on specifying a drawing engine.

On Macintosh computers, a virtual device represents either an area of memory, a video device, or an offscreen graphics world. You specify a virtual device by filling in fields of a **device structure,** defined by the `TQADevice` data type.

```
typedef struct TQADevice {
    TQADeviceType                   deviceType;
    TQAPlatformDevice               device;
} TQADevice;
```

The `deviceType` field indicates the type of virtual device you want to draw into. Currently, you can pass either `kQADeviceMemory` or `kQADeviceGDevice` to select a Macintosh device type. The `device` field indicates a platform device data structure, which is either of type `TQADeviceMemory` for memory devices or `GDHandle` for graphics devices.

```
typedef union TQAPlatformDevice {
    TQADeviceMemory                 memoryDevice;
    GDHandle                        gDevice;
} TQAPlatformDevice;
```

To specify a memory device, you fill in the fields of a **memory device structure,** defined by the `TQADeviceMemory` data type.

```
typedef struct TQADeviceMemory {
    long                                rowBytes;
    TQAImagePixelType                   pixelType;
    long                                width;
    long                                height;
    void                                *baseAddr;
} TQADeviceMemory;
```

Listing 23-1 shows how to initialize a memory device.

Listing 23-1 Initializing a memory device

```
TQADevice       myDevice;
long            myTargetMemory[100][100];

myDevice.deviceType = kQADeviceMemory;
myDevice.device.memoryDevice.rowBytes = 100 * sizeof(long);
myDevice.device.memoryDevice.pixelType = kQAPixel_ARGB32;
myDevice.device.memoryDevice.width = 100;
myDevice.device.memoryDevice.height = 100;
myDevice.device.memoryDevice.baseAddr = myTargetMemory;
```

Drawing to memory always occurs in the native pixel format of the platform. Note that not all drawing engines support drawing to memory. For information on determining what kinds of virtual devices a particular drawing engine supports, see “Finding a Drawing Engine” (page 1516).

Listing 23-2 shows how to initialize a virtual graphics device.

Listing 23-2 Initializing a graphics device

```
TQADevice       myDevice;
GDHandle        gDeviceHandle;

/*create a GDHandle (perhaps by calling NewGDevice)*/
...
myDevice.deviceType = kQADeviceGDevice;
myDevice.device.gDevice = gDeviceHandle;
```

The code in Listing 23-2 assumes that the `gDeviceHandle` global variable has been assigned a handle to a `GDevice` record. See *Inside Macintosh: Imaging With QuickDraw* for complete information on creating and configuring graphics devices.

Note
A draw context can be associated with only a single virtual device and hence with only a single `GDevice`. Macintosh windows can straddle several screens, each associated with a different `GDevice`. It is your responsibility to determine which graphics devices a window straddles and to create a separate draw context for each one. ◆

Finding a Drawing Engine

Not all drawing engines are capable of drawing into all type of virtual devices. For example, some drawing engines might not support memory devices at all, and other drawing engines might support only a particular graphics device. As a result, once you've initialized a virtual device, you need to find a drawing engine that is capable of drawing into that device. You do this by finding the available drawing engines and selecting one that is capable of drawing into the desired virtual device. If more than one engine supports that device, you need to choose one of them.

QuickDraw 3D versions 1.1 and later provide a control panel that allows the user to select the drawing engine to use for each available monitor.

You can search through the list of available drawing engines by calling the `QADeviceGetFirstEngine` and `QADeviceGetNextEngine` functions. The `QADeviceGetFirstEngine` function returns the preferred drawing engine for the specified device; in most cases, this engine is the best engine to use for high performance rendering. However, you might need specific drawing features that are not supported by the preferred drawing engine. If so, you can use the `QAEngineGestalt` function to query the engine's capabilities, as shown in Listing 23-3.

Listing 23-3 Finding a drawing engine with fast texture mapping

```
TQAEngine *MyFindPreferredEngine (TQADevice *device)
{
    TQAEngine               *myEngine;
    unsigned long           fast;

    for (myEngine = QADeviceGetFirstEngine(device);
            myEngine;
            myEngine = QADeviceGetNextEngine(device, myEngine)) {
        if (QAEngineGestalt(myEngine, kQAGestalt_FastFeatures, &fast) == kQANoErr) {
            if (fast & kQAFast_Texture)
                return(myEngine);
        }
    }
    return(NULL);
}
```

The `MyFindPreferredEngine` function defined in Listing 23-3 calls the `QADeviceGetFirstEngine` function to get the preferred drawing engine for the specified device. Then it calls `QAEngineGestalt`, passing the `kQAGestalt_FastFeatures` selector, to determine which (if any) features are accelerated by that engine. If the engine supports accelerated texture mapping, the `MyFindPreferredEngine` function returns that drawing engine. Otherwise, the `MyFindPreferredEngine` function loops through all engines capable of drawing into the specified device until it finds one that does support fast texture mapping. If none is found, `MyFindPreferredEngine` returns the value `NULL`.

Note
See “Gestalt Selectors” (page 1559) for a complete description of the selectors you can pass to the `QAEngineGestalt` function. ◆

Creating and Configuring a Draw Context

Once you’ve initialized a virtual device and selected a drawing engine capable of drawing to that device, you can call the `QADrawContextNew` function to create a new draw context. You pass the device and engine to that function, along with a drawing rectangle, a clipping region, and a set of draw context flags. The flags

specify features of the new draw context. Listing 23-4 illustrates how to create a double-buffered draw context with z buffering.

Listing 23-4 Creating a draw context

```
TQADrawContext          *myDrawContext;

if (QADrawContextNew(&myDevice, &myRect, &myClip, myEngine,
            kQAContext_DoubleBuffer, &myDrawContext) != kQANoErr) {
    /*Error! Could not create new draw context.*/
}
```

If `QADrawContextNew` succeeds, it returns the result code `kQANoErr` and sets the `myDrawContext` parameter to the new draw context. Otherwise, if an error occurs, `QADrawContextNew` returns some other result code and sets the `myDrawContext` parameter to the value `NULL`.

Note
When you are finished using the new draw context, you should free the memory and other resources it uses by calling the `QADrawContextDelete` function. ◆

QuickDraw 3D RAVE does not provide a function to reposition an existing draw context. If a window associated with a draw context is moved on the screen, you need to delete the existing draw context and create a new draw context at the new location. Similarly, QuickDraw 3D RAVE does not provide a function to change the clipping region of a draw context. If you want to change a clipping region, you need to delete the existing draw context and create a new draw context with the desired clipping region.

However, you can change a number of other features of a draw context without having to delete an existing draw context and create a new one. The features you can change are indicated by the state variables of the draw context. For example, to change the background color of a draw context to opaque black, you can use the code shown in Listing 23-5.

Listing 23-5 Setting a draw context state variable

```
void MySetBackgroundToBlack (TQADrawContext *drawContext);
{
    QASetFloat(drawContext, kQATag_ColorBG_a, 1.0);
    QASetFloat(drawContext, kQATag_ColorBG_r, 0.0);
    QASetFloat(drawContext, kQATag_ColorBG_g, 0.0);
    QASetFloat(drawContext, kQATag_ColorBG_b, 0.0);
}
```

The `QASetFloat` function sets a draw context state variable that has a floating-point value. QuickDraw 3D RAVE provides functions to get and set state variables with floating-point, long integer, or pointer values.

Note
See "Tags for State Variables," beginning on page 1539, for a complete description of the available draw context state variables. ◆

The `QASetFloat` function is defined using a C language macro:

```
#define QASetFloat(drawContext,tag,newValue) \
                (drawContext)->setFloat (drawContext,tag,newValue)
```

During compilation, the `QASetFloat` call is replaced by code that directly calls the drawing engine's floating-point setting method. This allows you to achieve the highest possible performance when configuring a draw context.

Drawing in a Draw Context

QuickDraw 3D RAVE allows you to draw five kinds of objects in a draw context: points, lines, triangles, triangle meshes, and bitmaps. You draw by calling a function to draw the desired type of object. For instance, to draw a single point, you can call the `QADrawPoint` function, as follows:

```
QADrawPoint(myDrawContext, myPoint);
```

Here, the `myPoint` parameter specifies the point to draw. All objects that a drawing engine can draw (except for bitmaps) are defined by points or vertices. QuickDraw 3D RAVE supports two different types of vertices: Gouraud vertices

and texture vertices. You use **Gouraud vertices** for drawing Gouraud-shaded triangles, and also for drawing points and lines. A Gouraud vertex is defined by the `TQAVGouraud` data structure, which specifies the position, depth, color, and transparency information.

You use **texture vertices** to define triangles to which a texture is to be mapped. A texture vertex is defined by the `TQAVTexture` data structure, which specifies the position, depth, transparency, and texture mapping information.

IMPORTANT

QuickDraw 3D RAVE does not currently support clipping to a draw context. All triangles and other objects drawn to a draw context must lie entirely within the draw context. ◆

Using a Draw Context as a Cache

QuickDraw 3D RAVE supports draw context caching, a technique that allows you to improve rendering performance when a large number of the objects in a scene don't change from frame to frame. A **draw context cache** is simply a draw context that contains an image and is designated as the initial context in a call to `QARenderStart`. The contents of that context are drawn into the destination draw context before any other objects.

To create a draw context cache, you first create a draw context by calling the `QADrawContextNew` function, where the `flags` parameter has the `QAContext_Cache` flag set. Then you draw the unchanging objects into the draw context cache. For example, suppose that you want to draw a series of frames in which two triangles remain constant from frame to frame but a third triangle changes every frame. Listing 23-6 shows how to do this.

Listing 23-6 Creating and using a draw context cache

```
TQAVGouraud                tri1[3], tri2[3], tri3[3];
TQADrawContext             *myCache, *myDest;

/*Create draw context cache and destination draw context.*/
QADrawContextNew(myDev, rect, NULL, myEng, QAContext_Cache, &myCache);
QADrawContextNew(myDev, rect, NULL, myEng, QAContext_DoubleBuffer, &myDest);

/*Set up the image in the cache context.*/
```

```
QARenderStart(myCache, NULL, NULL);
QADrawTriGouraud(myCache, &tri1[0], &tri1[1], &tri1[2], kQATriFlags_None);
QADrawTriGouraud(myCache, &tri2[0], &tri2[1], &tri2[2], kQATriFlags_None);
QARenderEnd(myCache, NULL);

/*Render frames using the cache and moving tri3 only.*/
while (gStillMovingTriangle3) {
    MyMoveTri(tri3);
    QARenderStart(myDest, NULL, myCache);
    QADrawTriGouraud(myDest, &tri3[0], &tri3[1], &tri3[2], kQATriFlags_None);
    QARenderEnd(myDest, NULL);
}
```

Not all drawing engines support draw context caching. If a drawing engine does not support caching, it should return the value `NULL` whenever you pass the `QAContext_Cache` flag to `QADrawContextNew`.

IMPORTANT

All draw context caches must be single buffered, and they must be created using the same device and rectangle as the destination draw contexts with which they will be used. ▲

Once you've created a draw context cache (by setting the `QAContext_Cache` flag when calling `QADrawContextNew`), you cannot use that draw context as a non-cached draw context. Objects rendered into a draw context cache never appear on a device (not even on a memory device). The only way to view objects rendered into a draw context cache is to use that cache to initialize a non-cached draw context, as illustrated in Listing 23-6. You can, however, use a draw context cache to initialize another draw context cache. Moreover, you can initialize a draw context cache with itself in order to add more objects to an existing draw context cache.

Using a Texture Map Alpha Channel

Texture maps whose pixel type is either `kQAPixel_ARGB16` or `kQAPixel_ARGB32` contain an alpha channel value for each pixel in the map. You can use the alpha channel value to control the transparency of an object on a pixel-by-pixel basis, or you can use the alpha channel value as a blend matte that exposes only certain portions of an image.

To use the alpha channel to control transparency, you should set the drawing engine's transparency blending mode to `kQABlend_Premultiply`. (You specify an engine's transparency blending mode by assigning a value to its `kQATag_Blend` state variable.) For pixels of type `kQAPixel_ARGB16`, the alpha channel value occupies bit 15; when the value is 1, the pixel is opaque; when the value is 0, the pixel is completely transparent. For pixels of type `kQAPixel_ARGB32`, the alpha channel value occupies bits 31 through 24; when the value is 255, the pixel is opaque; when the value is 0, the pixel is completely transparent.

IMPORTANT

The `kQABlend_Premultiply` transparency model assumes that the diffuse color of a pixel has been premultiplied by the alpha channel value. As a result, every pixel of the texture map must be premultiplied by its associated alpha channel value before you create the texture map by calling `QATextureNew`. ▲

Note that the specular highlight is unaffected by the diffuse transparency of an object. As a result, setting an object's alpha channel value to 0 when using the `kQABlend_Premultiply` transparency blending mode does not cause the object to vanish. The specular highlight is still rendered.

To use the alpha channel as a **blend matte** to cut out certain portions of a rendered object, you should set the drawing engine's transparency blending mode to `kQABlend_Interpolate`. If the alpha channel value of all pixels in an object is 0, neither the object nor its specular highlight is rendered. This effectively eliminates the object from the rendered image.

IMPORTANT

The `kQABlend_Interpolate` transparency model assumes that the diffuse color of a pixel has *not* been premultiplied by the alpha channel value. This multiplication is performed by the blending operation. ▲

The `kQABlend_Interpolate` blending mode cannot render a transparent surface as accurately as the `kQABlend_Premultiply` mode, because the specular highlight is scaled by the alpha value. In some cases, you can compensate for this behavior by increasing the brightness of the specular highlight.

Rendering With Antialiasing

A drawing engine may support an **antialiasing mode** that determines the kind of antialiasing applied to a drawing context. (**Antialiasing** is the smoothing of jagged edges on a displayed shape by modifying the transparencies of individual pixels along the shape's edge.)

Note
The antialiasing mode state variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_Antialias` feature. ◆

You specify an engine's antialiasing mode by assigning a value to its `kQATag_Antialias` state variable. QuickDraw 3D RAVE provides these constants for antialiasing modes:

```
#define kQAAntiAlias_Off                        0
#define kQAAntiAlias_Fast                       1
#define kQAAntiAlias_Mid                        2
#define kQAAntiAlias_Best                       3
```

The interpretation of these values is specific to each drawing engine. For example, a drawing engine might be able to support antialiased line drawing with no performance penalty but that same engine might incur a 50 percent slowdown when drawing antialiased triangles. Accordingly, this engine might interpret the `kQAAntiAlias_Fast` antialiasing mode as rendering antialiased lines only, and it might interpret the `kQAAntiAlias_Mid` mode as rendering both antialiased lines and triangles.

Note
QuickDraw 3D RAVE interprets antialiasing modes independently of the transparency blending modes, unlike some other rendering technologies. For instance, with OpenGL you must select specific blending modes when antialiasing is enabled. ◆

Writing a Drawing Engine

This section shows how to write a new drawing engine and add it to the QuickDraw 3D Acceleration Layer.

IMPORTANT

You need to read this section only if you are developing custom 3D acceleration hardware or software. If you simply want to create a draw context and draw into it using low-level drawing functions, see "Using QuickDraw 3D RAVE," beginning on page 1513. ▲

To develop a new drawing engine and add it to the QuickDraw 3D Acceleration Layer, you need to perform these seven steps:

1. Write methods for the public functions pointed to by the fields of a draw context structure (for example, `setInt`). These methods are described in detail in the section "Public Draw Context Methods," beginning on page 1618.
2. Write methods for the `TQADrawPrivateNew` and `TQADrawPrivateDelete` function prototypes. These functions are called internally by the `QADrawContextNew` and `QADrawContextDelete` functions, respectively. You use these methods to allocate and release any private data (such as state variables) maintained by your drawing engine. These methods are described in detail in the section "Private Draw Context Methods," beginning on page 1639.
3. Write methods for any texture and bitmap functions supported by your drawing engine (`TQATextureNew`, `TQATextureDetach`, `TQATextureDelete`, `TQABitmapNew`, `TQABitmapDetach`, and `TQABitmapDelete`). These functions are called by their public counterparts (for example, `QABitmapNew`). These methods are described in detail in the section "Texture and Bitmap Methods," beginning on page 1644.
4. Write a method to handle the `QAEngineGestalt` function when your drawing engine is the target engine. This method is described in detail on page 1642.
5. Write a method to handle the `QAEngineCheckDevice` function when your drawing engine is the target engine. QuickDraw 3D RAVE calls this method to determine which devices your drawing engine supports. This method is described in detail on page 1641.

6. Write a method for the `TQAEngineGetMethod` function prototype. QuickDraw 3D RAVE calls this method to get some of your engine's methods during engine registration. This method is described in detail on page 1650.
7. Build your code as a shared library. The initialization routine of the shared library should register your drawing engine with QuickDraw 3D RAVE by calling the `QARegisterEngine` function.

The following sections describe some of these steps in more detail. The section "Supporting OpenGL Hardware," beginning on page 1531 contains information that is useful if you are implementing a drawing engine to support hardware that is based on an OpenGL rasterization model.

Writing Public Draw Context Methods

As you've seen, the draw context structure (of type `TQADrawContext`) contains function pointers to the public draw context methods supported by your drawing engine. These methods are called whenever an application calls one of the public functions provided by QuickDraw 3D RAVE. For example, when an application calls the `QADrawPoint` function for a draw context associated with your drawing engine, your engine's `TQADrawPoint` method (pointed to by the `drawPoint` field) is called. The `TQADrawPoint` method is declared like this:

```
typedef void (*TQADrawPoint) (
                    const TQADrawContext *drawContext,
                    const TQAVGouraud *v);
```

A draw context structure is passed as the first parameter to all the public draw context methods you need to define. This allows your methods to find the private data associated with the draw context (which is pointed to by the `drawPrivate` field).

Notice that the function prototype for a point-drawing method passes the draw context as a `const` parameter. This indicates that your method should not alter any of the fields of the draw context structure passed to it. Only three draw context methods (namely `TQASetInt`, `TQASetFloat`, and `TQASetPtr`) are allowed to alter the draw context.

Listing 23-7 shows a sample definition for a point-drawing method.

Listing 23-7 A `TQADrawPoint` method

```
void MyDrawPoint (const TQADrawContext *drawContext, const TQAVGouraud *v)
{
    MyPrivateData               *myData;    /*our actual private data type*/

    /*Cast generic drawPrivate pointer to our actual private data type.*/
    myData = (MyPrivateData *) drawContext->drawPrivate;

    /*Call our z-buffered pixel drawing function with xyz and argb, and
      also pass it the current zfunction, which is stored in the private draw
      context data structure. Note that this isn't a complete implementation!
      (We should be using kQATag_Width, for example.)*/

    MyDrawPixelWithZ(v->x, v->y, v->z, v->a, v->r, v->g, v->b,
                                        myData->stateVariable[kQATag_ZFunction]);
}
```

Note
See "Public Draw Context Methods," beginning on page 1618 for complete information on the public draw context methods your drawing engine must define. ◆

Once you've defined the necessary public draw context methods, you need to insert pointers to those methods into a draw context structure. You accomplish this step in your `TQADrawPrivateNew` method, described in the next section.

Writing Private Draw Context Methods

Once you've written the public draw context methods supported by your drawing engine, you need to write several private draw context methods. In particular, you need to write a `TQADrawPrivateNew` method to initialize a draw context and a `TQADrawPrivateDelete` method to delete a draw context. The `TQADrawPrivateNew` method is called whenever an application creates a new draw context by calling the `QADrawContextNew` function. Listing 23-8 illustrates a sample `TQADrawPrivateNew` method.

Listing 23-8 A `TQADrawPrivateNew` method

```
TQAError MyDrawPrivateNew (
    TQADrawContext      *drawContext,
    const TQADevice     *device,
    const TQARect       *rect,
    const TQAClip       *clip,
    unsigned long       flags)
{
    MyPrivateData       *myData;

    /*Allocate a new MyPrivateData structure and store it in draw context.*/
    myData = MyDataNew(...);
    drawContext->drawPrivate = (TQADrawPrivate *) myData;
    if (!myData)
        return (kQAOutOfMemory);

    /*Set the method pointers of drawContext to point to our draw methods.*/
    newDrawContext->setFloat = MySetFloat;
    newDrawContext->setInt = MySetInt;
    ...
    return(kQANoErr);
}
```

As you can see, the `MyDrawPrivateNew` function defined in Listing 23-8 allocates space for its private data, installs a pointer to that data in the `drawPrivate` field of the draw context structure, and then installs pointers to all the public draw context methods supported by the drawing engine into the draw context structure.

Your `TQADrawPrivateDelete` method should simply undo any work done by your `TQADrawPrivateNew` method. In this case, the delete method just needs to release the private storage allocated by the `TQADrawPrivateNew` method. Listing 23-9 shows a sample `TQADrawPrivateDelete` method.

Listing 23-9 A `TQADrawPrivateDelete` method

```
void MyDrawPrivateDelete (TQADrawPrivate *drawPrivate)
{
    MyDataDelete((MyPrivateData *) drawPrivate);
}
```

You register your private draw context methods with QuickDraw 3D RAVE using another private method, the `TQAEngineGetMethod` method. See "Registering a Drawing Engine," beginning on page 1529 for details.

Handling Gestalt Selectors

To support calls to the public function `QAEngineGestalt`, your drawing engine must define a `TQAEngineGestalt` method. This method returns information about the capabilities of your drawing engine. For example, suppose that your drawing engine supports texture mapping and accelerates both Gouraud shading and line drawing. Suppose further that you have been assigned a vendor ID of 5, and that the engine ID of your engine is 1001. In that case, you could define a method like the one shown in Listing 23-10.

Listing 23-10 A `TQAEngineGestalt` method

```
TQAError MyEngineGestalt (TQAGestaltSelector selector, void *response)
{
    const static char       *myEngineName = "SurfDraw 3D";

    switch (selector) {
        case kQAGestalt_OptionalFeatures:
            *((unsigned long *) response) = kQAOptional_Texture;
            break;
        case kQAGestalt_FastFeatures:
            *((unsigned long *) response) = kQAFast_Line | kQAFast_Gouraud;
            break;
        case kQAGestalt_VendorID:
            *((long *) response) = 5;
            break;
        case kQAGestalt_EngineID:
            *((long *) response) = 1001;
```

```
            break;
        case kQAGestalt_Revision:
            *((long *) response) = 0;
            break;
        case kQAGestalt_ASCIINameLength:
            *((long *) response) = strlen(myEngineName);
            break;
        case kQAGestalt_ASCIIName:
            strcpy(response, myEngineName);
            break;
        default:                        /*must flag unrecognized selectors*/
            return (kQAParamErr);
    }
    return (kQANoErr);
}
```

If two different drawing engines should return identical vendor and engine IDs, QuickDraw 3D RAVE chooses the one that returns the most recent revision number (that is, the value returned for the `kQAGestalt_Revision` selector). The larger number is considered newer.

You register your `TQAEngineGestalt` method with QuickDraw 3D RAVE using the `TQAEngineGetMethod` method, described in the next section.

Registering a Drawing Engine

Once you written all the necessary public and private draw context methods, as well as methods to handle textures and bitmaps, you must write a `TQAEngineGetMethod` method that reports the addresses of some of those methods to QuickDraw 3D RAVE. Listing 23-11 shows a sample `TQAEngineGetMethod` method. Notice that this method returns the addresses only of the private draw context methods and the methods to handle textures and bitmaps. The pointers for the public draw context methods are assigned directly to the fields of a draw context structure by your `TQADrawPrivateNew` method (as shown in Listing 23-8).

Listing 23-11 A `TQAEngineGetMethod` method

```
TQAError MyEngineGetMethod (TQAEngineMethodTag methodTag, TQAEngineMethod *method)
{
    switch (methodTag) {
        case kQADrawPrivateNew:
            method->drawPrivateNew = MyDrawPrivateNew;
            break;
        case kQADrawPrivateDelete:
            method->drawPrivateDelete = MyDrawPrivateDelete;
            break;
        case kQAEngineCheckDevice:
            method->engineCheckDevice = MyEngineCheckDevice;
            break;
        case kQAEngineGestalt:
            method->engineGestalt = MyEngineGestalt;
            break;
        case kQABitmapNew:
            method->bitmapNew = MyBitmapNew;
            break;
        case kQABitmapDetach:
            method->bitmapDetach = MyBitmapDetach;
            break;
        case kQABitmapDelete:
            method->bitmapDelete = MyBitmapDelete;
            break;
        default:
            return(kQANotSupported);
    }
    return(kQANoErr);
}
```

Finally, you register your drawing engine by passing the address of your `TQAEngineGetMethod` method to the `QARegisterEngine` function:

```
QARegisterEngine(&MyEngineGetMethod);
```

You can call `QARegisterEngine` in two ways. During product development, you can link your drawing engine code directly with a test application, in which case you should call `QARegisterEngine` from your application's initialization code. Alternatively, once you've completed development, you should build

your engine's code into a shared library of type `'tnsl'`. In this case, you should call `QARegisterEngine` from the initialization routine of the shared library. When the shared library containing QuickDraw 3D RAVE is loaded, it searches for and loads any drawing engines contained in shared libraries in the current folder or in the Extensions folder.

Supporting OpenGL Hardware

This section contains information that is useful if you are implementing a drawing engine to support hardware that is based on an OpenGL rasterization model. It describes special considerations for handling transparency and texture mapping.

Transparency

QuickDraw 3D RAVE supports three transparency models: the premultiplied, interpolated, and OpenGL transparency models. Support for the OpenGL transparency model (indicated by the `kQABlend_OpenGL` constant) should be automatic for hardware that is based on the OpenGL rasterization model. The other two models, indicated by the `kQABlend_PreMultiply` and `kQABlend_Interpolate` constants) may require emulation by your drawing engine.

For example, consider the premultiplied blending function, specified by these equations:

$$
\begin{aligned}
a &= 1 - ((1 - a_s) \times (1 - a_d)) \\
r &= r_s + ((1 - a_s) \times r_d) \\
g &= g_s + ((1 - a_s) \times g_d) \\
b &= b_s + ((1 - a_s) \times b_d)
\end{aligned}
$$

(Here, the factors a_s, r_s, g_s, and b_s represent the alpha, red, green and blue components of a source pixel; the factors a_d, r_d, g_d, and b_d represent the alpha, red, green and blue components of a destination pixel.)

Note
A complete description of how transparent objects are blended together with each of these models is provided in "Blending Operations" (page 1550). ◆

OpenGL directly supports the premultiplied transparency blending function (and the interpolated transparency blending function) for the RGB components only. In other words, the alpha channel component (which is the same for both blending operations) cannot be directly implemented in OpenGL-compliant hardware. It is possible, however, to emulate these two transparency modes on OpenGL hardware, using several different methods. You can blend the RGB values only, or you can blend the ARGB values using a multipass algorithm. Which of these emulations you use depends on whether your drawing engine is associated with a frame buffer that stores an alpha channel or not.

If your drawing engine is associated with a frame buffer that doesn't store an alpha channel value, you can implement the premultiplied and interpolated blending functions by simply ignoring the alpha channel component. These functions are then equivalent to OpenGL blending modes. The premulitplied blending function, with its alpha channel ignored, can be emulated by this function:

```
gBlendFunc(GL_ONE, GL_ONE_MINUS_SRC_ALPHA);
```

Similarly, the interpolated blending function, with its alpha channel ignored, can be emulated by this function:

```
gBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

IMPORTANT

A drawing engine that uses this method of emulating the QuickDraw 3D RAVE blending functions on OpenGL hardware should not set the `kQAOptional_BlendAlpha` flag of the `kQAGestalt_OptionalFeatures` selector to the `QAEngineGestalt` function. ▲

To achieve a more complete blending, you can have your drawing engine rasterize each transparent object more than once, altering in each pass the blending mode, object alpha channel, and buffer write masks. The first pass should perform RGB blending. Accordingly, you should disable writing any alpha channel or z buffer data during this pass.

```
/*first pass*/
glColorMask(TRUE, TRUE, TRUE, FALSE);          /*disable alpha channel*/
glDepthMask(FALSE);                            /*disable Z buffer*/
if (premultpliedTransparency)
    glBlendFunc(GL_ONE, GL_ONE_MINUS_SRC_ALPHA);
```

```
else
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
/*render the object here*/
```

On the second pass, you should set the frame buffer alpha channel value to $(1–a_s)\times(1–a_d)$. To do this, you need to render the object again, with a different alpha value, as follows:

```
/*second pass*/
glColorMask(FALSE, FALSE, FALSE, TRUE);         /*enable alpha channel*/
glDepthMask(FALSE);                             /*disable Z buffer*/
glBlendFunc(GL_ONE_MINUS_DST_ALPHA, GL_ZERO);
/*render the object with alpha replaced with 1-a*/
```

Finally, the third pass should replace the value in the alpha channel with the final value $1–((1–a_s)\times(1–a_d))$. To do this, you need to render the object again, with its alpha value set to 1, as follows:

```
/*third pass*/
glColorMask(FALSE, FALSE, FALSE, TRUE);         /*enable alpha channel*/
glDepthMask(TRUE);                              /*enable Z buffer*/
glBlendFunc(GL_ONE_MINUS_DST_ALPHA, GL_ZERO);
/*render the object with alpha replaced with 1*/
```

After the third pass, the frame buffer contains the correctly blended object.

Texture Mapping

QuickDraw 3D RAVE supports several texture mapping operations, which are controlled by the flags in the `kQATag_TextureOp` state variable. Currently these flags are defined:

```
#define kQATextureOp_Modulate                       (1 << 0)
#define kQATextureOp_Highlight                      (1 << 1)
#define kQATextureOp_Decal                          (1 << 2)
#define kQATextureOp_Shrink                         (1 << 3)
```

Note
A complete description of texture mapping operations is provided in “Texture Operations” (page 1553). ◆

To support the `kQATextureOp_Modulate` mode on an OpenGL-compliant rasterizer, you can use the `GL_MODULATE` mode, where the `kd_r`, `kd_g`, and `kd_b` fields of a texture vertex specify the modulating color. Note, however, that `GL_MODULATE` does not allow these color values to be greater than 1.0, whereas QuickDraw 3D RAVE does allow them to be greater than 1.0. Values greater than 1.0 can provide improved image realism, and new hardware should support them. A more reasonable maximum modulation amplitude is 2.0.

You can support the `kQATextureOp_Highlight` mode by performing two rendering passes. The first pass should render the texture-mapped object (possibly also with modulation, as just described), and the second pass should add the specular highlight value.

```
/*first pass*/
glDepthMask(FALSE);                             /*disable Z buffer*/
/*render the texture-mapped object here*/

/*second pass*/
glDepthMask(TRUE);                              /*enable Z buffer*/
glBlendFunc(GL_ONE, GL_ONE);                    /*add highlight color*/
/*render the highlight color as a Gouraud-shaded object here*/
```

On the second pass, you should render the highlight color, using the `ks_r`, `ks_g`, and `ks_b` fields of a texture vertex, as a Gouraud-shaded object.

If the `kQATextureOp_Modulate` flag is clear (that is, is no texture map color modulation is to be performed), you can support the `kQATextureOp_Decal` mode using the OpenGL `GL_DECAL` mode. If, in addition, the `kQATextureOp_Highlight` flag is set, you need to perform two rendering passes, as just described.

IMPORTANT

There is currently no known method of accurately rendering to OpenGL-compliant hardware when *both* the `kQATextureOp_Decal` and the `kQATextureOp_Modulate` flags are set. You should determine the best method of implementing this mode correctly on your hardware. If your hardware cannot handle both modes at once, you should ignore the `kQATextureOp_Modulate` mode whenever `kQATextureOp_Decal` is set. ▲

QuickDraw 3D RAVE Reference

This section describes the constants, data structures, and routines provided by QuickDraw 3D RAVE. It also describes the functions you must define in order to write a drawing engine.

The application programming interfaces of QuickDraw 3D RAVE follow these simple naming conventions:

- All names of constants begin with the prefix `kQA` (for example, `kQATextureFilter_Fast`).
- All names of data types begin with the prefix `TQA` (for example, `TQADrawContext`).
- All names of functions begin with the prefix `QA` (for example, `QADrawContextNew`).

Constants

This section describes the constants provided by QuickDraw 3D RAVE.

Version Values

The `version` field of a draw context structure (of type `TQADrawContext`) specifies the current version of QuickDraw 3D RAVE. This field contains one of these constants:

```
typedef enum TQAVersion {
    kQAVersion_Prerelease               = 0,
    kQAVersion_1_0                      = 1,
    kQAVersion_1_0_5                    = 2,
    kQAVersion_1_1                      = 3
} TQAVersion;
```

Constant descriptions

`kQAVersion_Prerelease`
A prerelease version.

`kQAVersion_1_0`	Version 1.0. This is the version that supports the interactive renderer included with QuickDraw 3D version 1.0.
`kQAVersion_1_0_5`	Version 1.0.5. This version supports triangle meshes and color lookup tables.
`kQAVersion_1_1`	Version 1.1. This version supports notice methods, texture compression flags, and the `kQAGestalt_AvailableTexMem` selector for the `QAEngineGestalt` function.

Pixel Types

The `pixelType` field of a memory device structure (of type `TQADeviceMemory`) specifies a pixel format (that is, the size and organization of the memory associated with a single pixel in a memory pixmap). You use these constants to assign a value to that field and also to parameters to the `QATextureNew` and `QABitmapNew` functions.

```
typedef enum TQAImagePixelType {
    kQAPixel_Alpha1                 = 0,
    kQAPixel_RGB16                  = 1,
    kQAPixel_ARGB16                 = 2,
    kQAPixel_RGB32                  = 3,
    kQAPixel_ARGB32                 = 4,
    kQAPixel_CL4                    = 5,
    kQAPixel_CL8                    = 6,
    kQAPixel_RGB16_565              = 7,
    kQAPixel_RGB24                  = 8
} TQAImagePixelType;
```

Constant descriptions

`kQAPixel_Alpha1`	A pixel occupies 1 bit of memory, which is interpreted as an alpha channel value. This value is relevant only for the `QABitmapNew` function. When a bit is 1, it is opaque and is rendered in the color passed to the `QADrawBitmap` function; when the bit is 0, it is completely transparent.
`kQAPixel_RGB16`	A pixel occupies 16 bits of memory, with the red component in bits 14 through 10, the green component in bits 9 through 5, and the blue component in bits 4 through 0. There is no per-pixel alpha channel value. As a result, the pixmap (perhaps defining a texture) is treated as opaque.

(You can, however, apply transparency to the pixmap using the alpha channel values of a triangle vertex, for instance.)

`kQAPixel_ARGB16` A pixel occupies 16 bits of memory, with the red component in bits 14 through 10, the green component in bits 9 through 5, and the blue component in bits 4 through 0. In addition, the pixel's alpha channel value is in bit 15. When the alpha value is 1, the pixmap is opaque; when the alpha value is 0, the pixmap is completely transparent.

`kQAPixel_RGB32` A pixel occupies 32 bits of memory, with the red component in bits 23 through 16, the green component in bits 15 through 8, and the blue component in bits 7 through 0. There is no per-pixel alpha channel value. As a result, the pixmap (perhaps defining a texture) is treated as opaque. (You can, however, apply transparency to the pixmap using the alpha channel values of a triangle vertex, for instance.)

`kQAPixel_ARGB32` A pixel occupies 32 bits of memory, with the red component in bits 23 through 16, the green component in bits 15 through 8, and the blue component in bits 7 through 0. In addition, the pixel's alpha channel value is in bits 31 through 24. When the alpha value is 255, the pixmap is opaque; when the alpha value is 0, the pixmap is completely transparent.

`kQAPixel_CL4` A pixel value is an index into a 4-bit color lookup table. This color lookup table is always big-endian (that is, the high 4 bits affect the leftmost pixel). This pixel type is valid only as a parameter for the `QATextureNew` and `QABitmapNew` functions. Not all drawing engines support this pixel type; it is supported only when a drawing engine supports the `kQAOptional_CL4` feature.

`kQAPixel_CL8` A pixel value is an index into a 8-bit color lookup table. This pixel type is valid only as a parameter for the `QATextureNew` and `QABitmapNew` functions. Not all drawing engines support this pixel type; it is supported only when a drawing engine supports the `kQAOptional_CL8` feature.

`kQAPixel_RGB16_565` A pixel occupies 16 bits of memory, with the red component in bits 15 through 11, the green component in bits 10 through 5, and the blue component in bits 4 through 0. There is no per-pixel alpha channel value. This pixel type is currently defined only for Windows 32 devices.

`kQAPixel_RGB24` A pixel occupies 24 bits of memory, with the red component in bits 23 through 16, the green component in bits 15 through 8, and the blue component in bits 7 through 0. There is no per-pixel alpha channel value. This pixel type is currently defined only for Windows 32 devices.

Color Lookup Table Types

The `tableType` parameter of the `QAColorTableNew` function specifies a color lookup table type. QuickDraw 3D RAVE currently supports these types of color lookup tables:

```
typedef enum TQAColorTableType {
    kQAColorTable_CL8_RGB32             = 0,
    kQAColorTable_CL4_RGB32             = 1
} TQAColorTableType;
```

Constant descriptions

`kQAColorTable_CL8_RGB32`
The color lookup table contains 256 colors, and each color occupies 32 bits of memory, with the red component in bits 23 through 16, the green component in bits 15 through 8, and the blue component in bits 7 through 0.

`kQAColorTable_CL4_RGB32`
The color lookup table contains 16 colors, and each color occupies 32 bits of memory, with the red component in bits 23 through 16, the green component in bits 15 through 8, and the blue component in bits 7 through 0.

Device Types

The `deviceType` field of a device data structure (of type `TQADevice`) specifies a device type. You use these constants to assign a value to that field.

```
typedef enum TQADeviceType {
    kQADeviceMemory                     = 0,
    kQADeviceGDevice                    = 1,
```

```
    kQADeviceWin32DC                    = 2,
    kQADeviceDDSurface                  = 3
} TQADeviceType;
```

Constant descriptions

`kQADeviceMemory` A memory device.

`kQADeviceGDevice` A graphics device (of type `GDevice`).

`kQADeviceWin32DC` A Windows 32 device.

`kQADeviceDDSurface` A Windows direct draw surface.

Clip Types

The `clipType` field of a clip data structure (of type `TQAClip`) specifies a clip type. You use these constants to assign a value to that field.

```
typedef enum TQAClipType {
    kQAClipRgn                          = 0,
    kQAClipWin32Rgn                     = 1
} TQAClipType;
```

Constant descriptions

`kQAClipRgn` A clipping region.

`kQAClipWin32Rgn` A Windows 32 clipping region.

Tags for State Variables

A drawing engine maintains a large number of state variables that determine how the engine draws into a device. Each state variable has a state value, which is either an unsigned long integer, a floating-point value, or a pointer. You can read and write state values by calling QuickDraw 3D RAVE functions. (For instance, you can set a state value by calling `QASetInt`, `QASetFloat`, or `QASetPtr`.) You specify which state variable to get or set using a state tag, a unique identifier associated with that variable.

Note
All tag values greater than 0 and less than `kQATag_EngineSpecific_Minimum` are reserved for use by QuickDraw 3D RAVE. If you need to define engine-specific tags, you should assign them tag values greater than or equal to `kQATag_EngineSpecific_Minimum`. ◆

Here are the tags for state variables having unsigned long integer values:

```
typedef enum TQATagInt {
    kQATag_ZFunction                    = 0,    /*required variables*/
    kQATag_Antialias                    = 8,    /*optional variables*/
    kQATag_Blend                        = 9,
    kQATag_PerspectiveZ                 = 10,
    kQATag_TextureFilter                = 11,
    kQATag_TextureOp                    = 12,
    kQATag_CSGTag                       = 14,
    kQATag_CSGEquation                  = 15,
    kQATag_BufferComposite              = 16,
    kQATagGL_DrawBuffer                 = 100,  /*OpenGL variables*/
    kQATagGL_TextureWrapU               = 101,
    kQATagGL_TextureWrapV               = 102,
    kQATagGL_TextureMagFilter           = 103,
    kQATagGL_TextureMinFilter           = 104,
    kQATagGL_ScissorXMin                = 105,
    kQATagGL_ScissorYMin                = 106,
    kQATagGL_ScissorXMax                = 107,
    kQATagGL_ScissorYMax                = 108,
    kQATagGL_BlendSrc                   = 109,
    kQATagGL_BlendDst                   = 110,
    kQATagGL_LinePattern                = 111,
    kQATagGL_AreaPattern0               = 117,
    kQATagGL_AreaPattern31              = 148,
    kQATag_EngineSpecific_Minimum       = 1000
} TQATagInt;
```

Constant descriptions

`kQATag_ZFunction` The z sorting function of the drawing engine. This function determines which surfaces are to be removed during hidden surface removal. See “Z Sorting Function Selectors” (page 1548) for a description of the available z sorting

functions. The default value for a drawing engine that is z buffered is `kQAZFunction_LT`; the default value for a draw context that is not z buffered is `kQAZFunction_None`. The z sorting function state variable must be supported by all drawing engines.

`kQATag_Antialias` The antialiasing mode of the drawing engine. This mode determines how, if at all, antialiasing is applied to the draw context. See "Antialiasing Selectors" (page 1549) for a description of the available antialiasing modes. The default value for a drawing engine that supports antialiasing is `kQAAntiAlias_Fast`. The antialiasing mode state variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_Antialias` feature.

`kQATag_Blend` The transparency blending function of the drawing engine. See "Blending Operations" (page 1550) for a description of the available transparency blending functions. The default value for a drawing engine that supports blending is `kQABlend_Premultiply`. The transparency blending function state variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_Blend` feature.

`kQATag_PerspectiveZ`

The z perspective control of the drawing engine. This control determines how a drawing engine performs hidden surface removal. See "Z Perspective Selectors" (page 1551) for a description of the available z perspective controls. The default value for a drawing engine that supports z perspective is `kQAPerspectiveZ_Off`. The z perspective control state variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_PerspectiveZ` feature.

`kQATag_TextureFilter`

The texture mapping filter mode of the drawing engine. This mode determines how a drawing engine performs texture mapping. See "Texture Filter Selectors" (page 1552) for a description of the available texture mapping filter modes. The default value for a drawing engine that supports texture mapping is `kQATextureFilter_Fast`. The texture mapping filter state variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_Texture` feature.

`kQATag_TextureOp` The texture mapping operation of the drawing engine. This mode determines the current texture mapping operation of a drawing engine. See "Texture Operations" (page 1553) for a description of the available texture mapping operations. The default value for a drawing engine that supports texture mapping is `kQATextureOp_None`. The texture mapping operation variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_Texture` feature.

`kQATag_CSGTag` The CSG ID of triangles subsequently submitted to the drawing engine. See "CSG IDs" (page 1554) for a description of the available CSG IDs. The default value for a drawing engine that supports CSG operations is `kQACSGTag_None`. The CSG ID variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_CSG` feature.

`kQATag_CSGEquation` The CSG equation for the drawing engine, which determines the manner in which triangles with CSG IDs are combined into solid objects. See the book *3D Graphics Programming With QuickDraw 3D* for an explanation of how to specify a CSG equation. The CSG equation variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_CSG` feature.

`kQATag_BufferComposite`
The buffer compositing mode of the drawing engine. This mode determines how a drawing engine composites generated pixels with the initial contents of the drawing buffer. See "Buffer Compositing Modes" (page 1555) for a description of the available buffer compositing modes. The default value for a drawing engine that supports buffer compositing is `kQABufferComposite_None`. The buffer compositing state variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_BufferComposite` feature.

`kQATagGL_DrawBuffer`
The OpenGL color buffer of the drawing engine. This determines where a drawing engine draws when writing colors to a frame buffer. See "Buffer Drawing Operations" (page 1557) for a description of the buffer drawing modes. The default value of this variable for a drawing engine that

supports OpenGL buffering is `kQAGL_DrawBuffer_Front` for single-buffered contexts and `kQAGL_DrawBuffer_Back` for double-buffered contexts. The OpenGL color buffer state variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_OpenGL` feature.

`kQATagGL_TextureWrapU`

The OpenGL texture *u* wrapping mode of the drawing engine. See "Texture Wrapping Values" (page 1556) for a description of the wrapping modes. The default value of this variable for a drawing engine that supports OpenGL texture wrapping is `kQAGL_Repeat`. The OpenGL texture *u* wrapping mode state variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_OpenGL` feature.

`kQATagGL_TextureWrapV`

The OpenGL texture *v* wrapping mode of the drawing engine. See "Texture Wrapping Values" (page 1556) for a description of the wrapping modes. The default value of this variable for a drawing engine that supports OpenGL texture wrapping is `kQAGL_Repeat`. The OpenGL texture *v* wrapping mode state variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_OpenGL` feature.

`kQATagGL_TextureMagFilter`

The OpenGL **texture magnification function** of the drawing engine. This function is called when a pixel being textured maps to an area that is less than or equal to one texture element. The default value of this variable for a drawing engine that supports OpenGL texture magnification is `kQAGL_Linear`. The OpenGL texture magnification function state variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_OpenGL` feature.

`kQATagGL_TextureMinFilter`

The OpenGL **texture minifying function** of the drawing engine. This function is called when a pixel being textured maps to an area that is greater than one texture element. See **[to be supplied]** for a description of the available minifying functions. The default value of this variable for a drawing engine that supports OpenGL texture minifying is

kQAGL_ToBeSupplied. The OpenGL texture minifying function state variable is optional; it must be supported only when a drawing engine supports the kQAOptional_OpenGL feature.

kQATagGL_ScissorXMin
The minimum *x* value of the **scissor box,** a rectangle that determines which pixels can be modified by drawing commands. This state variable is optional; it must be supported only when a drawing engine supports the kQAOptional_OpenGL feature.

kQATagGL_ScissorYMin
The minimum *y* value of the scissor box. This state variable is optional; it must be supported only when a drawing engine supports the kQAOptional_OpenGL feature.

kQATagGL_ScissorXMax
The maximum *x* value of the scissor box. This state variable is optional; it must be supported only when a drawing engine supports the kQAOptional_OpenGL feature.

kQATagGL_ScissorYMax
The maximum *y* value of the scissor box. This state variable is optional; it must be supported only when a drawing engine supports the kQAOptional_OpenGL feature.

kQATagGL_BlendSrc The source blending operation of the drawing engine. This control determines how a drawing engine computes the red, green, blue, and alpha source-blending factors when performing transparency blending. The source blending operation state variable is optional; it must be supported only when a drawing engine supports both the kQAOptional_Blend and kQAOptional_OpenGL features.

kQATagGL_BlendDst The destination blending operation of the drawing engine. This control determines how a drawing engine computes the red, green, blue, and alpha destination-blending factors when performing transparency blending. The destination blending operation state variable is optional; it must be supported only when a drawing engine supports both the kQAOptional_Blend and kQAOptional_OpenGL features.

kQATagGL_LinePattern
The OpenGL **line stipple pattern** of the drawing engine.

This pattern specifies which bits in a line are to be drawn and which are masked out.

`kQATagGL_AreaPattern0`
The first of 32 registers that specify an **area stipple pattern.**

`kQATagGL_AreaPattern31`
The last of 32 area stipple pattern registers.

`kQATag_EngineSpecific_Minimum`
The minimum tag value to be used for variables that are specific to a particular drawing engine. Any custom variables you support must have tag values greater than or equal to this value. Note that you should use engine-specific tags only in exceptional circumstances, because the operations determined by the associated state variables are not generally accessible.

Here are the tags for state variables having floating-point values:

```
typedef enum TQATagFloat {
    kQATag_ColorBG_a                    = 1,    /*required variables*/
    kQATag_ColorBG_r                    = 2,
    kQATag_ColorBG_g                    = 3,
    kQATag_ColorBG_b                    = 4,
    kQATag_Width                        = 5,
    kQATag_ZMinOffset                   = 6,
    kQATag_ZMinScale                    = 7,
    kQATagGL_DepthBG                    = 112,  /*OpenGL variables*/
    kQATagGL_TextureBorder_a            = 113,
    kQATagGL_TextureBorder_r            = 114,
    kQATagGL_TextureBorder_g            = 115,
    kQATagGL_TextureBorder_b            = 116
} TQATagFloat;
```

Constant descriptions

`kQATag_ColorBG_a` The alpha channel value of a drawing engine's background color. This value must be greater than or equal to 0.0 and less than or equal to 1.0. The default value for the background color alpha channel is 0.0. The background color alpha channel state variable must be supported by all drawing engines.

`kQATag_ColorBG_r` The red component of a drawing engine's background color. This value must be greater than or equal to 0.0 and less than or equal to 1.0. The default value for the background color red component is 0.0. The background color red component state variable must be supported by all drawing engines.

`kQATag_ColorBG_g` The green component of a drawing engine's background color. This value must be greater than or equal to 0.0 and less than or equal to 1.0. The default value for the background color green component is 0.0. The background color green component state variable must be supported by all drawing engines.

`kQATag_ColorBG_b` The blue component of a drawing engine's background color. This value must be greater than or equal to 0.0 and less than or equal to 1.0. The default value for the background color blue component is 0.0. The background color blue component state variable must be supported by all drawing engines.

`kQATag_Width` The width (in pixels) of points or lines drawn by the drawing engine. This value must be greater than or equal to 0.0 and less than or equal to `kQAMaxWidth` (currently defined as 128.0). The default value for the width is 1.0. The width state variable must be supported by all drawing engines.

`kQATag_ZMinOffset` The minimum *z* offset that must be performed to guarantee that a drawn object passes the `kQAZFunction_LT` hidden surface test. This variable is read-only; you cannot set its value. In general, a drawing engine that employs fixed-point values for the *z* coordinate returns a small negative value (for example, –1/65536) for the minimum offset; a drawing engine that employs floating-point values for the *z* coordinate returns 0.0 for the minimum offset.

`kQATag_ZMinScale` The minimum *z* scale factor that must be applied to guarantee that a drawn object passes the `kQAZFunction_LT` hidden surface test. This variable is read-only; you cannot set its value. In general, a drawing engine that employs fixed-point values for the *z* coordinate returns 1.0 for the minimum scale factor; a drawing engine that employs floating-point values for the *z* coordinate returns a value

slightly less than 1.0 (for example, 0.9999) for the minimum scale factor.

`kQATagGL_DepthBG` The OpenGL background z of the drawing engine. The default value of this variable for a drawing engine that supports OpenGL texture magnification is `kQAGL_Linear`. The OpenGL background z state variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_OpenGL` feature.

`kQATagGL_TextureBorder_a`
The alpha component of a drawing engine's texture border color. This value must be greater than or equal to 0.0 and less than or equal to 1.0. The default value for the texture border color alpha component is 0.0. The texture border color alpha component state variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_OpenGL` feature.

`kQATagGL_TextureBorder_r`
The red component of a drawing engine's texture border color. This value must be greater than or equal to 0.0 and less than or equal to 1.0. The default value for the texture border color red component is 0.0. The texture border color red component state variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_OpenGL` feature.

`kQATagGL_TextureBorder_g`
The green component of a drawing engine's texture border color. This value must be greater than or equal to 0.0 and less than or equal to 1.0. The default value for the texture border color green component is 0.0. The texture border color green component state variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_OpenGL` feature.

`kQATagGL_TextureBorder_b`
The blue component of a drawing engine's texture border color. This value must be greater than or equal to 0.0 and less than or equal to 1.0. The default value for the texture border color blue component is 0.0. The texture border color blue component state variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_OpenGL` feature.

Here are the tags for state variables having pointer values:

```
typedef enum TQATagPtr {
    kQATag_Texture                      = 13
} TQATagPtr;
```

Constant descriptions

`kQATag_Texture` A pointer to the current texture map of the drawing engine, as created by the `QATextureNew` function. The default value for the texture map pointer is `NULL`. The texture map pointer state variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_Texture` feature.

Z Sorting Function Selectors

A drawing engine must support a **z sorting function** that determines which surfaces are to be removed during hidden surface removal. You specify an engine's z sorting function by assigning a value to its `kQATag_ZFunction` state variable. The default value of this variable for a drawing engine that is z buffered is `kQAZFunction_LT`; the default value (and also the only possible value) for a draw context that is not z buffered is `kQAZFunction_None`.

IMPORTANT

If a drawing engine supports `kQAOptional_PerspectiveZ` and if the state variable `kQATag_PerspectiveZ` is set to the value `kQAPerspectiveZ_On`, then the state variable `kQATag_ZFunction` should be interpreted so that it yields the same result as when the value of `kQATag_PerspectiveZ` is `kQAPerspectiveZ_Off`. ▲

```
#define kQAZFunction_None                       0
#define kQAZFunction_LT                         1
#define kQAZFunction_EQ                         2
#define kQAZFunction_LE                         3
#define kQAZFunction_GT                         4
#define kQAZFunction_NE                         5
#define kQAZFunction_GE                         6
#define kQAZFunction_True                       7
```

Constant descriptions

`kQAZFunction_None` The z value is neither tested nor written.

`kQAZFunction_LT` A new z value is visible if it is less than the value in the z buffer.

`kQAZFunction_EQ` A new z value is visible if it is equal to the value in the z buffer. This selector should be passed only to drawing engines that support the optional OpenGL features.

`kQAZFunction_LE` A new z value is visible if it is less than or equal to the value in the z buffer. This selector should be passed only to drawing engines that support the optional OpenGL features.

`kQAZFunction_GT` A new z value is visible if it is greater than the value in the z buffer. This selector should be passed only to drawing engines that support the optional OpenGL features.

`kQAZFunction_NE` A new z value is visible if it is not equal to the value in the z buffer. This selector should be passed only to drawing engines that support the optional OpenGL features.

`kQAZFunction_GE` A new z value is visible if it is greater than or equal to the value in the z buffer. This selector should be passed only to drawing engines that support the optional OpenGL features.

`kQAZFunction_True` A new z value is always visible.

Antialiasing Selectors

You specify an engine's antialiasing mode by assigning a value to its `kQATag_Antialias` state variable. The default value of this variable for a drawing engine that supports antialiasing is `kQAAntiAlias_Fast`.

```
#define kQAAntiAlias_Off                        0
#define kQAAntiAlias_Fast                       1
#define kQAAntiAlias_Mid                        2
#define kQAAntiAlias_Best                       3
```

Constant descriptions

`kQAAntiAlias_Off` Antialiasing is off.

`kQAAntiAlias_Fast`	The drawing engine performs whatever level of antialiasing it can do with no speed penalty. This often means that antialiasing is turned off.
`kQAAntiAlias_Mid`	The drawing engine performs a medium level of antialiasing. You should use this antialiasing mode when you want to perform antialiasing interactively.
`kQAAntiAlias_Best`	The drawing engine performs the highest level of antialiasing it can. This mode may be unsuitable for interactive rendering.

Blending Operations

A drawing engine may support a **transparency blending function** that determines the kind of transparency blending applied to a drawing context when combining new ("source") pixels with the pixels already in a frame buffer ("destination"). You specify an engine's transparency blending function by assigning a value to its `kQATag_Blend` state variable. The default value of this variable for a draw context that supports transparency blending is `kQABlend_PreMultiply`.

In the equations below, the factors a_s, r_s, g_s, and b_s represent the alpha, red, green and blue components of a source pixel; the factors a_d, r_d, g_d, and b_d represent the alpha, red, green and blue components of a destination pixel.

```
#define kQABlend_PreMultiply                    0
#define kQABlend_Interpolate                    1
#define kQABlend_OpenGL                         2
```

Constant descriptions

`kQABlend_PreMultiply`

The drawing engine uses a premultiplied blending function. The components of a pixel written to the frame buffer are computed using these equations:

$$a = 1 - ((1 - a_s) \times (1 - a_d))$$
$$r = r_s + ((1 - a_s) \times r_d)$$
$$g = g_s + ((1 - a_s) \times g_d)$$
$$b = b_s + ((1 - a_s) \times b_d)$$

In general, you should use the premultiplied blending function for rendering shaded transparent 3D primitives (such as triangles). The premultiplied function does not scale the source color components by the alpha value a_s; as a result, this function allows a transparent object to have a specular highlight value that is greater than its alpha channel value. For example, a sheet of glass might allow 99% of the light behind it to pass though (indicating an alpha channel value of 0.01). However, that same sheet of glass might have a specular highlight value much greater than 0.01. The premultiplied function allows the drawing engine to render this object correctly.

`kQABlend_Interpolate`

The drawing engine uses an interpolated blending function. The components of a pixel written to the frame buffer are computed using these equations:

$$a = 1 - ((1 - a_s) \times (1 - a_d))$$
$$r = (r_s \times a_s) + ((1 - a_s) \times r_d)$$
$$g = (g_s \times a_s) + ((1 - a_s) \times g_d)$$
$$b = (b_s \times a_s) + ((1 - a_s) \times b_d)$$

The interpolated blending function is not entirely suitable for rendering shaded transparent objects, but it is very effective for compositing bitmap images.

`kQABlend_OpenGL` The drawing engine uses the OpenGL blending function determined by the values of the `kQATagGL_BlendSrc` and `kQATagGL_BlendDest` state variables. For complete information on OpenGL blending functions, consult the description of the `glBlendFunc` function in *OpenGL™ Reference Manual*. OpenGL blending functions are supported only by drawing engines that support the `kQAOptional_OpenGL` feature.

Z Perspective Selectors

A drawing engine may support a **z perspective control** that determines whether the `z` or the `invW` field of a vertex (of type `TQAVGouraud` or `TQAVTexture`) is to be used for hidden surface removal. You specify an engine's z perspective

control by assigning a value to its `kQATag_PerspectiveZ` state variable. The default value of this variable for a drawing engine that supports z perspective is `kQAPerspectiveZ_Off`.

```
#define kQAPerspectiveZ_Off                     0
#define kQAPerspectiveZ_On                      1
```

Constant descriptions

`kQAPerspectiveZ_Off`
: The drawing engine performs hidden surface removal using `z` values, as is standard.

`kQAPerspectiveZ_On`
: The drawing engine performs hidden surface removal using `invW` values, which results in perspective-correct hidden surface removal.

Texture Filter Selectors

A drawing engine may support a **texture mapping filter mode** that determines how a drawing engine performs texture mapping. You specify an engine's texture filter by assigning a value to its `kQATag_TextureFilter` state variable. The default value of this variable for a drawing engine that supports texture mapping is `kQATextureFilter_Fast`.

```
#define kQATextureFilter_Fast                   0
#define kQATextureFilter_Mid                    1
#define kQATextureFilter_Best                   2
```

Constant descriptions

`kQATextureFilter_Fast`
: The drawing engine performs whatever level of texture filtering it can do with no speed penalty. This often means that no texture filtering is performed.

`kQATextureFilter_Mid`
: The drawing engine performs a medium level of texture filtering. You should use this texture mapping filter mode when you want to perform texture mapping interactively.

`kQATextureFilter_Best`
: The drawing engine performs the highest level of texture

filtering it can. This mode may be unsuitable for interactive rendering.

Texture Operations

A drawing engine may support a **texture mapping operation** that determines how a drawing engine performs texture mapping. You specify an engine's texture mapping operation by assigning a value to its `kQATag_TextureOp` state variable. The default value of this variable for a drawing engine that supports texture mapping is `kQATextureOp_None`.

You can use the following masks to specify a texture mapping operation. The bits are ORed together to determine the desired operation.

```
#define kQATextureOp_None                       0
#define kQATextureOp_Modulate                   (1 << 0)
#define kQATextureOp_Highlight                  (1 << 1)
#define kQATextureOp_Decal                      (1 << 2)
#define kQATextureOp_Shrink                     (1 << 3)
```

Constant descriptions

`kQATextureOp_None` The drawing engine supports no special texture mapping operations. The drawing engine simply replaces an object's color with the texture map color. This mode results in a flat-looking image with no lighting effects, which is most useful when the texture mapping engine is used as a 2D warping engine (for example, for video effects). The texture map's alpha channel values control the transparency of a rendered object on a per-pixel basis. The alpha channel value of a particular pixel is the product of texture map's alpha channel value and the vertex alpha channel value (which is interpolated from the `TQAVTexture` data structure).

`kQATextureOp_Modulate`
The texture map color is modulated with the interpolated diffuse colors (from the `kd_r`, `kd_g`, and `kd_b` fields of a texture vertex).

`kQATextureOp_Highlight`
The interpolated specular colors (from the `ks_r`, `ks_g`, and `ks_b` fields of a texture vertex) are added to the texture map color.

`kQATextureOp_Decal` The texture map alpha channel value is used to blend the texture map color and the interpolated decal colors (from the `r`, `g`, and `b` fields of a texture vertex). When the texture map alpha channel value is 0, the texture map color is replaced with the interpolated `r`, `g`, and `b` values.

`kQATextureOp_Shrink`
The drawing engine modifies any *u* and *v* values so that they always lie in the range 0.0 to 1.0 inclusive. This guarantees that wrapping not occur. In theory, *u* and *v* values in the range [0.0, 1.0] should never cause wrapping. In practice, however, errors that occur during *uv* interpolation can cause *uv* overflow or underflow, which can result in occasional one pixel texture wrapping at the 0.0 and 1.0 boundaries.

IMPORTANT

The clamping specified by the `kQATextureOp_Shrink` flag is not the same type of OpenGL texture clamping specified by the `kQATagGL_TextureWrapU` and `kQATagGL_TextureWrapV` state variables (see "Texture Wrapping Values" (page 1556)). OpenGL clamping is designed to accept arbitrary *uv* values, while clamping specified by the `kQATextureOp_Shrink` flag operates only on *uv* values in the range 0.0 to 1.0. The `kQATextureOp_Shrink` clamping is therefore less expensive to implement (perhaps simply by compressing the range of *u* and *v* slightly before beginning interpolation). Any drawing engine that does support OpenGL clamping can use that code to support `kQATextureOp_Shrink` clamping. ▲

CSG IDs

A drawing engine may support **CSG IDs** that determine what number a drawing engine assigns to triangles submitted for drawing. You specify an engine's CSG ID by assigning a value to its `kQATag_CSGTag` state variable. The default value of this variable for a drawing engine that supports CSG is `kQACSGTag_None`. You can use the following constants to specify a CSG ID.

```
#define kQACSGTag_None                  0xffffffffUL
#define kQACSGTag_0                     0
#define kQACSGTag_1                     1
#define kQACSGTag_2                     2
#define kQACSGTag_3                     3
#define kQACSGTag_4                     4
```

Constant descriptions

`kQACSGTag_None` Do not assign CSG IDs to submitted triangles.

`kQACSGTag_0` Submitted triangles have the CSG ID 0.

`kQACSGTag_1` Submitted triangles have the CSG ID 1.

`kQACSGTag_2` Submitted triangles have the CSG ID 2.

`kQACSGTag_3` Submitted triangles have the CSG ID 3.

`kQACSGTag_4` Submitted triangles have the CSG ID 4.

Buffer Compositing Modes

A drawing engine may support a buffer compositing mode that determines how the drawing engine composites generated pixels with the initial contents of the drawing buffer. You specify an engine's buffer compositing mode by assigning a value to its `kQATag_BufferComposite` state variable. The default value of this variable for a drawing engine that supports buffer compositing is `kQABufferComposite_None`. You can use the following constants to specify a buffer compositing mode.

```
#define kQABufferComposite_None         0
#define kQABufferComposite_PreMultiply  1
#define kQABufferComposite_Interpolate  2
```

Constant descriptions

`kQABufferComposite_None`
The drawing engine performs no compositing. Newly generated pixels overwrite the initial contexts of the buffer.

`kQABufferComposite_PreMultiply`
The drawing engine composites new pixels with existing pixels by premulitplying their color components.

`kQABufferComposite_Interpolate`
: The drawing engine composites new pixels with existing pixels by interpolating their color components.

Texture Wrapping Values

A drawing engine may support OpenGL texture wrapping, in which case you might need to specify a texture wrapping mode in the *u* and *v* parametric directions. You specify an engine's texture wrapping modes by assigning a value to its `kQATagGL_TextureWrapU` and `kQATagGL_TextureWrapV` state variables. The default value of both these variables for a drawing engine that supports OpenGl texture wrapping is `kQAGL_Repeat`. You can use the following constants to specify a texture wrapping mode.

```
#define kQAGL_Repeat                            0
#define kQAGL_Clamp                             1
```

Constant descriptions

`kQAGL_Repeat`
: The integer part of a *u* or *v* coordinate is ignored, thereby causing a texture to be repeated across the surface of an object.

`kQAGL_Clamp`
: The *u* or *v* coordinates are clamped to the range [0, 1]. This mode prevents wrapping artifacts from occurring when a single texture is mapped onto an object.

Source Blending Values

When a drawing engine's transparency blending function is set to the value `kQABlend_OpenGL`, the state variable `kQATagGL_BlendSrc` must be set to a value to indicate the red, green, blue, and alpha source blending factors. You can use these constants to define the source blending factors.

```
#define kQAGL_SourceBlend_XXX                   0
```

Constant descriptions

`kQAGL_SourceBlend_XXX`

Destination Blending Values

When a drawing engine's transparency blending function is set to the value `kQABlend_OpenGL`, the state variable `kQATagGL_BlendDst` must be set to a value to indicate the red, green, blue, and alpha destination blending factors. You can use these constants to define the destination blending factors.

```
#define kQAGL_DestBlend_XXX                         0
```

Constant descriptions

`kQAGL_DestBlend_XXX`

Buffer Drawing Operations

A drawing engine may support an OpenGL buffer drawing mode that determines which color buffers a drawing engine draws into. You specify one or more buffers by assigning a value to the `kQATagGL_DrawBuffer` state variable of that engine. The default value of this variable for a drawing engine that supports OpenGL buffering is `kQAGL_DrawBuffer_Front` for single-buffered contexts and `kQAGL_DrawBuffer_Back` for double-buffered contexts. You can use the following masks to specify a buffer drawing mode.

```
#define kQAGL_DrawBuffer_None                       0
#define kQAGL_DrawBuffer_FrontLeft                  (1 << 0)
#define kQAGL_DrawBuffer_FrontRight                 (1 << 1)
#define kQAGL_DrawBuffer_BackLeft                   (1 << 2)
#define kQAGL_DrawBuffer_BackRight                  (1 << 3)
#define kQAGL_DrawBuffer_Front                      \
        (kQAGL_DrawBuffer_FrontLeft | kQAGL_DrawBuffer_FrontRight)
#define kQAGL_DrawBuffer_Back                       \
        (kQAGL_DrawBuffer_BackLeft | kQAGL_DrawBuffer_BackRight)
```

Constant descriptions

`kQAGL_DrawBuffer_None`
: The drawing engine draws into no color buffer.

`kQAGL_DrawBuffer_FrontLeft`
: The drawing engine draws into the front left buffer only.

`kQAGL_DrawBuffer_FrontRight`
: The drawing engine draws into the front right buffer only.

`kQAGL_DrawBuffer_BackLeft`
: The drawing engine draws into the back left buffer only.

`kQAGL_DrawBuffer_BackRight`
: The drawing engine draws into the back right buffer only.

`kQAGL_DrawBuffer_Front`
: The drawing engine draws into the front left and right buffers only.

`kQAGL_DrawBuffer_Back`
: The drawing engine draws into the back left and right buffers only.

Vertex Modes

The `vertexMode` parameter for the `QADrawVGouraud` and `QADrawVTexture` functions specifies a **vertex mode,** which determines how the drawing engine interprets and draws an array of vertices.

```
typedef enum TQAVertexMode {
    kQAVertexMode_Point                 = 0,
    kQAVertexMode_Line                  = 1,
    kQAVertexMode_Polyline              = 2,
    kQAVertexMode_Tri                   = 3,
    kQAVertexMode_Strip                 = 4,
    kQAVertexMode_Fan                   = 5,
    kQAVertexMode_NumModes              = 6
} TQAVertexMode;
```

Constant descriptions

`kQAVertexMode_Point`
: Draw points. Each vertex in the array is drawn as a point. The engine draws `nVertices` points (where `nVertices` is the number of vertices in the vertex array).

`kQAVertexMode_Line` Draw line segments. Each successive pair of vertices in the array determines a single line segment. The engine draws `nVertices`/2 line segments.

`kQAVertexMode_Polyline`
: Draw connected line segments. Each vertex in the array and its predecessor determine a line segment. The engine draws `nVertices`–1 line segments.

`kQAVertexMode_Tri` Draw triangles. Each successive triple of vertices in the array determines a single triangle. The engine draws `nVertices`/3 triangles.

`kQAVertexMode_Strip`
Draw a strip of triangles. The first three vertices in the array determine a triangle, and each successive vertex and its two predecessors determine a triangle that abuts the existing strip of triangles. The engine draws `nVertices`–2 triangles.

`kQAVertexMode_Fan` Draw a fan. The first three vertices in the array determine a triangle; each successive vertex, its immediate predecessor, and the first vertex in the array determine a triangle that abuts the existing fan. The engine draws `nVertices`–2 triangles.

`kQAVertexMode_NumModes`
The number of vertex modes currently defined.

Gestalt Selectors

You can use the `QAEngineGestalt` function to get information about a drawing engine. You pass `QAEngineGestalt` a selector that determines the kind of information about the engine you want to receive and a pointer to a buffer into which the information is to be copied. The selectors are defined by constants. Note that your application must allocate space for the buffer (pointed to by the `response` parameter) into which the information is copied.

```
typedef enum TQAGestaltSelector {
    kQAGestalt_OptionalFeatures         = 0,
    kQAGestalt_FastFeatures             = 1,
    kQAGestalt_VendorID                 = 2,
    kQAGestalt_EngineID                 = 3,
    kQAGestalt_Revision                 = 4,
    kQAGestalt_ASCIINameLength          = 5,
    kQAGestalt_ASCIIName                = 6,
    kQAGestalt_TextureMemory            = 7,
    kQAGestalt_FastTextureMemory        = 8,
    kQAGestalt_NumSelectors             = 9
} TQAGestaltSelector;
```

Constant descriptions

`kQAGestalt_OptionalFeatures`
: `QAEngineGestalt` returns a value whose bits encode the optional features supported by the drawing engine. The `response` parameter must point to a buffer of type `unsigned long`. See "Gestalt Optional Features Response Masks" (page 1561) for a description of the meaning of the bits in the returned value.

`kQAGestalt_FastFeatures`
: `QAEngineGestalt` returns a value whose bits encode the features supported by the drawing engine that are accelerated. The `response` parameter must point to a buffer of type `unsigned long`. See "Gestalt Fast Features Response Masks" (page 1563) for a description of the meaning of the bits in the returned value.

`kQAGestalt_VendorID`
: `QAEngineGestalt` returns the vendor ID of the drawing engine. The `response` parameter must point to a buffer of type `long`. See "Vendor and Engine IDs" (page 1565) for a list of the currently defined vendor IDs.

`kQAGestalt_EngineID`
: `QAEngineGestalt` returns the engine ID of the drawing engine. The `response` parameter must point to a buffer of type `long`. See "Vendor and Engine IDs" (page 1565) for a list of the currently defined engine IDs.

`kQAGestalt_Revision`
: `QAEngineGestalt` returns the revision number of the drawing engine. (Larger numbers indicate more recent revisions.) The `response` parameter must point to a buffer of type `long`.

`kQAGestalt_ASCIINameLength`
: `QAEngineGestalt` returns the number of characters in the ASCII name of the drawing engine. The `response` parameter must point to a buffer of type `long`.

`kQAGestalt_ASCIIName`
: `QAEngineGestalt` returns the ASCII name of the drawing engine. The `response` parameter must point to a C string whose length you have determined by passing the `kQAGestalt_ASCIINameLength` selector to `QAEngineGestalt`.

`kQAGestalt_TextureMemory`
: `QAEngineGestalt` returns the size, in bytes, of the memory available for storing texture maps. Note that the amount of memory required to hold a particular texture map depends on the texture flags of that texture map—in particular, on the texture compression and mipmapping flags. (See "Texture Flags Masks" (page 1566) for details.) As a result, the size returned by `QAEngineGestalt` is only a rough indication of the number of texture maps that can be created. The `response` parameter must point to a buffer of type `Size`.

`kQAGestalt_FastTextureMemory`
: `QAEngineGestalt` returns the size, in bytes, of the fast memory available for storing texture maps. (Fast texture memory is memory located on a hardware accelerator.) Note that the amount of memory required to hold a particular texture map depends on the texture flags of that texture map—in particular, on the texture compression and mipmapping flags. (See "Texture Flags Masks" (page 1566) for details.) As a result, the size returned by `QAEngineGestalt` is only a rough indication of the number of texture maps that can be created. The `response` parameter must point to a buffer of type `Size`.

`kQAGestalt_NumSelectors`
: The number of selectors currently defined.

Gestalt Optional Features Response Masks

When you pass the `kQAGestalt_OptionalFeatures` selector to the `QAEngineGestalt` function, `QAEngineGestalt` returns (through its `response` parameter) a value that indicates the optional features supported by a drawing engine. You can use these masks to test that value for a specific feature. The bits corresponding to supported features are ORed together to determine the returned value.

Note
A drawing engine may support an optional feature in software only (that is, unaccelerated). You can use the `kQAGestalt_FastFeatures` selector to determine which, if any, features are accelerated by a drawing engine. ◆

```
#define kQAOptional_None                        0
#define kQAOptional_DeepZ                       (1 << 0)
#define kQAOptional_Texture                     (1 << 1)
#define kQAOptional_TextureHQ                   (1 << 2)
#define kQAOptional_TextureColor                (1 << 3)
#define kQAOptional_Blend                       (1 << 4)
#define kQAOptional_BlendAlpha                  (1 << 5)
#define kQAOptional_Antialias                   (1 << 6)
#define kQAOptional_ZSorted                     (1 << 7)
#define kQAOptional_PerspectiveZ                (1 << 8)
#define kQAOptional_OpenGL                      (1 << 9)
#define kQAOptional_NoClear                     (1 << 10)
#define kQAOptional_CSG                         (1 << 11)
#define kQAOptional_BoundToDevice               (1 << 12)
#define kQAOptional_CL4                         (1 << 13)
#define kQAOptional_CL8                         (1 << 14)
#define kQAOptional_BufferComposite             (1 << 15)
```

Constant descriptions

`kQAOptional_None` This value is returned if the drawing engine supports no optional features.

`kQAOptional_DeepZ` This bit is set if the drawing engine supports **deep z buffering** (that is, z buffering with a resolution of greater than or equal to 24 bits per pixel).

`kQAOptional_Texture`
This bit is set if the drawing engine supports texture mapping.

`kQAOptional_TextureHQ`
This bit is set if the drawing engine supports **high-quality texture mapping** (that is, texture mapping using trilinear interpolation or an equivalent algorithm).

`kQAOptional_TextureColor`
This bit is set if the drawing engine supports full color texture modulation and highlighting.

`kQAOptional_Blend` This bit is set if the drawing engine supports transparency blending.

`kQAOptional_BlendAlpha`
This bit is set if the drawing engine supports transparency blending that uses an alpha channel.

`kQAOptional_Antialias`
: This bit is set if the drawing engine supports antialiasing.

`kQAOptional_ZSorted`
: This bit is set if the drawing engine supports z sorted rendering (for example, for transparency). If this bit is clear, an application must submit transparent objects for rendering in back-to-front z order (or the blending functions will not yield correct results). In general, an application should submit opaque objects first, followed by any transparent objects in back-to-front z order.

`kQAOptional_PerspectiveZ`
: This bit is set if the drawing engine supports perspective-corrected hidden surface removal.

`kQAOptional_OpenGL`
: This bit is set if the drawing engine supports the extended OpenGL feature set.

`kQAOptional_NoClear`
: This bit is set if the drawing engine doesn't clear the buffer before drawing (so that double-buffering might not be required in some applications).

`kQAOptional_CSG`
: This bit is set if the drawing engine supports CSG.

`kQAOptional_BoundToDevice`
: This bit is set if the drawing engine is tightly bound to a specific graphics device.

`kQAOptional_CL4`
: This bit is set if the drawing engine supports the `kQAPixel_CL4` pixel type.

`kQAOptional_CL8`
: This bit is set if the drawing engine supports the `kQAPixel_CL8` pixel type.

`kQAOptional_BufferComposite`
: This bit is set if the drawing engine supports buffer compositing.

Gestalt Fast Features Response Masks

When you pass the `kQAGestalt_FastFeatures` selector to the `QAEngineGestalt` function, `QAEngineGestalt` returns (through its `response` parameter) a value that indicates which, if any, features supported by a drawing engine are accelerated. You can use these masks to test that value for a specific feature. The bits

corresponding to accelerated features are ORed together to determine the returned value.

Note
A feature is considered accelerated if it is performed substantially faster by the drawing engine than it would be if performed in software only. ◆

```
#define kQAFast_None                    0
#define kQAFast_Line                    (1 << 0)
#define kQAFast_Gouraud                 (1 << 1)
#define kQAFast_Texture                 (1 << 2)
#define kQAFast_TextureHQ               (1 << 3)
#define kQAFast_Blend                   (1 << 4)
#define kQAFast_Antialiasing            (1 << 5)
#define kQAFast_ZSorted                 (1 << 6)
#define kQAFast_CL4                     (1 << 7)
#define kQAFast_CL8                     (1 << 8)
```

Constant descriptions

`kQAFast_None` This value is returned if the drawing engine accelerates no features.

`kQAFast_Line` This bit is set if the drawing engine accelerates line drawing.

`kQAFast_Gouraud` This bit is set if the drawing engine accelerates Gouraud shading.

`kQAFast_Texture` This bit is set if the drawing engine accelerates texture mapping.

`kQAFast_TextureHQ` This bit is set if the drawing engine accelerates high-quality texture mapping.

`kQAFast_Blend` This bit is set if the drawing engine accelerates transparency blending.

`kQAFast_Antialiasing` This bit is set if the drawing engine accelerates antialiasing.

`kQAFast_ZSorted` This bit is set if the drawing engine accelerates z sorted rendering.

`kQAFast_CL4` This bit is set if the drawing engine accelerates `kQAPixel_CL4` pixel type rendering.

`kQAFast_CL8` This bit is set if the drawing engine accelerates `kQAPixel_CL8` pixel type rendering.

Vendor and Engine IDs

QuickDraw 3D RAVE defines constants for vendor IDs. You pass a vendor ID as a parameter to the `QAEngineEnable` and `QAEngineDisable` functions, and you receive a vendor ID when you pass the `kQAGestalt_VendorID` selector to the `QAEngineGestalt` function.

```
#define kQAVendor_BestChoice                    (-1)
#define kQAVendor_Apple                         0
#define kQAVendor_ATI                           1
#define kQAVendor_Radius                        2
#define kQAVendor_Mentor                        3
#define kQAVendor_Matrox                        4
#define kQAVendor_Yarc                          5
#define kQAVendor_DiamondMM                     6
```

Constant descriptions

`kQAVendor_BestChoice`
: The best drawing engine available for the target device. You should use this value as the default.

`kQAVendor_Apple`
: The vendor is Apple Computer, Inc.

`kQAVendor_ATI`
: The vendor is ATI Technologies Inc.

`kQAVendor_Radius`
: The vendor is Radius.

`kQAVendor_Mentor`
: The vendor is Mentor Software, Inc.

`kQAVendor_Matrox`
: The vendor is Matrox Graphics.

`kQAVendor_Yarc`
: The vendor is YARC Systems.

`kQAVendor_DiamondMM`
: The vendor is Diamond Multimedia.

For the vendor `kQAVendor_Apple`, QuickDraw 3D RAVE defines these constants for engine IDs.

```
#define kQAEngine_AppleSW                       0
#define kQAEngine_AppleHW                       (-1)
#define kQAEngine_AppleHW2                      1
#define kQAEngine_AppleHW3                      2
```

Constant descriptions

`kQAEngine_AppleSW`
: The default software rasterizer.

`kQAEngine_AppleHW`
: The QuickDraw 3D accelerator card.

`kQAEngine_AppleHW2` Another Apple 3D accelerator.

`kQAEngine_AppleHW3` Another Apple 3D accelerator.

Triangle Flags Masks

The `flags` parameter for the `QADrawTriGouraud` and `QADrawTriTexture` functions specifies a **triangle mode,** which determines how the drawing engine draws a triangle. You can use these masks to set the `flags` parameter.

```
#define kQATriFlags_None                        0
#define kQATriFlags_Backfacing                  (1 << 0)
```

Constant descriptions

`kQATriFlags_None` Pass this value for no triangle flags. The triangle is frontfacing or has an unspecified orientation.

`kQATriFlags_Backfacing`
The triangle is backfacing. You should set this bit for all triangles known to be backfacing (to help the drawing engine resolve ambiguous hidden surface removal situations).

Texture Flags Masks

The `flags` parameter for the `QATextureNew` function specifies a **texture mode,** which determines certain features of the new texture map. You can use these masks to set the `flags` parameter.

```
#define kQATexture_None                         0
#define kQATexture_Lock                         (1 << 0)
#define kQATexture_Mipmap                       (1 << 1)
#define kQATexture_NoCompression                (1 << 2)
#define kQATexture_HighCompression              (1 << 3)
```

Constant descriptions

`kQATexture_None` Pass this value for no texture features.

`kQATexture_Lock` The new texture map should remain locked in memory and not be swapped out. You should set this flag for texture maps that are heavily used during rendering. Note,

however, that this flag is usually ignored by software-based drawing engines.

`kQATexture_Mipmap` The new texture map is mipmapped.

`kQATexture_NoCompression`
The new texture map should not be compressed.

`kQATexture_HighCompression`
The new texture map should be compressed (even if doing so takes a considerable amount of time).

Bitmap Flags Masks

The `flags` parameter passed to the `QABitmapNew` function specifies a set of bit flags that control features of the new bitmap. You can use these masks to configure a `flags` parameter.

```
#define kQABitmap_None                  0
#define kQABitmap_Lock                  (1 << 1)
#define kQABitmap_NoCompression         (1 << 2)
#define kQABitmap_HighCompression       (1 << 3)
```

Constant descriptions

`kQABitmap_None` Pass this value for no bitmap features.

`kQABitmap_Lock` The new bitmap should remain locked in memory and not be swapped out. You should set this flag for bitmap that are heavily used during rendering. Note, however, that this flag is usually ignored by software-based drawing engines.

`kQABitmap_NoCompression`
The new bitmap should not be compressed.

`kQABitmap_HighCompression`
The new bitmap should be compressed (even if doing so takes a considerable amount of time).

Draw Context Flags Masks

The `flags` parameter passed to the `QADrawContextNew` function specifies a set of bit flags that control features of the new draw context. You can use these masks to configure the `flags` parameter.

```
#define kQAContext_None                         0
#define kQAContext_NoZBuffer                    (1 << 0)
#define kQAContext_DeepZ                        (1 << 1)
#define kQAContext_DoubleBuffer                 (1 << 2)
#define kQAContext_Cache                        (1 << 3)
```

Constant descriptions

`kQAContext_None` Pass this value for no draw context features.

`kQAContext_NoZBuffer`
The new draw context should not be z buffered.

`kQAContext_DeepZ` The new draw context should have a z buffer with at least 24 bits of precision.

`kQAContext_DoubleBuffer`
The new draw context should be double buffered.

`kQAContext_Cache` The new draw context should be used for a draw context cache. When you create a draw context with this feature, it is always considered a draw context cache. Accordingly, objects rendered into a draw context cache never appear on the device (not even on a memory device). The only way to view objects rendered into a draw context cache is to use that cache to initialize a non-cached draw context.

Drawing Engine Method Selectors

To determine the addresses of some of the methods defined by a drawing engine, QuickDraw 3D RAVE calls the engine's `TQAEngineGetMethod` function, passing a **method selector** in the `methodTag` parameter. This selector indicates of which method the engine should return the address in the `method` parameter.

```
typedef enum TQAEngineMethodTag {
    kQADrawPrivateNew                       = 0,
    kQADrawPrivateDelete                    = 1,
    kQAEngineCheckDevice                    = 2,
    kQAEngineGestalt                        = 3,
    kQATextureNew                           = 4,
    kQATextureDetach                        = 5,
    kQATextureDelete                        = 6,
    kQABitmapNew                            = 7,
    kQABitmapDetach                         = 8,
```

```
    kQABitmapDelete                             = 9,
    kQAColorTableNew                            = 10,
    kQAColorTableDelete                         = 11,
    kQATextureBindColorTable                    = 12,
    kQABitmapBindColorTable                     = 13
} TQAEngineMethodTag;
```

Constant descriptions

`kQADrawPrivateNew` The `TQADrawPrivateNew` method.

`kQADrawPrivateDelete` The `TQADrawPrivateDelete` method.

`kQAEngineCheckDevice` The `TQAEngineCheckDevice` method.

`kQAEngineGestalt` The `TQAEngineGestalt` method.

`kQATextureNew` The `TQATextureNew` method.

`kQATextureDetach` The `TQATextureDetach` method.

`kQATextureDelete` The `TQATextureDelete` method.

`kQABitmapNew` The `TQABitmapNew` method.

`kQABitmapDetach` The `TQABitmapDetach` method.

`kQABitmapDelete` The `TQABitmapDelete` method.

`kQAColorTableNew` The `TQAColorTableNew` method.

`kQAColorTableDelete` The `TQAColorTableDelete` method.

`kQATextureBindColorTable` The `TQATextureBindColorTable` method.

`kQABitmapBindColorTable` The `TQABitmapBindColorTable` method.

Public Draw Context Method Selectors

The `methodTag` parameter passed to the `QARegisterDrawMethod` function specifies a type of public draw context method. QuickDraw 3D RAVE defines these constants for method selectors.

```
typedef enum TQADrawMethodTag {
    kQASetFloat                                 = 0,
    kQASetInt                                   = 1,
```

```
    kQASetPtr                                   = 2,
    kQAGetFloat                                 = 3,
    kQAGetInt                                   = 4,
    kQAGetPtr                                   = 5,
    kQADrawPoint                                = 6,
    kQADrawLine                                 = 7,
    kQADrawTriGouraud                           = 8,
    kQADrawTriTexture                           = 9,
    kQADrawVGouraud                             = 10,
    kQADrawVTexture                             = 11,
    kQADrawBitmap                               = 12,
    kQARenderStart                              = 13,
    kQARenderEnd                                = 14,
    kQARenderAbort                              = 15,
    kQAFlush                                    = 16,
    kQASync                                     = 17,
    kQASubmitVerticesGouraud                    = 18,
    kQASubmitVerticesTexture                    = 19,
    kQADrawTriMeshGouraud                       = 20,
    kQADrawTriMeshTexture                       = 21,
    kQASetNoticeMethod                          = 22,
    kQAGetNoticeMethod                          = 23
} TQADrawMethodTag;
```

Constant descriptions

`kQASetFloat`	The `TQASetFloat` method.
`kQASetInt`	The `TQASetInt` method.
`kQASetPtr`	The `TQASetPtr` method.
`kQAGetFloat`	The `TQAGetFloat` method.
`kQAGetInt`	The `TQAGetInt` method.
`kQAGetPtr`	The `TQAGetPtr` method.
`kQADrawPoint`	The `TQADrawPoint` method.
`kQADrawLine`	The `TQADrawLine` method.
`kQADrawTriGouraud`	The `TQADrawTriGouraud` method.
`kQADrawTriTexture`	The `TQADrawTriTexture` method.
`kQADrawVGouraud`	The `TQADrawVGouraud` method.
`kQADrawVTexture`	The `TQADrawVTexture` method.
`kQADrawBitmap`	The `TQADrawBitmap` method.

`kQARenderStart` The `TQARenderStart` method.

`kQARenderEnd` The `TQARenderEnd` method.

`kQARenderAbort` The `TQARenderAbort` method.

`kQAFlush` The `TQAFlush` method.

`kQASync` The `TQASync` method.

`kQASubmitVerticesGouraud`
The `TQASubmitVerticesGouraud` method.

`kQASubmitVerticesTexture`
The `TQASubmitVerticesTexture` method.

`kQADrawTriMeshGouraud`
The `TQADrawTriMeshGouraud` method.

`kQADrawTriMeshTexture`
The `TQADrawTriMeshTexture` method.

`kQASetNoticeMethod` The `TQASetNoticeMethod` method.

`kQAGetNoticeMethod` The `TQAGetNoticeMethod` method.

Notice Method Selectors

The `method` parameter passed to the `QAGetNoticeMethod` and `QASetNoticeMethod` functions specifies a type of notice method. QuickDraw 3D RAVE defines these constants for method selectors.

```
typedef enum TQAMethodSelector {
    kQAMethod_RenderCompletion                  = 0,
    kQAMethod_DisplayModeChanged                = 1
    kQAMethod_ReloadTextures                    = 2,
    kQAMethod_BufferInitialize                  = 3,
    kQAMethod_BufferComposite                   = 4,
    kQAMethod_NumSelectors                      = 5
} TQAMethodSelector;
```

Constant descriptions

`kQAMethod_RenderCompletion`
The renderer has finished rendering an image. If the draw context is double buffered, the completion method is called after the front and back buffers have been swapped.

`kQAMethod_DisplayModeChanged`
: The display mode has changed. In response to this notification, you should verify that all your draw contexts are still visible on the screen.

`kQAMethod_ReloadTextures`
: The texture memory has become invalid. In response to this notification, you should reload any textures you're using.

`kQAMethod_BufferInitialize`
: A buffer needs to be initialized. This notification is sent before rendering starts. You are responsible for clearing the buffer to the desired image. Your buffer initialization method is given a reference to the device to clear, which is always a memory device. You can set a drawing engine's `kQATag_BufferComposite` state variable to indicate how you want the engine to composite generated pixels with the pixels in that image.

`kQAMethod_BufferComposite`
: Rendering is finished and it is safe to composite. This notification is sent after rendering has finished but before the buffers are swapped. Your buffer compositing method is given a reference to the device to composite into, which is always a memory device.

`kQAMethod_NumSelectors`
: The number of method selectors currently defined.

Data Structures

This section describes the data structures provided by QuickDraw 3D RAVE.

Memory Device Structure

You specify a memory device using a **memory device structure,** defined by the `TQADeviceMemory` data type.

```
typedef struct TQADeviceMemory {
    long                                rowBytes;
    TQAImagePixelType                   pixelType;
    long                                width;
```

```
    long                            height;
    void                            *baseAddr;
} TQADeviceMemory;
```

Field descriptions

`rowBytes`	The distance, in bytes, from the beginning of one row of the memory device to the beginning of the next row of the memory device.
`pixelType`	A value that specifies the size and organization of the memory associated with a pixel in the pixmap. See "Pixel Types" (page 1536) for information on the values you can assign to this field.
`width`	The width, in pixels, of the memory device.
`height`	The height, in pixels, of the memory device.
`baseAddr`	A pointer to the beginning of the memory device.

Rectangle Structure

You specify a rectangular region of memory (for instance, to define the area into which a drawing engine is to draw) using a **rectangle structure,** defined by the `TQARect` data type. All values are interpreted to be in device coordinates.

```
typedef struct TQARect {
    long                            left;
    long                            right;
    long                            top;
    long                            bottom;
} TQARect;
```

Field descriptions

`left`	The left side of the rectangle.
`right`	The right side of the rectangle.
`top`	The top side of the rectangle.
`bottom`	The bottom side of the rectangle.

Macintosh Device and Clip Structures

QuickDraw 3D RAVE supports two types of devices and one type of clipping on the Macintosh Operating System. The available devices and clipping are defined by unions of type `TQAPlatformDevice` and `TQAPlatformClip`.

```
typedef union TQAPlatformDevice {
    TQADeviceMemory                     memoryDevice;
    GDHandle                            gDevice;
} TQAPlatformDevice;
```

Field descriptions

`memoryDevice` A memory device data structure.

`gDevice` A handle to a graphics device (of type `GDevice`).

```
typedef union TQAPlatformClip {
    RgnHandle                           clipRgn;
} TQAPlatformClip;
```

Field descriptions

`clipRgn` A handle to a clipping region.

Windows Device and Clip Structures

QuickDraw 3D RAVE supports two types of devices and one type of clipping on Windows 32 systems. The available devices and clipping are defined by unions of type `TQAPlatformDevice` and `TQAPlatformClip`.

```
typedef union TQAPlatformDevice {
    TQADeviceMemory                     memoryDevice;
    HDC                                 hdc;
    struct {
        LPDIRECTDRAW                    lpDirectDraw;
        LPDIRECTDRAWSURFACE             lpDirectDrawSurface;
    };
} TQAPlatformDevice;
```

Field descriptions

`memoryDevice` A memory device data structure.

`hdc` A handle to a draw context.

```
lpDirectDraw
lpDirectDrawSurface
```

```
typedef union TQAPlatformClip {
    HRGN                                clipRgn;
} TQAPlatformClip
```

Field descriptions

`clipRgn` A handle to a clipping region.

Generic Device and Clip Structures

QuickDraw 3D RAVE supports one type of device and one type of clipping on generic operating systems. The available device and clipping are defined by unions of type `TQAPlatformDevice` and `TQAPlatformClip`.

```
typedef union TQAPlatformDevice {
    TQADeviceMemory                     memoryDevice;
} TQAPlatformDevice;
```

Field descriptions

`memoryDevice` A memory device data structure.

```
typedef union TQAPlatformClip {
    void                                *region;
} TQAPlatformClip;
```

Field descriptions

`region`

Device Structure

You specify a device (for example, when creating a new draw context with the `QADrawContextNew` function) by filling in a device structure, defined by the `TQADevice` data type.

```
typedef struct TQADevice {
    TQADeviceType                   deviceType;
    TQAPlatformDevice               device;
} TQADevice;
```

Field descriptions

`deviceType`	The device type. See "Device Types" (page 1538) for information on the types of devices that are currently supported.
`device`	A platform device data structure.

Clip Data Structure

You specify a clipping region (for example, when creating a new draw context with the `QADrawContextNew` function) by filling in a clip data structure, defined by the `TQAClip` data type. The clipping region determines which pixels are drawn to a device.

```
typedef struct TQAClip {
    TQAClipType                     clipType;
    TQAPlatformClip                 clip;
} TQAClip;
```

Field descriptions

`clipType`	The clip type. See "Clip Types" (page 1539) for the values you can assign to this field.
`clip`	A platform clip data structure.

Image Structure

Texture maps and bitmaps are defined using pixel images (or pixmaps). To specify a pixel image, you fill in an **image structure,** defined by the `TQAImage` data structure.

```
struct TQAImage {
    long                            width;
    long                            height;
    long                            rowBytes;
```

```
    void                                *pixmap;
};
typedef struct TQAImage TQAImage;
```

Field descriptions

`width` The width, in pixels, of the pixmap.

`height` The height, in pixels, of the pixmap.

`rowBytes` The distance, in bytes, from the beginning of one row of the image data to the beginning of the next row of the image data. (For some low-cost accelerators, setting the value in this field to the product of the value in the `width` field and the pixel size improves performance.)

`pixmap` A pointer to the image data.

Vertex Structures

QuickDraw 3D RAVE supports two different types of vertices: Gouraud vertices and texture vertices. You use **Gouraud vertices** for drawing Gouraud-shaded triangles, and also for drawing points and lines. A Gouraud vertex is defined by the `TQAVGouraud` data structure, which specifies the position, depth, color, and transparency information.

```
typedef struct TQAVGouraud {
    float                               x;
    float                               y;
    float                               z;
    float                               invW;
    float                               r;
    float                               g;
    float                               b;
    float                               a;
} TQAVGouraud;
```

Field descriptions

`x` The *x* coordinate of the vertex relative to the upper-left corner of the draw context rectangle (that is, the rectangle passed to the `QADrawContextNew` function). The value of this field is a floating-point value that specifies a number of pixels.

`y`	The *y* coordinate of the vertex relative to the upper-left corner of the draw context rectangle (that is, the rectangle passed to the `QADrawContextNew` function). The value of this field is a floating-point value that specifies a number of pixels.
`z`	The depth of the vertex. The value of this field is a floating-point number between 0.0 and 1.0 inclusive, where lower numbers specify points closer to the origin.
`invW`	The inverse *w* value (that is, the value $1/w$, where *w* is the homogeneous correction factor). This field is valid only for drawing engines that support the `kQAOptional_PerspectiveZ` feature. When the state variable `kQATag_PerspectiveZ` is set to `kQAPerspectiveZ_On`, hidden surface removal is performed using the value in this field rather than the value in the `z` field, thereby causing the hidden surface removal to be perspective corrected.
`r`	The red component of the vertex color.
`g`	The green component of the vertex color.
`b`	The blue component of the vertex color.
`a`	The alpha channel value of the vertex, where 1.0 represents opacity and 0.0 represents complete transparency.

You use **texture vertices** to define triangles to which a texture is to be mapped. A texture vertex is defined by the `TQAVTexture` data structure, which specifies the position, depth, transparency, and texture mapping information.

Note
Not all the fields of a `TQAVTexture` data structure need to be filled out. Many of these fields are used only when texture mapping operations are in force (that is, when the `kQATag_TextureOp` state variable has some value other than `kQATextureOp_None`). ◆

```
typedef struct TQAVTexture {
    float                               x;
    float                               y;
    float                               z;
    float                               invW;
    float                               r;
    float                               g;
```

```
    float                           b;
    float                           a;
    float                           uOverW;
    float                           vOverW;
    float                           kd_r;
    float                           kd_g;
    float                           kd_b;
    float                           ks_r;
    float                           ks_g;
    float                           ks_b;
} TQAVTexture;
```

Field descriptions

`x` The *x* coordinate of the vertex relative to the upper-left corner of the draw context rectangle (that is, the rectangle passed to the `QADrawContextNew` function). The value of this field is a floating-point value that specifies a number of pixels.

`y` The *y* coordinate of the vertex relative to the upper-left corner of the draw context rectangle (that is, the rectangle passed to the `QADrawContextNew` function). The value of this field is a floating-point value that specifies a number of pixels.

`z` The depth of the vertex. The value of this field is a floating-point number between 0.0 and 1.0 inclusive, where lower numbers specify points closer to the origin.

`invW` The inverse *w* value (that is, the value $1/w$, where *w* is the homogeneous correction factor). This field must contain a value. For drawing engines that support the `kQAOptional_PerspectiveZ` feature and when the state variable `kQATag_PerspectiveZ` is set to `kQAPerspectiveZ_On`, hidden surface removal is performed using the value in this field rather than the value in the `z` field. For non-perspective rendering, this field should be set to 1.0.

`r` The red component of the decal color. The value in this field is used only when the `kQATextureOp_Decal` texture mapping operation is enabled.

`g` The green component of the decal color. The value in this field is used only when the `kQATextureOp_Decal` texture mapping operation is enabled.

`b`	The blue component of the decal color. The value in this field is used only when the `kQATextureOp_Decal` texture mapping operation is enabled.
`a`	The alpha channel value of the vertex, where 1.0 represents opacity and 0.0 represents complete transparency.
`uOverW`	The perspective-corrected u coordinate of the vertex.
`vOverW`	The perspective-corrected v coordinate of the vertex.
`kd_r`	The red component of the diffuse color of the vertex. The value in this field is used only when the `kQATextureOp_Modulate` texture mapping operation is enabled. The value in this field can be greater than 1.0 to more accurately render scenes with high light intensities.
`kd_g`	The green component of the diffuse color of the vertex. The value in this field is used only when the `kQATextureOp_Modulate` texture mapping operation is enabled. The value in this field can be greater than 1.0 to more accurately render scenes with high light intensities.
`kd_b`	The blue component of the diffuse color of the vertex. The value in this field is used only when the `kQATextureOp_Modulate` texture mapping operation is enabled. The value in this field can be greater than 1.0 to more accurately render scenes with high light intensities.
`ks_r`	The red component of the specular color of the vertex. The value in this field is used only when the `kQATextureOp_Highlight` texture mapping operation is enabled.
`ks_g`	The green component of the specular color of the vertex. The value in this field is used only when the `kQATextureOp_Highlight` texture mapping operation is enabled.
`ks_b`	The blue component of the specular color of the vertex. The value in this field is used only when the `kQATextureOp_Highlight` texture mapping operation is enabled.

IMPORTANT

A drawing engine may choose to use a single modulation value instead of the three values `kd_r`, `kd_g`, and `kd_b`. This change is transparent to applications, except that colored lights applied to a texture appear white. As a result, a drawing engine that uses this simplification must negate the `kQAOptional_TextureColor` bit in the optional features value returned by `QAEngineGestalt`. Similarly, a drawing engine may choose to use a single highlight value instead of the three values `ks_r`, `ks_g`, and `ks_b`. This change is transparent to applications, except that a texture-mapped object's specular highlight appears white, not colored. As a result, a drawing engine that uses this simplification must negate the `kQAOptional_TextureColor` bit in the optional features value. ▲

Draw Context Structure

QuickDraw 3D RAVE drawing routines operate on a draw context, which maintains state information and other data associated with a drawing engine. You access a draw context using a **draw context structure,** defined by the `TQADrawContext` data type.

IMPORTANT

You should not directly access the fields of a draw context structure. Instead, you should use the draw context manipulation macros defined by QuickDraw 3D RAVE. See "Manipulating Draw Contexts," beginning on page 1598 for more information. ▲

```
struct TQADrawContext {
    TQADrawPrivate                  *drawPrivate;
    const TQAVersion                version;
    TQASetFloat                     setFloat;
    TQASetInt                       setInt;
    TQASetPtr                       setPtr;
    TQAGetFloat                     getFloat;
    TQAGetInt                       getInt;
    TQAGetPtr                       getPtr;
    TQADrawPoint                    drawPoint;
```

```
    TQADrawLine                     drawLine;
    TQADrawTriGouraud               drawTriGouraud;
    TQADrawTriTexture               drawTriTexture;
    TQADrawVGouraud                 drawVGouraud;
    TQADrawVTexture                 drawVTexture;
    TQADrawBitmap                   drawBitmap;
    TQARenderStart                  renderStart;
    TQARenderEnd                    renderEnd;
    TQARenderAbort                  renderAbort;
    TQAFlush                        flush;
    TQASync                         sync;
    TQASubmitVerticesGouraud        submitVerticesGouraud;
    TQASubmitVerticesTexture        submitVerticesTexture;
    TQADrawTriMeshGouraud           drawTriMeshGouraud;
    TQADrawTriMeshTexture           drawTriMeshTexture;
    TQASetNoticeMethod              setNoticeMethod;
    TQAGetNoticeMethod              getNoticeMethod;
};
typedef struct TQADrawContext TQADrawContext;
```

Field descriptions

`drawPrivate`	A pointer to the private data for the drawing engine associated with this draw context.
`version`	The version of QuickDraw 3D RAVE. This field is initialized when you call `QADrawContextNew`. See "Version Values" (page 1535) for the currently defined version numbers.
`setFloat`	A function pointer to the drawing engine's method for setting floating-point state variables.
`setInt`	A function pointer to the drawing engine's method for setting unsigned long integer state variables.
`setPtr`	A function pointer to the drawing engine's method for setting pointer state variables.
`getFloat`	A function pointer to the drawing engine's method for getting floating-point state variables.
`getInt`	A function pointer to the drawing engine's method for getting unsigned long integer state variables.
`getPtr`	A function pointer to the drawing engine's method for getting pointer state variables.

`drawPoint`	A function pointer to the drawing engine's method for drawing points.
`drawLine`	A function pointer to the drawing engine's method for drawing lines.
`drawTriGouraud`	A function pointer to the drawing engine's method for drawing triangles with Gouraud shading.
`drawTriTexture`	A function pointer to the drawing engine's method for drawing texture-mapped triangles.
`drawVGouraud`	A function pointer to the drawing engine's method for drawing vertices with Gouraud shading.
`drawVTexture`	A function pointer to the drawing engine's method for drawing texture-mapped vertices.
`drawBitmap`	A function pointer to the drawing engine's method for drawing a bitmap.
`renderStart`	A function pointer to the drawing engine's method for initializing in preparation for rendering.
`renderEnd`	A function pointer to the drawing engine's method for completing a rendering operation and displaying an image.
`renderAbort`	A function pointer to the drawing engine's method for canceling the current rendering operation and flushing any queued operations.
`flush`	A function pointer to the drawing engine's method for starting to render all queued drawing commands.
`sync`	A function pointer to the drawing engine's method for waiting until all queued drawing commands have been processed.
`submitVerticesGouraud`	A function pointer to the drawing engine's method for submitting Gouraud vertices.
`submitVerticesTexture`	A function pointer to the drawing engine's method for submitting texture vertices.
`drawTriMeshGouraud`	A function pointer to the drawing engine's method for drawing triangle meshes with Gouraud shading.
`drawTriMeshTexture`	A function pointer to the drawing engine's method for drawing texture-mapped triangle meshes.

`setNoticeMethod`	A function pointer to the drawing engine's method for setting a notice method.
`getNoticeMethod`	A function pointer to the drawing engine's method for getting a notice method.

Indexed Triangle Structure

The `QADrawTriMeshGouraud` and `QADrawTriMeshTexture` functions draw triangle meshes defined by an array of indexed triangles. An indexed triangle is represented by a data structure of type `TQAIndexedTriangle` that defines three vertices and a set of triangle flags.

```
typedef struct TQAIndexedTriangle {
    unsigned long                       triangleFlags;
    unsigned long                       vertices[3];
} TQAIndexedTriangle;
```

Field descriptions

`triangleFlags`	A set of triangle flags. See "Triangle Flags Masks," beginning on page 1566 for a complete description of the available flags.
`vertices`	An array of three indices into the array of vertices submitted by the most recent call to `QASubmitVerticesGouraud` or `QASubmitVerticesTexture`.

QuickDraw 3D RAVE Routines

This section describes the routines provided by QuickDraw 3D RAVE.

Creating and Deleting Draw Contexts

QuickDraw 3D RAVE provides routines that you can use to create and delete draw contexts.

QADrawContextNew

You can use the `QADrawContextNew` function to create a new draw context.

```
TQAError QADrawContextNew (
                    const TQADevice *device,
                    const TQARect *rect,
                    const TQAClip *clip,
                    const TQAEngine *engine,
                    unsigned long flags,
                    TQADrawContext **newDrawContext);
```

`device` A device.

`rect` The rectangular region (specified in device coordinates) of the specified device that can be drawn into by the drawing engine associated with the new draw context.

`clip` The two-dimensional clipping region for the new draw context, or `NULL` if no clipping is desired. This parameter must be set to `NULL` for devices of type `kQADeviceMemory`.

`engine` A drawing engine.

`flags` A set of bit flags specifying features of the new draw context. See "Draw Context Flags Masks" (page 1567) for complete information.

`newDrawContext`
On entry, the address of a pointer variable. On exit, that variable points to a new draw context. If a new draw context cannot be created, `*newDrawContext` is set to the value `NULL`.

DESCRIPTION

The `QADrawContextNew` function returns, through the `newDrawContext` parameter, a new draw context associated with the device specified by the `device` parameter and the drawing engine specified by the `engine` parameter.

QADrawContextDelete

You can use the `QADrawContextDelete` function to delete a draw context.

```
void QADrawContextDelete (TQADrawContext *drawContext);
```

`drawContext` A draw context.

DESCRIPTION

The `QADrawContextDelete` function deletes the draw context specified by the `drawContext` parameter. Any memory and other resources associated with that draw context are released.

Creating and Deleting Color Lookup Tables

QuickDraw 3D RAVE provides routines that you can use to create and dispose of color lookup tables.

QAColorTableNew

You can use the `QAColorTableNew` function to create a new color lookup table.

```
TQAError QAColorTableNew (
                    const TQAEngine *engine,
                    TQAColorTableType tableType,
                    void *pixelData,
                    long transparentIndexFlag,
                    TQAColorTable **newTable);
```

`engine` A drawing engine.

`tableType` The type of the new color lookup table. See “Color Lookup Table Types” (page 1538) for information on the available color lookup table types.

`pixelData` A pointer to the color lookup table entries.

`transparentIndexFlag`
A long integer, interpreted as a Boolean value, that indicates whether the color lookup table entry at index 0 is completely transparent (`TRUE`) or not (`FALSE`).

`newTable` On entry, the address of a pointer variable. On exit, that variable points to a new color lookup table. If a new color lookup table cannot be created, `*newTable` is set to the value `NULL`.

DESCRIPTION

The `QAColorTableNew` function returns, through the `newTable` parameter, a new color lookup table associated with the drawing engine specified by the `engine` parameter. The table entries for the new color lookup table are copied from the block of data pointed to by the `pixelData` parameter; if `QAColorTableNew` completes successfully, you can dispose of that block of memory. The data in that block of memory is interpreted according to the format specified by the `tableType` parameter. For example, if `tableType` is `kQAColorTable_CL8_RGB32`, then `pixelData` should point to a block of data that is at least 1024 bytes long and in which each 32-bit quantity is an RGB color value.

IMPORTANT

Currently, QuickDraw 3D RAVE supports only 32-bit RGB color lookup table entries. The specified drawing engine might reduce the size of individual color lookup table entries to fit into its on-board memory. ▲

Not all drawing engines support color lookup tables, and QuickDraw 3D RAVE does not provide color lookup table emulation for engines that do not support them.

SEE ALSO

Use the `QAColorTableDelete` function (next) to delete a color lookup table. Use the `QATextureBindColorTable` function page 1590 to bind a color lookup table to a texture map. Use the `QABitmapBindColorTable` function page 1593 to bind a color lookup table to a bitmap.

QAColorTableDelete

You can use the `QAColorTableDelete` function to delete a color lookup table.

```
void QAColorTableDelete (
                    const TQAEngine *engine,
                    TQAColorTable *colorTable);
```

`engine` A drawing engine.

`colorTable` A color lookup table.

DESCRIPTION

The `QAColorTableDelete` function deletes the color lookup table specified by the `colorTable` parameter. Any memory and other resources associated with that color lookup table are released.

SEE ALSO

Use the `QAColorTableNew` function page 1586 to create a color lookup table.

Manipulating Textures and Bitmaps

QuickDraw 3D RAVE provides routines that you can use to create and dispose of texture maps and bitmaps. It also provides routines that you can use to bind color lookup tables to texture maps and bitmaps.

QATextureNew

You can use the `QATextureNew` function to create a new texture map.

```
TQAError QATextureNew (
                    const TQAEngine *engine,
                    unsigned long flags,
                    TQAImagePixelType pixelType,
                    const TQAImage images[],
                    TQATexture **newTexture);
```

`engine`	A drawing engine.
`flags`	A set of bit flags specifying features of the new texture map. See "Texture Flags Masks" (page 1566) for complete information.
`pixelType`	The type of pixels in the new texture map. See "Pixel Types" (page 1536) for a description of the values you can pass in this parameter.
`images`	An array of pixel images to use for the new texture map. The values in the `width` and `height` fields of these structures must be an even power of 2.
`newTexture`	On entry, the address of a pointer variable. On exit, that variable points to a new texture map. If a new texture map cannot be created, `*newTexture` is set to the value `NULL`.

DESCRIPTION

The `QATextureNew` function returns, through the `newTexture` parameter, a new texture map associated with the drawing engine specified by the `engine` parameter. You can use the returned texture map to set the value of the `kQATag_Texture` state variable.

The `flags` parameter specifies a set of texture map features. If the `kQATexture_Lock` bit in that parameter is set but the drawing engine cannot guarantee that the texture will remain locked in memory, the `QATextureNew` function returns an error.

If the `kQATexture_Mipmap` bit of the `flags` parameter is clear, the `images` parameter points to a single pixel image that defines the texture map. If the `kQATexture_Mipmap` bit is set, the `images` parameter points to an array of pixel images of varying pixel depths. The first element in the array must be the mipmap page having the highest resolution, with a width and height that are even powers of 2. Each subsequent pixel image in the array should have a width and height that are half those of the previous image (with a minimum width and height of 1).

SPECIAL CONSIDERATIONS

`QATextureNew` does not automatically copy the pixmap data pointed to by the `images` parameter. As a result, you should not release or reuse the storage occupied by the pixel images until you've called `QATextureDetach`. Note,

however, that QATextureNew does copy all of the information contained in the TQAImage structures in the array, so you can free or reuse that memory after QATextureNew completes successfully.

QATextureDetach

You can use the QATextureDetach function to detach a texture map from a drawing engine.

```
TQAError QATextureDetach (const TQAEngine *engine, TQATexture *texture);
```

engine	A drawing engine.
texture	A texture map.

DESCRIPTION

The QATextureDetach function causes the drawing engine specified by the engine parameter to copy the data associated with the texture map specified by the texture parameter. Once the data are copied, you can reuse or dispose of the memory you originally specified in a call to QATextureNew.

QATextureBindColorTable

You can use the QATextureBindColorTable function to bind a color lookup table to a texture map.

```
TQAError QATextureBindColorTable (
                    const TQAEngine *engine,
                    TQATexture *texture,
                    TQAColorTable *colorTable);
```

engine	A drawing engine.
texture	A texture map.
colorTable	A color lookup table (as returned by a previous call to QAColorTableNew).

DESCRIPTION

The `QATextureBindColorTable` function binds the color lookup table specified by the `colorTable` parameter to the texture map specified by the `texture` parameter. Before you can draw any texture map whose pixel type is either `kQAPixel_CL4` or `kQAPixel_CL8`, you must bind a color lookup table to it. In addition, the type of the specified color lookup table must match that of the pixel type of the texture map to which it is bound. For example, a color lookup table of type `kQAColorTable_CL8_RGB32` can be bound only to a texture map whose pixel type is `kQAPixel_CL8`.

QATextureDelete

You can use the `QATextureDelete` function to delete a texture map.

```
void QATextureDelete (const TQAEngine *engine, TQATexture *texture);
```

`engine` A drawing engine.

`texture` A texture map.

DESCRIPTION

The `QATextureDelete` function deletes the texture map specified by the `texture` parameter from the drawing engine specified by the `engine` parameter.

QABitmapNew

You can use the `QABitmapNew` function to create a new bitmap.

```
TQAError QABitmapNew (
                    const TQAEngine *engine,
                    unsigned long flags,
                    TQAImagePixelType pixelType,
                    const TQAImage *image,
                    TQABitmap **newBitmap);
```

`engine`	A drawing engine.
`flags`	A set of bit flags specifying features of the new bitmap. See "Bitmap Flags Masks" (page 1567) for complete information
`pixelType`	The type of pixels in the new bitmap. See "Pixel Types" (page 1536) for a description of the values you can pass in this parameter.
`image`	A pixel image to use for the new bitmap. The `width` and `height` fields of this image can have any values greater than 0.
`newBitmap`	On entry, the address of a pointer variable. On exit, that variable points to a new bitmap. If a new bitmap cannot be created, `*newBitmap` is set to the value `NULL`.

DESCRIPTION

The `QABitmapNew` function returns, through the `newBitmap` parameter, a pointer to a new bitmap associated with the drawing engine specified by the `engine` parameter. You can draw the returned bitmap by calling the `QADrawBitmap` function.

The `flags` parameter specifies a set of bitmap features. If the `kQABitmap_Lock` bit in that parameter is set but the drawing engine cannot guarantee that the bitmap will remain locked in memory, the `QABitmapNew` function returns an error.

SPECIAL CONSIDERATIONS

`QABitmapNew` does not automatically copy the pixmap data pointed to by the `images` parameter. As a result, you should not release or reuse the storage occupied by the pixel image until you've called `QABitmapDetach`. Note, however, that `QABitmapNew` does copy all of the information contained in the `TQAImage` structure, so you can free or reuse that memory after `QABitmapNew` completes successfully.

QABitmapDetach

You can use the `QABitmapDetach` function to detach a bitmap from a drawing engine.

```
TQAError QABitmapDetach (const TQAEngine *engine, TQABitmap *bitmap);
```

`engine` A drawing engine.

`bitmap` A bitmap.

DESCRIPTION

The `QABitmapDetach` function causes the drawing engine specified by the `engine` parameter to copy the data associated with the bitmap specified by the `bitmap` parameter. Once the data are copied, you can reuse or dispose of the memory you originally specified in a call to `QABitmapNew`.

QABitmapBindColorTable

You can use the `QABitmapBindColorTable` function to bind a color lookup table to a bitmap.

```
TQAError QABitmapBindColorTable (
                    const TQAEngine *engine,
                    TQABitmap *bitmap,
                    TQAColorTable *colorTable);
```

`engine` A drawing engine.

`bitmap` A bitmap.

`colorTable` A color lookup table (as returned by a previous call to `QAColorTableNew`).

DESCRIPTION

The `QABitmapBindColorTable` function binds the color lookup table specified by the `colorTable` parameter to the bitmap specified by the `bitmap` parameter.

Before you can draw any bitmap whose pixel type is either `kQAPixel_CL4` or `kQAPixel_CL8`, you must bind a color lookup table to it. In addition, the type of the specified color lookup table must match that of the pixel type of the bitmap to which it is bound. For example, a color lookup table of type `kQAColorTable_CL8_RGB32` can be bound only to a bitmap whose pixel type is `kQAPixel_CL8`.

QABitmapDelete

You can use the `QABitmapDelete` function to delete a bitmap.

```
void QABitmapDelete (const TQAEngine *engine, TQABitmap *bitmap);
```

`engine` A drawing engine.

`bitmap` A bitmap.

DESCRIPTION

The `QABitmapDelete` function deletes the bitmap specified by the `bitmap` parameter from the drawing engine specified by the `engine` parameter.

Managing Drawing Engines

QuickDraw 3D RAVE provides routines that you can use to manage drawing engines. For example, you can use these routines to find a drawing engine for a particular device.

QADeviceGetFirstEngine

You can use the `QADeviceGetFirstEngine` function to get the first drawing engine that can draw to a particular device.

```
TQAEngine *QADeviceGetFirstEngine (const TQADevice *device);
```

`device` A device, or the value `NULL`.

DESCRIPTION

The `QADeviceGetFirstEngine` function returns, as its function result, the first drawing engine that is capable of drawing into the device specified by the `device` parameter. The first engine is defined to be the first engine that satisfies one of these criteria:

1. The drawing engine selected by the user (for example, using the RAVE control panel).
2. The drawing engine that is tightly coupled to the specified device (that is, that can render only to that device).
3. The drawing engine that accelerates more features than any other drawing engine.

If you pass the value `NULL` in the `device` parameter, `QADeviceGetFirstEngine` returns a drawing engine without regard for its ability to drive any particular device. You can use this technique to find all available engines.

QADeviceGetNextEngine

You can use the `QADeviceGetNextEngine` function to get the next drawing engine that can draw to a particular device.

```
TQAEngine *QADeviceGetNextEngine (
                    const TQADevice *device,
                    const TQAEngine *currentEngine);
```

`device` A device, or the value `NULL`.

`currentEngine` A drawing engine.

DESCRIPTION

The `QADeviceGetNextEngine` function returns, as its function result, the drawing engine that supports the device specified by the `device` parameter that follows the engine specified by the `currentEngine` parameter. The value you pass in the `currentEngine` parameter should have been obtained from a previous call to `QADeviceGetFirstEngine` or `QADeviceGetNextEngine`.

If you pass the value NULL in the device parameter, QADeviceGetNextEngine returns a the next drawing engine without regard for its ability to drive any particular device. You can use this technique to find all available engines.

QAEngineCheckDevice

You can use the QAEngineCheckDevice function to determine whether a particular drawing engine can draw into a particular device.

```
TQAError QAEngineCheckDevice (
                    const TQAEngine *engine,
                    const TQADevice *device);
```

engine A drawing engine.

device A device.

DESCRIPTION

The QAEngineCheckDevice function returns, as its function result, a code that indicates whether the drawing engine specified by the engine parameter can draw into the device specified by the device parameter (kQANoErr) or not (kQAError).

QAEngineGestalt

You can use the QAEngineGestalt function to get information about a drawing engine.

```
TQAError QAEngineGestalt (
                    const TQAEngine *engine,
                    TQAGestaltSelector selector,
                    void *response);
```

engine A drawing engine.

`selector`	A selector that determines what kind of information is to be returned about the specified drawing engine. See “Gestalt Selectors” (page 1559) for complete information about the available selectors and the information they return.
`response`	A pointer to a buffer into which the returned information is copied. Your application is responsible for allocating this buffer. The size and meaning of the data copied to the buffer depend on the selector you pass in the `selector` parameter.

DESCRIPTION

The `QAEngineGestalt` function returns, in the `response` parameter, a buffer of information about features of the type specified by the `selector` parameter associated with the drawing engine specified by the `engine` parameter.

SEE ALSO

See “Finding a Drawing Engine” (page 1516) for code illustrating how to call `QAEngineGestalt`.

QAEngineEnable

You can use the `QAEngineEnable` function to enable a drawing engine.

```
TQAError QAEngineEnable (long vendorID, long engineID);
```

`vendorID`	A vendor ID.
`engineID`	A drawing engine ID.

DESCRIPTION

The `QAEngineEnable` function enables the drawing engine specified by the `vendorID` and `engineID` parameters.

QAEngineDisable

You can use the `QAEngineDisable` function to disable a drawing engine.

```
TQAError QAEngineDisable (long vendorID, long engineID);
```

`vendorID` A vendor ID.

`engineID` An engine ID.

DESCRIPTION

The `QAEngineDisable` function disables the drawing engine specified by the `vendorID` and `engineID` parameters.

Manipulating Draw Contexts

QuickDraw 3D RAVE provides routines that you can use to manipulate draw contexts. For example, you can use the `QASetInt` routine to set an integer-valued state variable associated with a draw context.

IMPORTANT

These functions are currently implemented as C language macros that call the methods of the drawing engine. Your application should use these macros for all draw context manipulation. ▲

See the section "Application-Defined Routines," beginning on page 1618 for complete information on the draw context methods invoked by these macros.

Note

There is one macro for each method whose address is stored in a draw context structure (of type `TQADrawContext`). ◆

QAGetFloat

You can use the `QAGetFloat` function to get a floating-point value of a draw context state variable.

```
#define QAGetFloat(drawContext,tag) \
                (drawContext)->getFloat (drawContext,tag)
```

`drawContext`	A draw context.
`tag`	A state variable tag.

DESCRIPTION

The `QAGetFloat` function returns, as its function result, the floating-point value of the draw context state variable specified by the `drawContext` and `tag` parameters. If the specified tag is not recognized or supported by that draw context, `QAGetFloat` returns the value 0.

QASetFloat

You can use the `QASetFloat` function to set a floating-point value for a draw context state variable.

```
#define QASetFloat(drawContext,tag,newValue) \
                (drawContext)->setFloat (drawContext,tag,newValue)
```

`drawContext`	A draw context.
`tag`	A state variable tag.
`newValue`	The new value of the specified state variable.

DESCRIPTION

The `QASetFloat` function sets the value of the draw context state variable specified by the `drawContext` and `tag` parameters to the floating-point value specified by the `newValue` parameter.

QAGetInt

You can use the `QAGetInt` function to get a long integer value of a draw context state variable.

```
#define QAGetInt(drawContext,tag) \
                (drawContext)->getInt (drawContext,tag)
```

`drawContext`	A draw context.
`tag`	A state variable tag.

DESCRIPTION

The `QAGetInt` function returns, as its function result, the long integer value of the draw context state variable specified by the `drawContext` and `tag` parameters. If the specified tag is not recognized or supported by that draw context, `QAGetInt` returns the value 0.

QASetInt

You can use the `QASetInt` function to set a long integer value for a draw context state variable.

```
#define QASetInt(drawContext,tag,newValue) \
                (drawContext)->setInt (drawContext,tag,newValue)
```

`drawContext`	A draw context.
`tag`	A state variable tag.
`newValue`	The new value of the specified state variable.

DESCRIPTION

The `QASetInt` function sets the value of the draw context state variable specified by the `drawContext` and `tag` parameters to the long integer value specified by the `newValue` parameter.

QAGetPtr

You can use the `QAGetPtr` function to get a pointer value of a draw context state variable.

```
#define QAGetPtr(drawContext,tag) \
                (drawContext)->getPtr (drawContext,tag)
```

`drawContext`	A draw context.
`tag`	A state variable tag.

DESCRIPTION

The `QAGetPtr` function returns, as its function result, the pointer value of the draw context state variable specified by the `drawContext` and `tag` parameters. If the specified tag is not recognized or supported by that draw context, `QAGetPtr` returns the value 0.

QASetPtr

You can use the `QASetPtr` function to set a pointer value for a draw context state variable.

```
#define QASetPtr(drawContext,tag,newValue) \
                (drawContext)->setPtr (drawContext,tag,newValue)
```

`drawContext`	A draw context.
`tag`	A state variable tag.
`newValue`	The new value of the specified state variable.

DESCRIPTION

The `QASetPtr` function sets the value of the draw context state variable specified by the `drawContext` and `tag` parameters to the pointer value specified by the `newValue` parameter.

QADrawPoint

You can use the `QADrawPoint` function to draw a point.

```
#define QADrawPoint(drawContext,v) \
                (drawContext)->drawPoint (drawContext,v)
```

`drawContext` A draw context.

`v` A Gouraud vertex.

DESCRIPTION

The `QADrawPoint` function draws the single point specified by the `v` parameter to the draw context specified by the `drawContext` parameter. The size of the point is determined by the `kQATag_Width` state variable of the draw context.

QADrawLine

You can use the `QADrawLine` function to draw a line between two points.

```
#define QADrawLine(drawContext,v0,v1) \
                (drawContext)->drawLine (drawContext,v0,v1)
```

`drawContext` A draw context.

`v0` A Gouraud vertex.

`v1` A Gouraud vertex.

DESCRIPTION

The `QADrawLine` function draws the line specified by the `v0` and `v1` parameters to the draw context specified by the `drawContext` parameter. The size of the line is determined by the `kQATag_Width` state variable of the draw context. If the specified vertices have different colors, the line color is interpolated smoothly between the two vertex colors.

QADrawTriGouraud

You can use the `QADrawTriGouraud` function to draw Gouraud-shaded triangles.

```
#define QADrawTriGouraud(drawContext,v0,v1,v2,flags) \
        (drawContext)->drawTriGouraud (drawContext,v0,v1,v2,flags)
```

`drawContext`	A draw context.
`v0`	A Gouraud vertex.
`v1`	A Gouraud vertex.
`v2`	A Gouraud vertex.
`flags`	A set of triangle flags. See “Triangle Flags Masks,” beginning on page 1566 for a complete description of the available flags.

DESCRIPTION

The `QADrawTriGouraud` function draws the Gouraud-shaded triangle determined by the three points specified by the `v0`, `v1`, and `v2` parameters into the draw context specified by the `drawContext` parameter. Features of the triangle are determined by the `flags` parameter. Currently, this parameter is used to specify an orientation for the triangle.

QADrawTriTexture

You can use the `QADrawTriTexture` function to draw texture-mapped triangles.

```
#define QADrawTriTexture(drawContext,v0,v1,v2,flags) \
        (drawContext)->drawTriTexture (drawContext,v0,v1,v2,flags)
```

`drawContext`	A draw context.
`v0`	A texture vertex.
`v1`	A texture vertex.
`v2`	A texture vertex.

`flags`	A set of triangle flags. See "Triangle Flags Masks," beginning on page 1566 for a complete description of the available flags.

DESCRIPTION

The `QADrawTriTexture` function draws the texture-mapped triangle determined by the three points specified by the `v0`, `v1`, and `v2` parameters into the draw context specified by the `drawContext` parameter. The texture used for the mapping is determined by the value of the `kQATag_Texture` state variable. Features of the triangle are determined by the `flags` parameter. Currently, this parameter is used to specify an orientation for the triangle.

SPECIAL CONSIDERATIONS

The `QADrawTriTexture` function is optional and must be supported only by drawing engines that support texture mapping.

QASubmitVerticesGouraud

You can use the `QASubmitVerticesGouraud` function to submit Gouraud vertices.

```
#define QASubmitVerticesGouraud(drawContext,nVertices,vertices) \
    (drawContext)->submitVerticesGouraud(drawContext,nVertices,vertices)
```

`drawContext`	A draw context.
`nVertices`	The number of Gouraud vertices pointed to by the `vertices` parameter.
`vertices`	A pointer to an array of Gouraud vertices.

DESCRIPTION

The `QASubmitVerticesGouraud` function submits the list of vertices pointed to by the `vertices` parameter to the draw context specified by the `drawContext` parameter. The vertices define a triangle mesh. Note, however, that `QASubmitVerticesGouraud` does not draw the specified mesh, but simply defines the mesh for a subsequent call to `QADrawTriMeshGouraud`.

Your application is responsible for managing the memory occupied by the Gouraud vertices. `QASubmitVerticesGouraud` does not copy the vertex data pointed to by the `vertices` parameter. Accordingly, you must not dispose of or reuse that memory until you've finished drawing the triangle mesh defined by `QASubmitVerticesGouraud`.

SPECIAL CONSIDERATIONS

If a drawing engine does not support triangle meshes, QuickDraw 3D RAVE decomposes a triangle mesh into individual triangles. As a result, you can always use the `QASubmitVerticesGouraud` function to submit a triangle mesh.

QASubmitVerticesTexture

You can use the `QASubmitVerticesTexture` function to submit texture vertices.

```
#define QASubmitVerticesTexture(drawContext,nVertices,vertices) \
    (drawContext)->submitVerticesTexture(drawContext,nVertices,vertices)
```

`drawContext`	A draw context.
`nVertices`	The number of texture vertices pointed to by the `vertices` parameter.
`vertices`	A pointer to an array of texture vertices.

DESCRIPTION

The `QASubmitVerticesTexture` function submits the list of vertices pointed to by the `vertices` parameter to the draw context specified by the `drawContext` parameter. The vertices define a triangle mesh. Note, however, that `QASubmitVerticesTexture` does not draw the specified mesh, but simply defines the mesh for a subsequent call to `QADrawTriMeshTexture`.

Your application is responsible for managing the memory occupied by the texture vertices. `QASubmitVerticesTexture` does not copy the vertex data pointed to by the `vertices` parameter. Accordingly, you must not dispose of or reuse that memory until you've finished drawing the triangle mesh defined by `QASubmitVerticesTexture`.

SPECIAL CONSIDERATIONS

The `QASubmitVerticesTexture` function is optional and must be supported only by drawing engines that support texture mapping.

If a drawing engine does not support triangle meshes, QuickDraw 3D RAVE decomposes a triangle mesh into individual triangles.

QADrawTriMeshGouraud

You can use the `QADrawTriMeshGouraud` function to draw a triangle mesh with Gouraud shading.

```
#define QADrawTriMeshGouraud(drawContext,nTriangle,triangles) \
    (drawContext)->drawTriMeshGouraud (drawContext,nTriangle,triangles)
```

`drawContext`	A draw context.
`nTriangle`	The number of indexed triangles pointed to by the `triangles` parameter.
`triangles`	A pointer to an array of indexed triangles. See “Indexed Triangle Structure” (page 1584) for a description of indexed triangles.

DESCRIPTION

The `QADrawTriMeshGouraud` function draws, with Gouraud shading, the triangle mesh specified by the `triangles` parameter into the draw context specified by the `drawContext` parameter. Each triangle in the mesh is defined by a `TQAIndexedTriangle` data structure, which contains three indices into the array of Gouraud vertices previously submitted to the draw context by a call to the `QASubmitVerticesGouraud` function.

SPECIAL CONSIDERATIONS

`QADrawTriMeshGouraud` operates only on a triangle mesh previously submitted using the `QASubmitVerticesGouraud` function. Use `QADrawTriMeshTexture` to draw a triangle mesh submitted using the `QASubmitVerticesTexture` function.

QADrawTriMeshTexture

You can use the `QADrawTriMeshTexture` function to draw a texture-mapped triangle mesh.

```
#define QADrawTriMeshTexture(drawContext,nTriangle,triangles) \
    (drawContext)->drawTriMeshTexture (drawContext,nTriangle,triangles)
```

`drawContext`	A draw context.
`nTriangle`	The number of indexed triangles pointed to by the `triangles` parameter.
`triangles`	A pointer to an array of indexed triangles. See "Indexed Triangle Structure" (page 1584) for a description of indexed triangles.

DESCRIPTION

The `QADrawTriMeshTexture` function draws the texture-mapped triangle mesh specified by the `triangles` parameter into the draw context specified by the `drawContext` parameter. Each triangle in the mesh is defined by a `TQAIndexedTriangle` data structure, which contains three indices into the array of texture vertices previously submitted to the draw context by a call to the `QASubmitVerticesTexture` function.

SPECIAL CONSIDERATIONS

`QADrawTriMeshTexture` operates only on a triangle mesh previously submitted using the `QASubmitVerticesTexture` function. Use `QADrawTriMeshGouraud` to draw a triangle mesh submitted using the `QASubmitVerticesGouraud` function.

The `QADrawTriMeshTexture` function is optional and must be supported only by drawing engines that support texture mapping.

QADrawVGouraud

You can use the `QADrawVGouraud` function to draw Gouraud-shaded objects defined by vertices.

```
#define QADrawVGouraud(drawContext,nVertices,vertexMode,vertices,flags) \
 (drawContext)->drawVGouraud(drawContext,nVertices,vertexMode,vertices,flags)
```

`drawContext`	A draw context.
`nVertices`	The number of vertices contained in the `vertices` array.
`vertexMode`	A vertex mode. See "Vertex Modes" (page 1558) for a description of the available vertex modes.
`vertices`	An array of Gouraud vertices.
`flags`	An array of triangle flags, or the value `NULL`. See "Triangle Flags Masks" (page 1566) for a description of the available triangle flags. This parameter is valid only if the `vertexMode` parameter contains the value `kQAVertexMode_Tri`, `kQAVertexMode_Strip`, or `kQAVertexMode_Fan`.

DESCRIPTION

The `QADrawVGouraud` function draws the vertices in the array specified by the `vertices` parameter into the draw context specified by the `drawContext` parameter, according to the vertex modes flag specified by the `vertexMode` parameter. For instance, if the value of the `vertexMode` parameter is `kQAVertexMode_Polyline`, then the vertices in that array are interpreted as defining a polyline (a set of connected line segments). Gouraud shading is applied to whatever objects are drawn.

SPECIAL CONSIDERATIONS

The `QADrawVGouraud` function is optional and must be supported only by drawing engines that do not want calls to `QADrawVGouraud` to be replaced by calls to the `QADrawPoint`, `QADrawLine`, or `QADrawTriGouraud` functions.

QADrawVTexture

You can use the `QADrawVTexture` function to draw texture-mapped objects defined by vertices.

```
#define QADrawVTexture(drawContext,nVertices,vertexMode,vertices,flags) \
 (drawContext)->drawVTexture(drawContext,nVertices,vertexMode,vertices,flags)
```

`drawContext`	A draw context.
`nVertices`	The number of vertices contained in the `vertices` array.
`vertexMode`	A vertex mode. See "Vertex Modes" (page 1558) for a description of the available vertex modes.
`vertices`	An array of texture vertices.
`flags`	An array of triangle flags, or the value `NULL`. See "Triangle Flags Masks" (page 1566) for a description of the available triangle flags. This parameter is valid only if the `vertexMode` parameter contains the value `kQAVertexMode_Tri`, `kQAVertexMode_Strip`, or `kQAVertexMode_Fan`.

DESCRIPTION

The `QADrawVTexture` function draws the vertices in the array specified by the `vertices` parameter into the draw context specified by the `drawContext` parameter, according to the vertex modes flag specified by the `vertexMode` parameter. For instance, if the value of the `vertexMode` parameter is `kQAVertexMode_Polyline`, then the vertices in that array are interpreted as defining a polyline (a set of connected line segments). Texture mapping (using the texture determined by the value of the `kQATag_Texture` state variable) is applied to whatever objects are drawn.

IMPORTANT

The vertex modes `kQAVertexMode_Point` and `kQAVertexMode_Line` are supported only by drawing engines that support the `kQAOptional_OpenGL` feature. All other drawing engines should ignore requests to texture map points or lines. ▲

SPECIAL CONSIDERATIONS

The `QADrawVTexture` function is optional and must be supported only by drawing engines that support texture mapping and do not want calls to `QADrawVTexture` to be replaced by calls to the `QADrawPoint`, `QADrawLine`, or `QADrawTriTexture` methods.

QADrawBitmap

You can use the `QADrawBitmap` function to draw bitmaps into a draw context.

```
#define QADrawBitmap(drawContext,v,bitmap) \
                    (drawContext)->drawBitmap (drawContext,v,bitmap)
```

`drawContext` A draw context.

`v` A Gouraud vertex.

`bitmap` A pointer to a bitmap (returned by a previous call to `QABitmapNew`).

DESCRIPTION

The `QADrawBitmap` function draws the bitmap specified by the `bitmap` parameter into the draw context specified by the `drawContext` parameter, with the upper-left corner of the bitmap located at the point specified by the `v` parameter. The `v` parameter can contain negative values in its `x` or `y` fields, so you can position upper-left corner of the bitmap outside the draw context rectangle. This allows you to move the bitmap smoothly off any edge of the draw context.

QARenderStart

You can use the `QARenderStart` function to initialize a draw context before an engine performs any rendering into that context.

```
#define QARenderStart(drawContext,dirtyRect,initialContext) \
(drawContext)->renderStart (drawContext,dirtyRect,initialContext)
```

`drawContext` A draw context.

`dirtyRect` The minimum area of the specified draw context to clear, or the value `NULL`.

`initialContext`
A previously cached draw context, or the value `NULL`.

DESCRIPTION

The `QARenderStart` function performs any operations necessary to initialize the draw context specified by the `drawContext` parameter. This includes clearing the z buffer and the color buffers of the draw context. If the value of the `initialContext` parameter is `NULL`, then `QARenderStart` clears the z buffer to 1.0 and sets the color buffers to the values of the `kQATag_ColorBG_a`, `kQATag_ColorBG_r`, `kQATag_ColorBG_g`, and `kQATag_ColorBG_b` draw context state variables. If, however, the value of the `initialContext` parameter is not `NULL`, then `QARenderStart` uses the previously cached draw context specified by that parameter to initialize the draw context specified by the `drawContext` parameter.

The `dirtyRect` parameter indicates the minimum area, in local coordinates of the draw context, of the specified draw context to clear on initialization. If the value of the `dirtyRect` parameter is `NULL`, the entire draw context is cleared. If the value of the `dirtyRect` parameter is not `NULL`, it indicates the rectangle in the draw context to clear. Some drawing engines may exhibit improved performance when an area that is smaller than the entire draw context rectangle is passed. However, the interpretation of the `dirtyRect` parameter is dependent on the drawing engine, which may choose to initialize the entire draw context. As a result, you should not use this parameter as a means to avoid clearing all of a draw context or to perform incremental rendering. Instead, you should use the `initialContext` parameter to achieve such effects.

SPECIAL CONSIDERATIONS

You should call `QARenderStart` before performing any rendering operations in the specified draw context, and you should call either `QARenderEnd` to signal the end of rendering operations or `QARenderAbort` to cancel rendering operations. However, when a drawing engine is performing OpenGL rendering, the `QARenderStart` function operates just like the OpenGL function `glClear`. In OpenGL mode, it is not necessary that a call to `QARenderStart` always be balanced by a matching call to `QARenderEnd`, and drawing commands may occur at any time.

SEE ALSO

See “Using a Draw Context as a Cache” (page 1520) for information on creating a draw context cache (that is, a draw context you can use as the initial context specified in the `initialContext` parameter).

QARenderEnd

You can use the `QARenderEnd` function to signal the end of any rendering into a draw context.

```
#define QARenderEnd(drawContext,modifiedRect) \
                (drawContext)->renderEnd (drawContext,modifiedRect)
```

`drawContext` A draw context.

`modifiedRect` The minimum area of the back buffer of the specified draw context to display, or the value `NULL`.

DESCRIPTION

The `QARenderEnd` function performs any operations necessary to display an image rendered into the draw context specified by the `drawContext` parameter. If the draw context is double buffered, `QARenderEnd` displays the back buffer. If the draw context is single buffered, `QARenderEnd` calls `QAFlush`.

The `modifiedRect` parameter indicates the minimum area of the back buffer of the specified draw context that should be displayed. If the value of the `modifiedRect` parameter is `NULL`, the entire back buffer is displayed. If the value of the `modifiedRect` parameter is not `NULL`, it indicates the rectangle in the back buffer to display. Some drawing engines may exhibit improved performance when an area that is smaller than the entire draw context rectangle is passed (to avoid unnecessary pixel copying). However, the interpretation of the `modifiedRect` parameter is dependent on the drawing engine, which may choose to draw the entire back buffer.

The `QARenderEnd` function returns a result code (of type `TQAError`) indicating whether any errors have occurred since the previous call to `QARenderStart`. If all rendering commands completed successfully, the value `kQANoErr` is returned. If any other value is returned, you should assume that the rendered image is incorrect.

SPECIAL CONSIDERATIONS

You should call `QARenderStart` before performing any rendering operations in the specified draw context, and you should call either `QARenderEnd` to signal the end of rendering operations or `QARenderAbort` to cancel rendering operations.

Once you have called `QARenderEnd`, you should not submit any drawing requests until you have called `QARenderStart` again.

QARenderAbort

You can use the `QARenderAbort` function to cancel any asynchronous drawing requests for a draw context.

```
#define QARenderAbort(drawContext) \
                        (drawContext)->renderAbort (drawContext)
```

`drawContext` A draw context.

DESCRIPTION

The `QARenderAbort` function immediately stops the draw context specified by the `drawContext` parameter from processing any asynchronous drawing commands it is currently processing and causes it to discard any queued commands.

The `QARenderAbort` function returns a result code (of type `TQAError`) indicating whether any errors have occurred since the previous call to `QARenderStart`. If all rendering commands completed successfully, the value `kQANoErr` is returned. If any other value is returned, you should assume that the rendered image is incorrect.

SPECIAL CONSIDERATIONS

You should call either `QARenderEnd` or `QARenderAbort`, but not both.

QAFlush

You can use the `QAFlush` function to flush a draw context.

```
#define QAFlush(drawContext) (drawContext)->flush (drawContext)
```

`drawContext` A draw context.

DESCRIPTION

The `QAFlush` function causes the drawing engine associated with the draw context specified by the `drawContext` parameter to begin rendering all drawing commands that are queued in a buffer awaiting processing. QuickDraw 3D RAVE allows a drawing engine to buffer as many drawing commands as desired. Accordingly, the successful completion of a drawing command (such as `QADrawPoint`) does not guarantee that the specified object is visible on the screen. You can call `QAFlush` to have a drawing engine start processing queued commands. Note, however, that `QAFlush` is not a blocking call—that is, the successful completion of `QAFlush` does not guarantee that all buffered commands have been processed. Calling `QAFlush` guarantees only that all queued commands will eventually be processed.

Typically, you should occasionally call `QAFlush` to update the screen image during a lengthy set of rendering operations in a single-buffered draw context. `QAFlush` has no visible effect when called on a double-buffered draw context, but it does initiate rendering to the back buffer.

The `TQAFlush` function returns a result code (of type `TQAError`) indicating whether any errors have occurred since the previous call to `QARenderStart`. If all rendering commands completed successfully, the value `kQANoErr` is returned. If any other value is returned, you should assume that the rendered image is incorrect.

SPECIAL CONSIDERATIONS

The `QARenderEnd` function automatically calls `QAFlush`.

SEE ALSO

To ensure that all buffered commands have been processed, you can call `QASync` instead of `QAFlush`.

QASync

You can use the `QASync` function to synchronize a draw context.

```
#define QASync(drawContext) (drawContext)->sync (drawContext)
```

`drawContext` A draw context.

DESCRIPTION

The `QASync` function operates just like the `QAFlush` function, except that it waits until all queued drawing commands have been processed before returning. See the description of `QAFlush` page 1613 for complete details.

QAGetNoticeMethod

You can use the `QAGetNoticeMethod` function to get the notice method of a draw context.

```
#define QAGetNoticeMethod(drawContext, method, completionCallBack, refCon) \
    (drawContext)->getNoticeMethod (drawContext, method, completionCallBack, refCon)
```

`drawContext` A draw context.

`method` A method selector. See “Notice Method Selectors” (page 1571) for a description of the available method selectors.

`completionCallBack`
On exit, a pointer to the current draw context notice method of the specified type.

`refCon` On exit, the reference constant of the specified notice method.

DESCRIPTION

The `QAGetNoticeMethod` function returns, in the `completionCallBack` parameter, a pointer to the current notice method of the draw context specified by the `drawContext` parameter that has the type specified by the `method` parameter. `QAGetNoticeMethod` also returns, in the `refCon` parameter, the reference constant associated with that notice method.

SEE ALSO

Use `QASetNoticeMethod` (next) to set the notice method for a draw context.

QASetNoticeMethod

You can use the `QASetNoticeMethod` function to set the notice method of a draw context.

```
#define QASetNoticeMethod(drawContext, method, completionCallBack, refCon) \
    (drawContext)->setNoticeMethod (drawContext, method, completionCallBack, refCon)
```

`drawContext` A draw context.

`method` A method selector. See "Notice Method Selectors" (page 1571) for a description of the available method selectors.

`completionCallBack`
A pointer to the desired draw context notice method of the specified type. See "Notice Methods" (page 1651) for information about notice methods.

`refCon` A reference constant for the specified notice method. This value is passed unchanged to the notice method when it is called.

DESCRIPTION

The `QASetNoticeMethod` function sets the notice method of type `method` of the draw context specified by the `drawContext` parameter to the function pointed to by the `completionCallBack` parameter. `QASetNoticeMethod` also sets the reference constant of that method to the value specified by the `refCon` parameter.

Registering a Custom Drawing Engine

QuickDraw 3D RAVE provides functions that you can use to register a custom drawing engine and its drawing methods.

QARegisterEngine

You can use the `QARegisterEngine` function to register a custom drawing engine with QuickDraw 3D RAVE.

```
TQAError QARegisterEngine (TQAEngineGetMethod engineGetMethod);
```

`engineGetMethod`
The method retrieval method of your drawing engine. See "Method Reporting Methods" (page 1650) for a complete description of this method.

DESCRIPTION

The `QARegisterEngine` function registers your custom drawing engine with QuickDraw 3D RAVE. You should call this function at startup time (usually from the initialization routine in the shared library containing the code for your drawing engine). QuickDraw 3D RAVE uses the method specified by the `engineGetMethod` parameter to retrieve function pointers for the non-drawing methods defined in your drawing engine.

SPECIAL CONSIDERATIONS

You should call `QARegisterEngine` only to register a custom drawing engine. Applications using QuickDraw 3D RAVE to draw into one or more draw contexts do not need to use this function.

QARegisterDrawMethod

You can use the `QARegisterDrawMethod` function to register a public draw context method with QuickDraw 3D RAVE.

```
TQAError QARegisterDrawMethod (
                    TQADrawContext *drawContext,
                    TQADrawMethodTag methodTag,
                    TQADrawMethod method);
```

`drawContext` A draw context.

`methodTag`	A selector that determines which draw context method is to be registered for the specified draw context. See "Public Draw Context Method Selectors" (page 1569) for complete information about the available method selectors.
`method`	A pointer to the draw context method of the specified type.

DESCRIPTION

The `QARegisterDrawMethod` function changes the method pointer of the draw context specified by the `drawContext` parameter that has the type specified by the `methodTag` parameter to the method specified by the `method` parameter. You should call `QARegisterDrawMethod` instead of directly changing the fields of a draw context structure.

Application-Defined Routines

This section describes the routines you might need to define to add a new drawing engine to the QuickDraw 3D Acceleration Layer.

Note
See "Writing a Drawing Engine," beginning on page 1524 for step-by-step details on writing a drawing engine. ◆

This section also describes the notice method you might need to define to have your application receive notices when certain events occur. See "Notice Methods" (page 1651) for details.

Public Draw Context Methods

To write a drawing engine, you need to implement a number of drawing methods, pointers to which are contained in a draw context structure (of type `TQADrawContext`). These functions are called whenever an application uses one of the drawing macros described earlier (in "Manipulating Draw Contexts," beginning on page 1598). For example, when an application uses the `QADrawPoint` macro to draw a point in a draw context linked to your drawing engine, your engine's `TQADrawPoint` method is called.

A draw context structure is passed as the first parameter to all these draw context methods. This allows you to retrieve your draw context's private data, which is pointed to by the first field of that structure.

IMPORTANT

Most of the draw context methods declare the draw context structure passed to them as `const`, in which case you should not alter any fields of that structure. Only three methods are allowed to change fields of the draw context structure: `TQASetFloat`, `TQASetInt`, and `TQASetPtr`. Failure to heed the `const` declaration may cause any code calling your engine (including QuickDraw 3D) to fail. ▲

Pointers to your drawing engine's public draw context methods are assigned to the fields of a draw context structure by your `TQADrawPrivateNew` method. See page 1640 for details.

TQAGetFloat

A drawing engine must define a method to get a floating-point value of a draw context state variable.

```
typedef float (*TQAGetFloat) (
                    const TQADrawContext *drawContext,
                    TQATagFloat tag);
```

`drawContext` A draw context.

`tag` A state variable tag.

DESCRIPTION

Your `TQAGetFloat` function should return, as its function result, the floating-point value of the draw context state variable specified by the `drawContext` and `tag` parameters. If you do not recognize or support the specified tag, your `TQAGetFloat` function should return the value 0.

TQASetFloat

A drawing engine must define a method to set a floating-point value for a draw context state variable.

```
typedef void (*TQASetFloat) (
                    TQADrawContext *drawContext,
                    TQATagFloat tag,
                    float newValue);
```

`drawContext`	A draw context.
`tag`	A state variable tag.
`newValue`	The new value of the specified state variable.

DESCRIPTION

Your `TQASetFloat` function should set the value of the draw context state variable specified by the `drawContext` and `tag` parameters to the floating-point value specified by the `newValue` parameter.

Your drawing engine must accept all possible values for the `tag` parameter. If you encounter a value in the `tag` parameter that you cannot recognize, you should do nothing. Similarly, you should do nothing if the `tag` parameter specifies a state variable for optional features your drawing engine does not support.

SPECIAL CONSIDERATIONS

If your `TQASetFloat` function needs to change one or more of the function pointers in the specified draw context, it must call the `QARegisterDrawMethod` function to do so. It should not directly change the fields of a draw context.

TQAGetInt

A drawing engine must define a method to get a long integer value of a draw context state variable.

```
typedef unsigned long (*TQAGetInt) (
                    const TQADrawContext *drawContext,
                    TQATagInt tag);
```

`drawContext` A draw context.

`tag` A state variable tag.

DESCRIPTION

Your `TQAGetInt` function should return, as its function result, the long integer value of the draw context state variable specified by the `drawContext` and `tag` parameters. If you do not recognize or support the specified tag, your `TQAGetInt` function should return the value 0.

TQASetInt

A drawing engine must define a method to set a long integer value for a draw context state variable.

```
typedef void (*TQASetInt) (
                    TQADrawContext *drawContext,
                    TQATagInt tag,
                    unsigned long newValue);
```

`drawContext` A draw context.

`tag` A state variable tag.

`newValue` The new value of the specified state variable.

DESCRIPTION

Your `TQASetInt` function should set the value of the draw context state variable specified by the `drawContext` and `tag` parameters to the long integer value specified by the `newValue` parameter.

Your drawing engine must accept all possible values for the `tag` parameter. If you encounter a value in the `tag` parameter that you cannot recognize, you should do nothing. Similarly, you should do nothing if the `tag` parameter specifies a state variable for optional features your drawing engine does not support.

SPECIAL CONSIDERATIONS

If your `TQASetInt` function needs to change one or more of the function pointers in the specified draw context, it must call the `QARegisterDrawMethod` function to do so. It should not directly change the fields of a draw context.

TQAGetPtr

A drawing engine must define a method to get a pointer value of a draw context state variable.

```
typedef void *(*TQAGetPtr) (
                    const TQADrawContext *drawContext,
                    TQATagPtr tag);
```

`drawContext` A draw context.

`tag` A state variable tag.

DESCRIPTION

Your `TQAGetPtr` function should return, as its function result, the pointer value of the draw context state variable specified by the `drawContext` and `tag` parameters. If you do not recognize or support the specified tag, your `TQAGetPtr` function should return the value 0.

TQASetPtr

A drawing engine must define a method to set a pointer value for a draw context state variable.

```
typedef void (*TQASetPtr) (
                    TQADrawContext *drawContext,
                    TQATagPtr tag,
                    const void *newValue);
```

`drawContext`	A draw context.
`tag`	A state variable tag.
`newValue`	The new value of the specified state variable.

DESCRIPTION

Your `TQASetPtr` function should set the value of the draw context state variable specified by the `drawContext` and `tag` parameters to the pointer value specified by the `newValue` parameter.

Your drawing engine must accept all possible values for the `tag` parameter. If you encounter a value in the `tag` parameter that you cannot recognize, you should do nothing. Similarly, you should do nothing if the `tag` parameter specifies a state variable for optional features your drawing engine does not support.

SPECIAL CONSIDERATIONS

If your `TQASetPtr` function needs to change one or more of the function pointers in the specified draw context, it must call the `QARegisterDrawMethod` function to do so. It should not directly change the fields of a draw context.

TQADrawPoint

A drawing engine must define a method to draw a point.

```
typedef void (*TQADrawPoint) (
                    const TQADrawContext *drawContext,
                    const TQAVGouraud *v);
```

`drawContext` A draw context.

`v` A Gouraud vertex.

DESCRIPTION

Your `TQADrawPoint` function should draw the single point specified by the `v` parameter to the draw context specified by the `drawContext` parameter. The size of the point is determined by the `kQATag_Width` state variable of the draw context.

TQADrawLine

A drawing engine must define a method to draw a line between two points.

```
typedef void (*TQADrawLine) (
                    const TQADrawContext *drawContext,
                    const TQAVGouraud *v0,
                    const TQAVGouraud *v1);
```

`drawContext` A draw context.

`v0` A Gouraud vertex.

`v1` A Gouraud vertex.

DESCRIPTION

Your `TQADrawLine` function should draw the line specified by the `v0` and `v1` parameters to the draw context specified by the `drawContext` parameter. The size of the line is determined by the `kQATag_Width` state variable of the draw context.

If the specified vertices have different colors, the line color is interpolated smoothly between the two vertex colors.

TQADrawTriGouraud

A drawing engine must define a method to draw Gouraud-shaded triangles.

```
typedef void (*TQADrawTriGouraud) (
                    const TQADrawContext *drawContext,
                    const TQAVGouraud *v0,
                    const TQAVGouraud *v1,
                    const TQAVGouraud *v2,
                    unsigned long flags);
```

`drawContext`	A draw context.
`v0`	A Gouraud vertex.
`v1`	A Gouraud vertex.
`v2`	A Gouraud vertex.
`flags`	A set of triangle flags. See “Triangle Flags Masks,” beginning on page 1566 for a complete description of the available flags.

DESCRIPTION

Your `TQADrawTriGouraud` function should draw the Gouraud-shaded triangle determined by the three points specified by the `v0`, `v1`, and `v2` parameters into the draw context specified by the `drawContext` parameter. Features of the triangle are determined by the `flags` parameter. Currently, this parameter is used to specify an orientation for the triangle.

TQADrawTriTexture

A drawing engine may define a method to draw texture-mapped triangles. This method is optional and must be supported only by drawing engines that support texture mapping.

```
typedef void (*TQADrawTriTexture) (
                    const TQADrawContext *drawContext,
                    const TQAVTexture *v0,
                    const TQAVTexture *v1,
                    const TQAVTexture *v2,
                    unsigned long flags);
```

`drawContext`	A draw context.
`v0`	A texture vertex.
`v1`	A texture vertex.
`v2`	A texture vertex.
`flags`	A set of triangle flags. See "Triangle Flags Masks," beginning on page 1566 for a complete description of the available flags.

DESCRIPTION

Your `TQADrawTriTexture` function should draw the texture-mapped triangle determined by the three points specified by the `v0`, `v1`, and `v2` parameters into the draw context specified by the `drawContext` parameter. The texture used for the mapping is determined by the value of the `kQATag_Texture` state variable. Features of the triangle are determined by the `flags` parameter. Currently, this parameter is used to specify an orientation for the triangle.

TQASubmitVerticesGouraud

A drawing engine may define a method to submit Gouraud vertices.

```
typedef void (*TQASubmitVerticesGouraud) (
                    const TQADrawContext *drawContext,
                    unsigned long nVertices,
                    const TQAVGouraud *vertices);
```

`drawContext` A draw context.

`nVertices` The number of Gouraud vertices pointed to by the `vertices` parameter.

`vertices` A pointer to an array of Gouraud vertices.

DESCRIPTION

Your `TQASubmitVerticesGouraud` function should prepare to render a Gouraud-shaded triangular mesh in the draw context specified by the `drawContext` parameter using the vertices pointed to by the `vertices` parameter. The actual triangulation and drawing of the mesh does not occur until an application calls the `QADrawTriMeshGouraud` function.

The calling application is responsible for managing the memory occupied by the Gouraud vertices. Your `TQASubmitVerticesGouraud` function should not copy the vertex data pointed to by the `vertices` parameter.

SPECIAL CONSIDERATIONS

The `TQASubmitVerticesGouraud` method is optional. If your drawing engine does not support triangle meshes, QuickDraw 3D RAVE decomposes a triangle mesh into individual triangles when the user calls the `QASubmitVerticesGouraud` function to submit a triangle mesh.

There is no QuickDraw 3D RAVE function that an application can use to unsubmit a triangle mesh. Your drawing engine must manage memory in some appropriate manner.

TQASubmitVerticesTexture

A drawing engine may define a method to submit texture vertices. This method is optional and must be supported only by drawing engines that support texture mapping.

```
typedef void (*TQASubmitVerticesTexture) (
                    const TQADrawContext *drawContext,
                    unsigned long nVertices,
                    const TQAVTexture *vertices);
```

`drawContext` A draw context.

`nVertices` The number of texture vertices pointed to by the `vertices` parameter.

`vertices` A pointer to an array of texture vertices.

DESCRIPTION

Your `TQASubmitVerticesTexture` function should prepare to render a texture-mapped triangular mesh in the draw context specified by the `drawContext` parameter using the vertices pointed to by the `vertices` parameter. The actual triangulation and drawing of the mesh does not occur until an application calls the `QADrawTriMeshTexture` function.

The calling application is responsible for managing the memory occupied by the texture vertices. Your `TQASubmitVerticesTexture` function should not copy the vertex data pointed to by the `vertices` parameter.

SPECIAL CONSIDERATIONS

The `TQASubmitVerticesTexture` method is optional. If your drawing engine does not support triangle meshes, QuickDraw 3D RAVE decomposes a triangle mesh into individual triangles when the user calls the `QASubmitVerticesTexture` function to submit a triangle mesh.

There is no QuickDraw 3D RAVE function that an application can use to unsubmit a triangle mesh. Your drawing engine must manage memory in some appropriate manner.

TQADrawTriMeshGouraud

A drawing engine may define a method to draw a triangle mesh with Gouraud shading.

```
typedef void (*TQADrawTriMeshGouraud) (
                    const TQADrawContext *drawContext,
                    unsigned long nTriangles,
                    const TQAIndexedTriangle *triangles);
```

`drawContext` A draw context.

`nTriangle` The number of indexed triangles pointed to by the `triangles` parameter.

`triangles` A pointer to an array of indexed triangles. See “Indexed Triangle Structure” (page 1584) for a description of indexed triangles.

DESCRIPTION

Your `TQADrawTriMeshGouraud` function should draw, with Gouraud shading, the triangle mesh specified by the `triangles` parameter into the draw context specified by the `drawContext` parameter. Each triangle in the mesh is defined by a `TQAIndexedTriangle` data structure, which contains three indices into the array of Gouraud vertices previously submitted to the draw context by a call to the `QASubmitVerticesGouraud` function.

SPECIAL CONSIDERATIONS

The `TQADrawTriMeshGouraud` method is optional. If your drawing engine does not support triangle meshes, QuickDraw 3D RAVE decomposes a triangle mesh into individual triangles when the user calls the `QASubmitVerticesGouraud` function to submit a triangle mesh.

TQADrawTriMeshTexture

A drawing engine may define a method to draw a texture-mapped triangle mesh. This method is optional and must be supported only by drawing engines that support texture mapping.

```
typedef void (*TQADrawTriMeshTexture) (
                    const TQADrawContext *drawContext,
                    unsigned long nTriangles,
                    const TQAIndexedTriangle *triangles);
```

`drawContext` A draw context.

`nTriangle` The number of indexed triangles pointed to by the `triangles` parameter.

`triangles` A pointer to an array of indexed triangles. See "Indexed Triangle Structure" (page 1584) for a description of indexed triangles.

DESCRIPTION

Your `TQADrawTriMeshTexture` function should draw the texture-mapped triangle mesh specified by the `triangles` parameter into the draw context specified by the `drawContext` parameter. Each triangle in the mesh is defined by a `TQAIndexedTriangle` data structure, which contains three indices into the array of texture vertices previously submitted to the draw context by a call to the `QASubmitVerticesTexture` function.

SPECIAL CONSIDERATIONS

The `TQADrawTriMeshTexture` method is optional. If your drawing engine does not support triangle meshes, QuickDraw 3D RAVE decomposes a triangle mesh into individual triangles when the user calls the `QASubmitVerticesTexture` function to submit a triangle mesh.

TQADrawVGouraud

A drawing engine may define a method to draw Gouraud-shaded objects defined by vertices. This method is optional and must be supported only by

drawing engines that do not want calls to `QADrawVGouraud` to be replaced by calls to the `QADrawPoint`, `QADrawLine`, or `QADrawTriGouraud` methods.

```
typedef void (*TQADrawVGouraud) (
                    const TQADrawContext *drawContext,
                    unsigned long nVertices,
                    TQAVertexMode vertexMode,
                    const TQAVGouraud vertices[],
                    const unsigned long flags[]);
```

`drawContext`	A draw context.
`nVertices`	The number of vertices contained in the `vertices` array.
`vertexMode`	A vertex mode. See "Vertex Modes" (page 1558) for a description of the available vertex modes.
`vertices`	An array of Gouraud vertices.
`flags`	An array of triangle flags, or the value `NULL`. See "Triangle Flags Masks" (page 1566) for a description of the available triangle flags. This parameter is valid only if the `vertexMode` parameter contains the value `kQAVertexMode_Tri`, `kQAVertexMode_Strip`, or `kQAVertexMode_Fan`.

DESCRIPTION

Your `TQADrawVGouraud` function should draw the vertices in the array specified by the `vertices` parameter into the draw context specified by the `drawContext` parameter, according to the vertex modes flag specified by the `vertexMode` parameter. For instance, if the value of the `vertexMode` parameter is `kQAVertexMode_Polyline`, then the vertices in that array are interpreted as defining a polyline (a set of connected line segments). Gouraud shading should be applied to whatever objects are drawn.

TQADrawVTexture

A drawing engine may define a method to draw texture-mapped objects defined by vertices. This method is optional and must be supported only by drawing engines that support texture mapping and do not want calls to

QADrawVTexture to be replaced by calls to the QADrawPoint, QADrawLine, or QADrawTriTexture methods.

```
typedef void (*TQADrawVTexture) (
                    const TQADrawContext *drawContext,
                    unsigned long nVertices,
                    TQAVertexMode vertexMode,
                    const TQAVTexture vertices[],
                    const unsigned long flags[]);
```

drawContext	A draw context.
nVertices	The number of vertices contained in the vertices array.
vertexMode	A vertex mode. See "Vertex Modes" (page 1558) for a description of the available vertex modes.
vertices	An array of texture vertices.
flags	An array of triangle flags, or the value NULL. See "Triangle Flags Masks" (page 1566) for a description of the available triangle flags. This parameter is valid only if the vertexMode parameter contains the value kQAVertexMode_Tri, kQAVertexMode_Strip, or kQAVertexMode_Fan.

DESCRIPTION

Your TQADrawVTexture function should draw the vertices in the array specified by the vertices parameter into the draw context specified by the drawContext parameter, according to the vertex modes flag specified by the vertexMode parameter. For instance, if the value of the vertexMode parameter is kQAVertexMode_Polyline, then the vertices in that array are interpreted as defining a polyline (a set of connected line segments). Texture mapping (using the texture determined by the value of the kQATag_Texture state variable) should be applied to whatever objects are drawn.

IMPORTANT

The vertex modes kQAVertexMode_Point and kQAVertexMode_Line are supported only by drawing engines that support the kQAOptional_OpenGL feature. All other drawing engines should ignore requests to texture map points or lines. ▲

TQADrawBitmap

A drawing engine must define a method to draw bitmaps into a draw context.

```
typedef void (*TQADrawBitmap) (
                    const TQADrawContext *drawContext,
                    const TQAVGouraud *v,
                    TQABitmap *bitmap);
```

`drawContext` A draw context.

`v` A Gouraud vertex.

`bitmap` A pointer to a bitmap (returned by a previous call to `QABitmapNew`).

DESCRIPTION

Your `TQADrawBitmap` function should draw the bitmap specified by the `bitmap` parameter into the draw context specified by the `drawContext` parameter, with the upper-left corner of the bitmap located at the point specified by the `v` parameter. The `v` parameter can contain negative values in its `x` or `y` fields, so you can position upper-left corner of the bitmap outside the draw context rectangle. This allows you to move the bitmap smoothly off any edge of the draw context.

TQARenderStart

A drawing engine must define a method to initialize a draw context before the engine performs any rendering into that context.

```
typedef void (*TQARenderStart) (
                    const TQADrawContext *drawContext,
                    const TQARect *dirtyRect,
                    const TQADrawContext *initialContext);
```

`drawContext` A draw context.

`dirtyRect` The minimum area of the specified draw context to clear, or the value `NULL`.

`initialContext`
A previously cached draw context, or the value `NULL`.

DESCRIPTION

Your `TQARenderStart` function should perform any operations necessary to initialize the draw context specified by the `drawContext` parameter. This includes clearing the z buffer and the color buffers of the draw context. If the value of the `initialContext` parameter is `NULL`, then your `TQARenderStart` function should clear the z buffer to 1.0 and set the color buffers to the values of the `kQATag_ColorBG_a`, `kQATag_ColorBG_r`, `kQATag_ColorBG_g`, and `kQATag_ColorBG_b` draw context state variables. If, however, the value of the `initialContext` parameter is not `NULL`, then your `TQARenderStart` function should use the previously cached draw context specified by that parameter to initialize the draw context specified by the `drawContext` parameter.

The `dirtyRect` parameter indicates the minimum area of the specified draw context that should be cleared on initialization. If the value of the `dirtyRect` parameter is `NULL`, the entire draw context is cleared. If the value of the `dirtyRect` parameter is not `NULL`, it indicates the rectangle in the draw context to clear. Some drawing engines may exhibit improved performance when an area that is smaller than the entire draw context rectangle is passed. However, the interpretation of the `dirtyRect` parameter is dependent on the drawing engine, which may choose to initialize the entire draw context. As a result, code calling your `QARenderStart` function should not use this parameter as a means to avoid clearing all of a draw context or to perform incremental rendering. Instead, that code should use the `initialContext` parameter to achieve such effects.

SPECIAL CONSIDERATIONS

Applications should call `QARenderStart` before performing any rendering operations in the specified draw context, and they should call either `QARenderEnd` to signal the end of rendering operations or `QARenderAbort` to cancel rendering operations. However, when a drawing engine is performing OpenGL rendering, the `QARenderStart` function operates just like the OpenGL function `glClear`. In OpenGL mode, it is not necessary that a call to `QARenderStart` always be balanced by a matching call to `QARenderEnd`, and drawing commands may occur at any time.

TQARenderEnd

A drawing engine must define a method to signal the end of any rendering into a draw context.

```
typedef TQAError (*TQARenderEnd) (
                    const TQADrawContext *drawContext,
                    const TQARect *modifiedRect);
```

`drawContext` A draw context.

`modifiedRect` The minimum area of the back buffer of the specified draw context to display, or the value `NULL`.

DESCRIPTION

Your `TQARenderEnd` function should perform any operations necessary to display an image rendered into the draw context specified by the `drawContext` parameter. If the draw context is double buffered, your function should display the back buffer. If the draw context is single buffered, your function should call `QAFlush`. In either case, your drawing engine should unlock any frame buffers or other memory that is locked, remove any cursor shields, and so forth.

The `modifiedRect` parameter indicates the minimum area of the back buffer of the specified draw context that should be displayed. If the value of the `modifiedRect` parameter is `NULL`, the entire back buffer is displayed. If the value of the `modifiedRect` parameter is not `NULL`, it indicates the rectangle in the back buffer to display. Some drawing engines may exhibit improved performance when an area that is smaller than the entire draw context rectangle is passed (to avoid unnecessary pixel copying). However, the interpretation of the `modifiedRect` parameter is dependent on the drawing engine, which may choose to draw the entire back buffer.

Your `TQARenderEnd` function should return a result code (of type `TQAError`) indicating whether any errors have occurred since the previous call to your `TQARenderStart` function. If all rendering commands completed successfully, you should return the value `kQANoErr`. If you return any other value, the code that called `QARenderEnd` should assume that the rendered image is incorrect.

SPECIAL CONSIDERATIONS

Applications should call `QARenderStart` before performing any rendering operations in the specified draw context, and they should call either `QARenderEnd` to signal the end of rendering operations or `QARenderAbort` to cancel rendering operations. Once an application has called `QARenderEnd`, it should not submit any drawing requests until it has called `QARenderStart` again.

TQARenderAbort

A drawing engine must define a method to cancel any asynchronous drawing requests for a draw context.

```
typedef TQAError (*TQARenderAbort) (
                    const TQADrawContext *drawContext);
```

`drawContext` A draw context.

DESCRIPTION

Your `TQARenderAbort` function should immediately stop processing any asynchronous drawing command it is currently processing and it should discard any queued commands associated with the draw context specified by the `drawContext` parameter.

Your `TQARenderAbort` function should return a result code (of type `TQAError`) indicating whether any errors have occurred since the previous call to your `TQARenderStart` function. If all rendering commands completed successfully, you should return the value `kQANoErr`. If you return any other value, the code that called `QARenderEnd` should assume that the rendered image is incorrect.

TQAFlush

A drawing engine must define a method to flush a draw context.

```
typedef TQAError (*TQAFlush) (const TQADrawContext *drawContext);
```

`drawContext` A draw context.

DESCRIPTION

Your `TQAFlush` function should cause your drawing engine to begin rendering all drawing commands that are queued in a buffer awaiting processing for the draw context specified by the `drawContext` parameter. QuickDraw 3D RAVE allows a drawing engine to buffer as many drawing commands as desired. Accordingly, the successful completion of a drawing command (such as `QADrawPoint`) does not guarantee that the specified object is visible on the screen. An application can call `QAFlush` to have your drawing engine start processing queued commands. Note, however, that `QAFlush` is not a blocking call—that is, the successful completion of `QAFlush` does not guarantee that all buffered commands have been processed. Calling `QAFlush` guarantees only that all queued commands will eventually be processed.

Typically, applications should occasionally call `QAFlush` to update the screen image during a lengthy set of rendering operations in a single-buffered draw context. `QAFlush` has no visible effect when called on a double-buffered draw context, but it does initiate rendering to the back buffer.

Your `TQAFlush` function should return a result code (of type `TQAError`) indicating whether any errors have occurred since the previous call to your `TQARenderStart` function. If all rendering commands completed successfully, you should return the value `kQANoErr`. If you return any other value, the code that called `QAFlush` should assume that the rendered image is incorrect.

TQASync

A drawing engine must define a method to synchronize a draw context.

```
typedef TQAError (*TQASync) (const TQADrawContext *drawContext);
```

`drawContext` A draw context.

DESCRIPTION

Your `TQASync` function should operate just like your `TQAFlush` function, except that it should wait until all queued drawing commands have been processed before returning. See the description of `TQAFlush` page 1636 for complete details.

TQAGetNoticeMethod

A drawing engine must define a method to return the notice method of a draw context.

```
typedef TQAError (*TQAGetNoticeMethod) (
                    const TQADrawContext *drawContext,
                    TQAMethodSelector method,
                    TQANoticeMethod *completionCallBack,
                    void **refCon);
```

`drawContext` A draw context.

`method` A method selector. See "Notice Method Selectors" (page 1571) for a description of the available method selectors.

`completionCallBack`
On exit, a pointer to the current draw context notice method of the specified type.

`refCon` On exit, the reference constant of the specified notice method.

DESCRIPTION

Your `TQAGetNoticeMethod` function should return, in the `completionCallBack` parameter, a pointer to the current notice method of the draw context specified by the `drawContext` parameter that has the type specified by the `method` parameter. `TQAGetNoticeMethod` should also return, in the `refCon` parameter, the reference constant associated with that notice method.

TQASetNoticeMethod

A drawing engine must define a method to set the notice method of a draw context.

```
typedef TQAError (*TQASetNoticeMethod) (
                    const TQADrawContext *drawContext,
                    TQAMethodSelector method,
                    TQANoticeMethod completionCallBack,
                    void *refCon);
```

`drawContext` A draw context.

`method` A method selector. See “Notice Method Selectors” (page 1571) for a description of the available method selectors.

`completionCallBack`
A pointer to the desired draw context notice method of the specified type. See “Notice Methods” (page 1651) for information about notice methods.

`refCon` A reference constant for the specified notice method. This value is passed unchanged to the notice method when it is called.

DESCRIPTION

Your `TQASetNoticeMethod` function should set the notice method of type `method` of the draw context specified by the `drawContext` parameter to the function pointed to by the `completionCallBack` parameter. `TQASetNoticeMethod` should also set the reference constant of that method to the value specified by the `refCon` parameter.

Private Draw Context Methods

To write a drawing engine, you need to implement several private methods for managing draw contexts.

Pointers to your drawing engine’s private draw context methods are returned to QuickDraw 3D RAVE by your `TQAEngineGetMethod` method. See page 1650 for details.

TQADrawPrivateNew

A drawing engine must define a method to create its own private data and initialize a new draw context.

```
typedef TQAError (*TQADrawPrivateNew) (
                    TQADrawContext *newDrawContext,
                    const TQADevice *device,
                    const TQARect *rect,
                    const TQAClip *clip,
                    unsigned long flags);
```

`newDrawContext`
The draw context to initialize. On entry, all the fields of this structure have the value `NULL`.

`device` A device.

`rect` The rectangular region (specified in device coordinates) of the specified device that can be drawn into by the drawing engine associated with the new draw context.

`clip` The two-dimensional clipping region for the new draw context, or `NULL` if no clipping is desired. This parameter must be set to `NULL` for devices of type `kQADeviceMemory`.

`flags` A set of bit flags specifying features of the new draw context. See "Draw Context Flags Masks" (page 1567) for complete information.

DESCRIPTION

Your `TQADrawPrivateNew` function is called whenever an application calls `QADrawContextNew` to create a new draw context associated with your drawing engine. Your function should perform any initialization required for the new draw context. In particular, it should return a pointer to the draw context's private data in the `drawPrivate` field of the draw context structure pointed to by the `newDrawContext` parameter. In addition, your `TQADrawPrivateNew` function should set any other fields of that draw context structure to point to public draw context methods defined by the drawing engine.

Because it is the responsibility of your `TQADrawPrivateNew` function to initialize the fields of a draw context structure, you can load different methods

depending on the features of the device or draw context specified by the `device` and `flags` parameters. For instance, you might load one line drawing function for a device that displays 16 bits per pixel and a different line drawing function for a device that displays 32 bits per pixel. This technique allows you to avoid testing the display depth each time you draw a line.

SEE ALSO

See Listing 23-8 (page 1527) for a sample `TQADrawPrivateNew` function.

TQADrawPrivateDelete

A drawing engine must define a method to delete its private data.

```
typedef void (*TQADrawPrivateDelete) (TQADrawPrivate *drawPrivate);
```

`drawPrivate` The draw context's private data.

DESCRIPTION

Your `TQADrawPrivateDelete` function is called whenever an application calls `QADrawContextDelete`. Your function should release any memory or other resources that were allocated by your `TQADrawPrivateNew` function.

TQAEngineCheckDevice

A drawing engine must define a method to indicate whether the engine can draw to a particular device.

```
typedef TQAError (*TQAEngineCheckDevice) (const TQADevice *device);
```

`device` A device.

DESCRIPTION

Your `TQAEngineCheckDevice` function should return, as its function result, a code that indicates whether your drawing engine can draw into the device specified by the `device` parameter (`kQANoErr`) or not (`kQAError`).

TQAEngineGestalt

A drawing engine must define a method to return information about its capabilities.

```
typedef TQAError (*TQAEngineGestalt) (
                    TQAGestaltSelector selector,
                    void *response);
```

`selector` A selector that determines what kind of information is to be returned about your drawing engine. See "Gestalt Selectors" (page 1559) for complete information about the available selectors and the information you should return.

`response` A pointer to a buffer into which the returned information is to be copied. The calling application is responsible for allocating this buffer. The size and meaning of the data to be copied depends on the selector passed in the `selector` parameter.

DESCRIPTION

Your `TQAEngineGestalt` function is called whenever an application calls `QAEngineGestalt`. Your function should return, in the buffer pointed to by the `response` parameter, information about features of the type specified by the `selector` parameter.

Color Lookup Table Methods

To write a drawing engine, you might need to implement several private methods for creating and disposing of color lookup tables. Pointers to your drawing engine's color lookup table methods are returned to QuickDraw 3D RAVE by your `TQAEngineGetMethod` method. See page 1650 for details.

TQAColorTableNew

A drawing engine may define a method to create a new color lookup table. This method is optional and must be supported only by drawing engines that support color lookup tables.

```
typedef TQAError (*TQAColorTableNew) (
                    TQAColorTableType pixelType,
                    void *pixelData,
                    long transparentIndex,
                    TQAColorTable **newTable);
```

`pixelType` The type of the new color lookup table. See "Color Lookup Table Types" (page 1538) for information on the available color lookup table types.

`pixelData` A pointer to the color lookup table entries.

`transparentIndexFlag`
A long integer, interpreted as a Boolean value, that indicates whether the color lookup table entry at index 0 is completely transparent (`TRUE`) or not (`FALSE`).

`newTable` On entry, the address of a pointer variable. On exit, set that variable to point to a new color lookup table. If a new color lookup table cannot be created, set `*newTable` to the value `NULL`.

DESCRIPTION

Your `TQAColorTableNew` function is called whenever an application calls `QAColorTableNew`. Your function should return, in the buffer pointed to by the `newTable` parameter, a pointer to a new color lookup table of the type specified by the `pixelType` parameter. The color table data is passed to your function in the `pixelData` parameter. Your method should copy that data so that the caller can dispose of the memory it occupies.

IMPORTANT

Currently, QuickDraw 3D RAVE supports only 32-bit RGB color lookup table entries. Your drawing engine might reduce the size of individual color lookup table entries to fit into its on-board memory. ▲

SPECIAL CONSIDERATIONS

Not all drawing engines need to support color lookup tables, but QuickDraw 3D RAVE does not provide color lookup table emulation for engines that do not support them.

TQAColorTableDelete

A drawing engine may define a method to dispose of color lookup table. This method is optional and must be supported only by drawing engines that support color lookup tables.

```
typedef void (*TQAColorTableDelete) (TQAColorTable *colorTable);
```

`colorTable` A color lookup table.

DESCRIPTION

Your `TQAColorTableDelete` function is called whenever an application calls `QAColorTableDelete`. Your function should delete the color lookup table specified by the `colorTable` parameter. Any memory and other resources associated with that color lookup table should be released.

Texture and Bitmap Methods

To write a drawing engine, you need to implement several private methods for managing bitmaps. If your engine supports texture mapping, you also need to implement several private methods for managing textures.

Pointers to your drawing engine's texture and bitmap methods are returned to QuickDraw 3D RAVE by your `TQAEngineGetMethod` method. See page 1650 for details.

TQATextureNew

A drawing engine may define a method to create a new texture map. This method is optional and must be supported only by drawing engines that support texture mapping.

```
typedef TQAError (*TQATextureNew) (
                    unsigned long flags,
                    TQAImagePixelType pixelType,
                    const TQAImage images[],
                    TQATexture **newTexture);
```

`flags` A set of bit flags specifying features of the new texture map. See "Texture Flags Masks" (page 1566) for complete information.

`pixelType` The type of pixels in the new texture map. See "Pixel Types" (page 1536) for a description of the values you can pass in this parameter.

`images` An array of pixel images to use for the new texture map. The values in the `width` and `height` fields of these structures must be an even power of 2.

`newTexture` On entry, the address of a pointer variable. On exit, that variable points to a new texture map. If a new texture map cannot be created, `*newTexture` is set to the value `NULL`.

DESCRIPTION

Your `TQATextureNew` function is called whenever an application calls `QATextureNew`. Your function should perform any tasks required to use the texture in texture-mapping operations. This might involve loading the texture into memory on the device associated with your drawing engine. If so, your `TQATextureNew` function should not return until the texture has been completely loaded.

The `flags` parameter specifies a set of texture map features. If the `kQATexture_Lock` bit in that parameter is set but your drawing engine cannot guarantee that the texture will remain locked in memory, your `TQATextureNew` function should return an error.

If the `kQATexture_Mipmap` bit of the `flags` parameter is clear, the `images` parameter points to a single pixel image that defines the texture map. If the

kQATexture_Mipmap bit is set, the images parameter points to an array of pixel images of varying pixel depths. The first element in the array must be the mipmap page having the highest resolution, with a width and height that are even powers of 2. Each subsequent pixel image in the array should have a width and height that are half those of the previous image (with a minimum width and height of 1).

TQATextureDetach

A drawing engine may define a method to detach a texture map. This method is optional and must be supported only by drawing engines that support texture mapping.

```
typedef TQAError (*TQATextureDetach) (TQATexture *texture);
```

texture A texture map.

DESCRIPTION

Your TQATextureDetach function is called whenever an application calls QATextureDetach. Your function should, if necessary, load the texture specified by the texture parameter into memory on the device associated with your drawing engine (so that the caller can release the memory occupied by the texture). Your TQATextureDetach function should not return until the texture has been completely loaded.

TQATextureBindColorTable

A drawing engine may define a method to bind a color lookup table to a texture map.

```
typedef TQAError (*TQATextureBindColorTable) (
                    TQATexture *texture,
                    TQAColorTable *colorTable);
```

texture A texture map.

`colorTable` A color lookup table (as returned by a previous call to `QAColorTableNew`).

DESCRIPTION

Your `TQATextureBindColorTable` function is called whenever an application calls `QATextureBindColorTable`. Your function should bind the color lookup table specified by the `colorTable` parameter to the texture map specified by the `texture` parameter. Note that the type of the specified color lookup table must match that of the pixel type of the texture map to which it is bound. For example, a color lookup table of type `kQAColorTable_CL8_RGB32` can be bound only to a texture map whose pixel type is `kQAPixel_CL8`.

TQATextureDelete

A drawing engine may define a method to delete a texture map. This method is optional and must be supported only by drawing engines that support texture mapping.

```
typedef void (*TQATextureDelete) (TQATexture *texture);
```

`texture` A texture map.

DESCRIPTION

Your `TQATextureDelete` function is called whenever an application calls `QATextureDelete`. Your function should delete the texture map specified by the `texture` parameter.

TQABitmapNew

A drawing engine must define a method to create a new bitmap.

```
typedef TQAError (*TQABitmapNew) (
                    unsigned long flags,
                    TQAImagePixelType pixelType,
                    const TQAImage *image,
                    TQABitmap **newBitmap);
```

`flags` A set of bit flags specifying features of the new bitmap. See "Bitmap Flags Masks" (page 1567) for complete information

`pixelType` The type of pixels in the new bitmap. See "Pixel Types" (page 1536) for a description of the values you can pass in this parameter.

`image` A pixel image to use for the new bitmap. The `width` and `height` fields of this image can have any values greater than 0.

`newBitmap` On entry, the address of a pointer variable. On exit, that variable points to a new bitmap. If a new bitmap cannot be created, `*newBitmap` is set to the value `NULL`.

DESCRIPTION

Your `TQABitmapNew` function is called whenever an application calls `QABitmapNew`. Your function should perform any tasks required to draw the bitmap in the draw context associated with your drawing engine. This might involve loading the bitmap into memory on the device associated with your drawing engine. If so, your `TQABitmapNew` function should not return until the bitmap has been completely loaded.

The `flags` parameter specifies a set of bitmap features. If the `kQABitmap_Lock` bit in that parameter is set but your drawing engine cannot guarantee that the bitmap will remain locked in memory, your `TQABitmapNew` function should return an error.

TQABitmapDetach

A drawing engine must define a method to detach a bitmap from a drawing engine.

```
typedef TQAError (*TQABitmapDetach) (TQABitmap *bitmap);
```

`bitmap` A bitmap.

DESCRIPTION

Your `TQABitmapDetach` function is called whenever an application calls `QABitmapDetach`. Your function should, if necessary, load the bitmap specified by the `bitmap` parameter into memory on the device associated with your drawing engine (so that the caller can release the memory occupied by the bitmap). Your `TQABitmapDetach` function should not return until the bitmap has been completely loaded.

TQABitmapBindColorTable

A drawing engine may define a method to bind a color lookup table to a bitmap.

```
typedef TQAError (*TQABitmapBindColorTable) (
                    TQABitmap *bitmap,
                    TQAColorTable *colorTable);
```

`bitmap` A bitmap.

`colorTable` A color lookup table (as returned by a previous call to `QAColorTableNew`).

DESCRIPTION

Your `TQABitmapBindColorTable` function is called whenever an application calls `QABitmapBindColorTable`. Your function should bind the color lookup table specified by the `colorTable` parameter to the bitmap specified by the `bitmap` parameter. Note that the type of the specified color lookup table must match

that of the pixel type of the bitmap to which it is bound. For example, a color lookup table of type `kQAColorTable_CL8_RGB32` can be bound only to a bitmap whose pixel type is `kQAPixel_CL8`.

TQABitmapDelete

A drawing engine must define a method to delete a bitmap.

```
typedef void (*TQABitmapDelete) (TQABitmap *bitmap);
```

`bitmap` A bitmap.

DESCRIPTION

Your `TQABitmapDelete` function is called whenever an application calls `QABitmapDelete`. Your function should delete the bitmap specified by the `bitmap` parameter.

Method Reporting Methods

To write a drawing engine, you need to implement a method for reporting some of your engine's methods to QuickDraw 3D RAVE.

A pointer to your drawing engine's method reporting method is passed as a parameter to the `QARegisterEngine` function. See page 1617 for details.

TQAEngineGetMethod

A drawing engine must define a method to return pointers to some of its methods.

```
typedef TQAError (*TQAEngineGetMethod) (
                    TQAEngineMethodTag methodTag,
                    TQAEngineMethod *method);
```

`methodTag`	A selector that determines which method is to be returned about your drawing engine. See "Drawing Engine Method Selectors" (page 1568) for complete information about the available method selectors.
`method`	On exit, a pointer to your drawing engine's method of the specified type.

DESCRIPTION

Your `TQAEngineGetMethod` function is called by QuickDraw 3D RAVE to retrieve the addresses of some of your engine's methods. Your function should return, in the `method` parameter, a pointer to the drawing engine method whose type is specified by the `methodTag` parameter.

Notice Methods

Your application can define a standard notice method that is called at specific times (for example, when the renderer is finished rendering an image). You can also define a buffer notice method to handle buffer-related notifications.

A pointer to your notice method is passed as a parameter to the `QASetNoticeMethod` function.

TQAStandardNoticeMethod

An application can define a method to respond asynchronously to certain events associated with the operation of QuickDraw 3D RAVE.

```
typedef void (*TQAStandardNoticeMethod)
                    (const TQADrawContext *drawContext, void *refCon);
```

`drawContext`	A draw context.
`refCon`	The reference constant associated with the notice method.

DESCRIPTION

Your `TQAStandardNoticeMethod` function is called by QuickDraw 3D RAVE at the times specified when an application installed the notice method using the `QASetNoticeMethod` function. For example, if the value of the `method` parameter passed to `QASetNoticeMethod` was `kQAMethod_RenderCompletion`, then the standard notice method is called whenever the renderer finishes rendering an image in the draw context specified by the `drawContext` parameter. The `refCon` parameter is an application-defined reference constant; this is simply the value of the `refCon` parameter that was passed to `QASetNoticeMethod`.

Note
You can install one notice method for each defined notice selector. See page 1571 for a description of the available notice selectors. ◆

TQABufferNoticeMethod

An application can define a method to respond asynchronously to certain events associated with the operation of QuickDraw 3D RAVE buffers.

```
typedef void (*TQABufferNoticeMethod)
                    (const TQADrawContext *drawContext,
                     const TQADevice *buffer,
                     const TQARect *dirtyRect,
                     void *refCon);
```

`drawContext`	A draw context.
`buffer`	The back buffer.
`dirtyRect`	A pointer to a rectangle describing the smallest area to update. If this parameter is `NULL`, you should process the entire buffer.
`refCon`	The reference constant associated with the notice method.

DESCRIPTION

Your `TQABufferNoticeMethod` function is called by QuickDraw 3D RAVE to handle a buffer-specific notification. Currently, your buffer notice method might

receive `kQAMethod_BufferInitialize` or `kQAMethod_BufferInitialize` notifications. On entry, the `buffer` parameter is a reference to the affected buffer.

The `refCon` parameter is an application-defined reference constant; this is simply the value of the `refCon` parameter that was passed to `QASetNoticeMethod`.

Note
You can install one notice method for each defined notice selector. See page 1571 for a description of the available notice selectors. ◆

Summary of QuickDraw 3D RAVE

C Summary

Constants

Platform Values

```
#define kQAMacOS                                1
#define kQAGeneric                              2
#define kQAWin32                                3
```

Version Values

```
typedef enum TQAVersion {
    kQAVersion_Prerelease               = 0,
    kQAVersion_1_0                      = 1,
    kQAVersion_1_0_5                    = 2,
    kQAVersion_1_1                      = 3
} TQAVersion;
```

Pixel Types

```
typedef enum TQAImagePixelType {
    kQAPixel_Alpha1                     = 0,
    kQAPixel_RGB16                      = 1,
    kQAPixel_ARGB16                     = 2,
    kQAPixel_RGB32                      = 3,
    kQAPixel_ARGB32                     = 4,
    kQAPixel_CL4                        = 5,
    kQAPixel_CL8                        = 6,
```

```
    kQAPixel_RGB16_565              = 7,
    kQAPixel_RGB24                  = 8
} TQAImagePixelType;
```

Color Lookup Table Types

```
typedef enum TQAColorTableType {
    kQAColorTable_CL8_RGB32         = 0,
    kQAColorTable_CL4_RGB32         = 1
} TQAColorTableType;
```

Device Types

```
typedef enum TQADeviceType {
    kQADeviceMemory                 = 0,
    kQADeviceGDevice                = 1,
    kQADeviceWin32DC                = 2,
    kQADeviceDDSurface              = 3
} TQADeviceType;
```

Clip Types

```
typedef enum TQAClipType {
    kQAClipRgn                      = 0,
    kQAClipWin32Rgn                 = 1
} TQAClipType;
```

Tags for State Variables

```
typedef enum TQATagInt {
    kQATag_ZFunction                = 0,        /*required variables*/
    kQATag_Antialias                = 8,        /*optional variables*/
    kQATag_Blend                    = 9,
    kQATag_PerspectiveZ             = 10,
    kQATag_TextureFilter            = 11,
    kQATag_TextureOp                = 12,
    kQATag_CSGTag                   = 14,
    kQATag_CSGEquation              = 15,
    kQATag_BufferComposite          = 16,
    kQATagGL_DrawBuffer             = 100,      /*OpenGL variables*/
    kQATagGL_TextureWrapU           = 101,
```

```
    kQATagGL_TextureWrapV               = 102,
    kQATagGL_TextureMagFilter           = 103,
    kQATagGL_TextureMinFilter           = 104,
    kQATagGL_ScissorXMin                = 105,
    kQATagGL_ScissorYMin                = 106,
    kQATagGL_ScissorXMax                = 107,
    kQATagGL_ScissorYMax                = 108,
    kQATagGL_BlendSrc                   = 109,
    kQATagGL_BlendDst                   = 110,
    kQATagGL_LinePattern                = 111,
    kQATagGL_AreaPattern0               = 117,
    kQATagGL_AreaPattern31              = 148,
    kQATag_EngineSpecific_Minimum       = 1000
} TQATagInt;

typedef enum TQATagFloat {
    kQATag_ColorBG_a                    = 1,        /*required variables*/
    kQATag_ColorBG_r                    = 2,
    kQATag_ColorBG_g                    = 3,
    kQATag_ColorBG_b                    = 4,
    kQATag_Width                        = 5,
    kQATag_ZMinOffset                   = 6,
    kQATag_ZMinScale                    = 7,
    kQATagGL_DepthBG                    = 112,      /*OpenGL variables*/
    kQATagGL_TextureBorder_a            = 113,
    kQATagGL_TextureBorder_r            = 114,
    kQATagGL_TextureBorder_g            = 115,
    kQATagGL_TextureBorder_b            = 116
} TQATagFloat;

typedef enum TQATagPtr {
    kQATag_Texture                      = 13
} TQATagPtr;
```

Z Sorting Function Selectors

```
/*values for kQATag_ZFunction*/
#define kQAZFunction_None                           0
#define kQAZFunction_LT                             1
#define kQAZFunction_EQ                             2
#define kQAZFunction_LE                             3
#define kQAZFunction_GT                             4
```

```
#define kQAZFunction_NE                         5
#define kQAZFunction_GE                         6
#define kQAZFunction_True                       7
```

Antialiasing Selectors

```
/*values for kQATag_Antialias*/
#define kQAAntiAlias_Off                        0
#define kQAAntiAlias_Fast                       1
#define kQAAntiAlias_Mid                        2
#define kQAAntiAlias_Best                       3
```

Blending Operations

```
/*values for kQATag_Blend*/
#define kQABlend_PreMultiply                    0
#define kQABlend_Interpolate                    1
#define kQABlend_OpenGL                         2
```

Z Perspective Selectors

```
/*values for kQATag_PerspectiveZ*/
#define kQAPerspectiveZ_Off                     0
#define kQAPerspectiveZ_On                      1
```

Texture Filter Selectors

```
/*values for kQATag_TextureFilter*/
#define kQATextureFilter_Fast                   0
#define kQATextureFilter_Mid                    1
#define kQATextureFilter_Best                   2
```

Texture Operations

```
/*masks for kQATag_TextureOp*/
#define kQATextureOp_None                       0
#define kQATextureOp_Modulate                   (1 << 0)
#define kQATextureOp_Highlight                  (1 << 1)
#define kQATextureOp_Decal                      (1 << 2)
#define kQATextureOp_Shrink                     (1 << 3)
```

CSG IDs

```
/*values for kQATag_CSGTag*/
#define kQACSGTag_None                          0xffffffffUL
#define kQACSGTag_0                             0
#define kQACSGTag_1                             1
#define kQACSGTag_2                             2
#define kQACSGTag_3                             3
#define kQACSGTag_4                             4
```

Buffer Compositing Modes

```
#define kQABufferComposite_None                 0
#define kQABufferComposite_PreMultiply          1
#define kQABufferComposite_Interpolate          2
```

Texture Wrapping Values

```
/*values for kQATagGL_TextureWrapU and kQATagGL_TextureWrapV*/
#define kQAGL_Repeat                            0
#define kQAGL_Clamp                             1
```

Source Blending Values

```
/*values for kQATagGL_BlendSrc*/
#define kQAGL_SourceBlend_XXX                   0
```

Destination Blending Values

```
/*values for kQATagGL_BlendDst*/
#define kQAGL_DestBlend_XXX                     0
```

Buffer Drawing Operations

```
/*masks for kQATagGL_DrawBuffer*/
#define kQAGL_DrawBuffer_None                   0
#define kQAGL_DrawBuffer_FrontLeft              (1 << 0)
#define kQAGL_DrawBuffer_FrontRight             (1 << 1)
#define kQAGL_DrawBuffer_BackLeft               (1 << 2)
#define kQAGL_DrawBuffer_BackRight              (1 << 3)
#define kQAGL_DrawBuffer_Front                  \
```

```
                            (kQAGL_DrawBuffer_FrontLeft | kQAGL_DrawBuffer_FrontRight)
#define kQAGL_DrawBuffer_Back                       \
                            (kQAGL_DrawBuffer_BackLeft | kQAGL_DrawBuffer_BackRight)
```

Line and Point Widths

```
/*values for kQATag_Width*/
#define kQAMaxWidth                                 128.0
```

Vertex Modes

```
typedef enum TQAVertexMode {
    kQAVertexMode_Point                 = 0,
    kQAVertexMode_Line                  = 1,
    kQAVertexMode_Polyline              = 2,
    kQAVertexMode_Tri                   = 3,
    kQAVertexMode_Strip                 = 4,
    kQAVertexMode_Fan                   = 5,
    kQAVertexMode_NumModes              = 6
} TQAVertexMode;
```

Gestalt Selectors

```
typedef enum TQAGestaltSelector {
    kQAGestalt_OptionalFeatures         = 0,
    kQAGestalt_FastFeatures             = 1,
    kQAGestalt_VendorID                 = 2,
    kQAGestalt_EngineID                 = 3,
    kQAGestalt_Revision                 = 4,
    kQAGestalt_ASCIINameLength          = 5,
    kQAGestalt_ASCIIName                = 6,
    kQAGestalt_TextureMemory            = 7,
    kQAGestalt_FastTextureMemory        = 8,
    kQAGestalt_NumSelectors             = 9
} TQAGestaltSelector;
```

Gestalt Optional Features Response Masks

```
#define kQAOptional_None                            0
#define kQAOptional_DeepZ                           (1 << 0)
#define kQAOptional_Texture                         (1 << 1)
```

```
#define kQAOptional_TextureHQ                   (1 << 2)
#define kQAOptional_TextureColor                (1 << 3)
#define kQAOptional_Blend                       (1 << 4)
#define kQAOptional_BlendAlpha                  (1 << 5)
#define kQAOptional_Antialias                   (1 << 6)
#define kQAOptional_ZSorted                     (1 << 7)
#define kQAOptional_PerspectiveZ                (1 << 8)
#define kQAOptional_OpenGL                      (1 << 9)
#define kQAOptional_NoClear                     (1 << 10)
#define kQAOptional_CSG                         (1 << 11)
#define kQAOptional_BoundToDevice               (1 << 12)
#define kQAOptional_CL4                         (1 << 13)
#define kQAOptional_CL8                         (1 << 14)
#define kQAOptional_BufferComposite             (1 << 15)
```

Gestalt Fast Features Response Masks

```
#define kQAFast_None                            0
#define kQAFast_Line                            (1 << 0)
#define kQAFast_Gouraud                         (1 << 1)
#define kQAFast_Texture                         (1 << 2)
#define kQAFast_TextureHQ                       (1 << 3)
#define kQAFast_Blend                           (1 << 4)
#define kQAFast_Antialiasing                    (1 << 5)
#define kQAFast_ZSorted                         (1 << 6)
#define kQAFast_CL4                             (1 << 7)
#define kQAFast_CL8                             (1 << 8)
```

Vendor and Engine IDs

```
#define kQAVendor_BestChoice                    (-1)
#define kQAVendor_Apple                         0
#define kQAVendor_ATI                           1
#define kQAVendor_Radius                        2
#define kQAVendor_Mentor                        3
#define kQAVendor_Matrox                        4
#define kQAVendor_Yarc                          5
#define kQAVendor_DiamondMM                     6
```

```
#define kQAEngine_AppleSW                       0
#define kQAEngine_AppleHW                       (-1)
#define kQAEngine_AppleHW2                      1
```

Triangle Flags Masks

```
#define kQATriFlags_None                        0
#define kQATriFlags_Backfacing                  (1 << 0)
```

Texture Flags Masks

```
#define kQATexture_None                         0
#define kQATexture_Lock                         (1 << 0)
#define kQATexture_Mipmap                       (1 << 1)
#define kQATexture_NoCompression                (1 << 2)
#define kQATexture_HighCompression              (1 << 3)
```

Bitmap Flags Masks

```
#define kQABitmap_None                          0
#define kQABitmap_Lock                          (1 << 1)
#define kQABitmap_NoCompression                 (1 << 2)
#define kQABitmap_HighCompression               (1 << 3)
```

Draw Context Flags Masks

```
#define kQAContext_None                         0
#define kQAContext_NoZBuffer                    (1 << 0)
#define kQAContext_DeepZ                        (1 << 1)
#define kQAContext_DoubleBuffer                 (1 << 2)
#define kQAContext_Cache                        (1 << 3)
```

Drawing Engine Method Selectors

```
typedef enum TQAEngineMethodTag {
    kQADrawPrivateNew                           = 0,
    kQADrawPrivateDelete                        = 1,
    kQAEngineCheckDevice                        = 2,
    kQAEngineGestalt                            = 3,
    kQATextureNew                               = 4,
```

```
    kQATextureDetach                        = 5,
    kQATextureDelete                        = 6,
    kQABitmapNew                            = 7,
    kQABitmapDetach                         = 8,
    kQABitmapDelete                         = 9,
    kQAColorTableNew                        = 10,
    kQAColorTableDelete                     = 11,
    kQATextureBindColorTable                = 12,
    kQABitmapBindColorTable                 = 13
} TQAEngineMethodTag;
```

Public Draw Context Method Selectors

```
typedef enum TQADrawMethodTag {
    kQASetFloat                             = 0,
    kQASetInt                               = 1,
    kQASetPtr                               = 2,
    kQAGetFloat                             = 3,
    kQAGetInt                               = 4,
    kQAGetPtr                               = 5,
    kQADrawPoint                            = 6,
    kQADrawLine                             = 7,
    kQADrawTriGouraud                       = 8,
    kQADrawTriTexture                       = 9,
    kQADrawVGouraud                         = 10,
    kQADrawVTexture                         = 11,
    kQADrawBitmap                           = 12,
    kQARenderStart                          = 13,
    kQARenderEnd                            = 14,
    kQARenderAbort                          = 15,
    kQAFlush                                = 16,
    kQASync                                 = 17,
    kQASubmitVerticesGouraud                = 18,
    kQASubmitVerticesTexture                = 19,
    kQADrawTriMeshGouraud                   = 20,
    kQADrawTriMeshTexture                   = 21,
    kQASetNoticeMethod                      = 22,
    kQAGetNoticeMethod                      = 23
} TQADrawMethodTag;
```

Notice Method Selectors

```
typedef enum TQAMethodSelector {
    kQAMethod_RenderCompletion                  = 0,
    kQAMethod_DisplayModeChanged                = 1
    kQAMethod_ReloadTextures                    = 2,
    kQAMethod_BufferInitialize                  = 3,
    kQAMethod_BufferComposite                   = 4,
    kQAMethod_NumSelectors                      = 5
} TQAMethodSelector;
```

Data Types

Basic Data Types

```
typedef struct TQAEngine                        TQAEngine;

typedef struct TQATexture                       TQATexture;

typedef struct TQABitmap                        TQABitmap;

typedef struct TQAColorTable                    TQAColorTable;

typedef struct TQADrawPrivate                   TQADrawPrivate;
```

Memory Device Structure

```
typedef struct TQADeviceMemory {
    long                            rowBytes;
    TQAImagePixelType               pixelType;
    long                            width;
    long                            height;
    void                            *baseAddr;
} TQADeviceMemory;
```

Rectangle Structure

```
typedef struct TQARect {
    long                            left;
    long                            right;
```

```
    long                                top;
    long                                bottom;
} TQARect;
```

Macintosh Device and Clip Structures

```
typedef union TQAPlatformDevice {
    TQADeviceMemory                     memoryDevice;
    GDHandle                            gDevice;
} TQAPlatformDevice;

typedef union TQAPlatformClip {
    RgnHandle                           clipRgn;
} TQAPlatformClip;
```

Windows Device and Clip Structures

```
typedef union TQAPlatformDevice {
    TQADeviceMemory                     memoryDevice;
    HDC                                 hdc;
    struct {
        LPDIRECTDRAW                    lpDirectDraw;
        LPDIRECTDRAWSURFACE             lpDirectDrawSurface;
    };
} TQAPlatformDevice;

typedef union TQAPlatformClip {
    HRGN                                clipRgn;
} TQAPlatformClip
```

Generic Device and Clip Structures

```
typedef union TQAPlatformDevice {
    TQADeviceMemory                     memoryDevice;
} TQAPlatformDevice;

typedef union TQAPlatformClip {
    void                                *region;
} TQAPlatformClip;
```

Device Structure

```
typedef struct TQADevice {
    TQADeviceType                       deviceType;
    TQAPlatformDevice                   device;
} TQADevice;
```

Clip Data Structure

```
typedef struct TQAClip {
    TQAClipType                         clipType;
    TQAPlatformClip                     clip;
} TQAClip;
```

Image Structure

```
struct TQAImage {
    long                                width;
    long                                height;
    long                                rowBytes;
    void                                *pixmap;
};
typedef struct TQAImage TQAImage;
```

Vertex Structures

```
typedef struct TQAVGouraud {
    float                               x;
    float                               y;
    float                               z;
    float                               invW;
    float                               r;
    float                               g;
    float                               b;
    float                               a;
} TQAVGouraud;

typedef struct TQAVTexture {
    float                               x;
    float                               y;
    float                               z;
```

```
    float                           invW;
    float                           r;
    float                           g;
    float                           b;
    float                           a;
    float                           uOverW;
    float                           vOverW;
    float                           kd_r;
    float                           kd_g;
    float                           kd_b;
    float                           ks_r;
    float                           ks_g;
    float                           ks_b;
} TQAVTexture;
```

Draw Context Structure

```
struct TQADrawContext {
    TQADrawPrivate                  *drawPrivate;
    const TQAVersion                version;
    TQASetFloat                     setFloat;
    TQASetInt                       setInt;
    TQASetPtr                       setPtr;
    TQAGetFloat                     getFloat;
    TQAGetInt                       getInt;
    TQAGetPtr                       getPtr;
    TQADrawPoint                    drawPoint;
    TQADrawLine                     drawLine;
    TQADrawTriGouraud               drawTriGouraud;
    TQADrawTriTexture               drawTriTexture;
    TQADrawVGouraud                 drawVGouraud;
    TQADrawVTexture                 drawVTexture;
    TQADrawBitmap                   drawBitmap;
    TQARenderStart                  renderStart;
    TQARenderEnd                    renderEnd;
    TQARenderAbort                  renderAbort;
    TQAFlush                        flush;
    TQASync                         sync;
    TQASubmitVerticesGouraud        submitVerticesGouraud;
    TQASubmitVerticesTexture        submitVerticesTexture;
    TQADrawTriMeshGouraud           drawTriMeshGouraud;
```

```
    TQADrawTriMeshTexture               drawTriMeshTexture;
    TQASetNoticeMethod                  setNoticeMethod;
    TQAGetNoticeMethod                  getNoticeMethod;
};
typedef struct TQADrawContext TQADrawContext;
```

Drawing Engine Methods Union

```
typedef union TQAEngineMethod {
    TQADrawPrivateNew                   drawPrivateNew;
    TQADrawPrivateDelete                drawPrivateDelete;
    TQAEngineCheckDevice                engineCheckDevice;
    TQAEngineGestalt                    engineGestalt;
    TQATextureNew                       textureNew;
    TQATextureDetach                    textureDetach;
    TQATextureDelete                    textureDelete;
    TQABitmapNew                        bitmapNew;
    TQABitmapDetach                     bitmapDetach;
    TQABitmapDelete                     bitmapDelete;
    TQAColorTableNew                    colorTableNew;
    TQAColorTableDelete                 colorTableDelete;
    TQATextureBindColorTable            textureBindColorTable;
    TQABitmapBindColorTable             bitmapBindColorTable;
} TQAEngineMethod;
```

Public Draw Context Methods Union

```
typedef union TQADrawMethod {
    TQASetFloat                         setFloat;
    TQASetInt                           setInt;
    TQASetPtr                           setPtr;
    TQAGetFloat                         getFloat;
    TQAGetInt                           getInt;
    TQAGetPtr                           getPtr;
    TQADrawPoint                        drawPoint;
    TQADrawLine                         drawLine;
    TQADrawTriGouraud                   drawTriGouraud;
    TQADrawTriTexture                   drawTriTexture;
    TQADrawVGouraud                     drawVGouraud;
    TQADrawVTexture                     drawVTexture;
    TQADrawBitmap                       drawBitmap;
```

```
    TQARenderStart                      renderStart;
    TQARenderEnd                        renderEnd;
    TQARenderAbort                      renderAbort;
    TQAFlush                            flush;
    TQASync                             sync;
    TQASubmitVerticesGouraud            submitVerticesGouraud;
    TQASubmitVerticesTexture            submitVerticesTexture;
    TQADrawTriMeshGouraud               drawTriMeshGouraud;
    TQADrawTriMeshTexture               drawTriMeshTexture;
    TQASetNoticeMethod                  setNoticeMethod;
    TQAGetNoticeMethod                  getNoticeMethod;
} TQADrawMethod;
```

Indexed Triangle Structure

```
typedef struct TQAIndexedTriangle {
    unsigned long                       triangleFlags;
    unsigned long                       vertices[3];
} TQAIndexedTriangle;
```

Notice Methods

```
typedef union TQANoticeMethod {
    TQAStandardNoticeMethod             standardNoticeMethod;
    TQABufferNoticeMethod               bufferNoticeMethod;
} TQANoticeMethod;
```

QuickDraw 3D RAVE Routines

Creating and Deleting Draw Contexts

```
TQAError QADrawContextNew         (const TQADevice *device,
                                   const TQARect *rect,
                                   const TQAClip *clip,
                                   const TQAEngine *engine,
                                   unsigned long flags,
                                   TQADrawContext **newDrawContext);
void QADrawContextDelete          (TQADrawContext *drawContext);
```

Creating and Deleting Color Lookup Tables

```
TQAError QAColorTableNew          (const TQAEngine *engine,
                                   TQAColorTableType tableType,
                                   void *pixelData,
                                   long transparentIndexFlag,
                                   TQAColorTable **newTable);

void QAColorTableDelete           (const TQAEngine *engine, TQAColorTable *colorTable);
```

Manipulating Textures and Bitmaps

```
TQAError QATextureNew             (const TQAEngine *engine,
                                   unsigned long flags,
                                   TQAImagePixelType pixelType,
                                   const TQAImage images[],
                                   TQATexture **newTexture);

TQAError QATextureDetach          (const TQAEngine *engine, TQATexture *texture);

TQAError QATextureBindColorTable  (const TQAEngine *engine,
                                   TQATexture *texture,
                                   TQAColorTable *colorTable);

void QATextureDelete              (const TQAEngine *engine, TQATexture *texture);

TQAError QABitmapNew              (const TQAEngine *engine,
                                   unsigned long flags,
                                   TQAImagePixelType pixelType,
                                   const TQAImage *image,
                                   TQABitmap **newBitmap);

TQAError QABitmapDetach           (const TQAEngine *engine, TQABitmap *bitmap);

TQAError QABitmapBindColorTable   (const TQAEngine *engine,
                                   TQABitmap *bitmap,
                                   TQAColorTable *colorTable);

void QABitmapDelete               (const TQAEngine *engine, TQABitmap *bitmap);
```

Managing Drawing Engines

```
TQAEngine *QADeviceGetFirstEngine (const TQADevice *device);

TQAEngine *QADeviceGetNextEngine  (const TQADevice *device,
                                   const TQAEngine *currentEngine);

TQAError QAEngineCheckDevice      (const TQAEngine *engine, const TQADevice *device);
```

```
TQAError QAEngineGestalt          (const TQAEngine *engine,
                                   TQAGestaltSelector selector,
                                   void *response);
TQAError QAEngineEnable           (long vendorID, long engineID);
TQAError QAEngineDisable          (long vendorID, long engineID);
```

Manipulating Draw Contexts

```
#define QAGetFloat(drawContext,tag) \
                            (drawContext)->getFloat (drawContext,tag)

#define QASetFloat(drawContext,tag,newValue) \
                            (drawContext)->setFloat (drawContext,tag,newValue)

#define QAGetInt(drawContext,tag) \
                            (drawContext)->getInt (drawContext,tag)

#define QASetInt(drawContext,tag,newValue) \
                            (drawContext)->setInt (drawContext,tag,newValue)

#define QAGetPtr(drawContext,tag) \
                            (drawContext)->getPtr (drawContext,tag)

#define QASetPtr(drawContext,tag,newValue) \
                            (drawContext)->setPtr (drawContext,tag,newValue)

#define QADrawPoint(drawContext,v) \
                            (drawContext)->drawPoint (drawContext,v)

#define QADrawLine(drawContext,v0,v1) \
                            (drawContext)->drawLine (drawContext,v0,v1)

#define QADrawTriGouraud(drawContext,v0,v1,v2,flags) \
                    (drawContext)->drawTriGouraud (drawContext,v0,v1,v2,flags)

#define QADrawTriTexture(drawContext,v0,v1,v2,flags) \
                    (drawContext)->drawTriTexture (drawContext,v0,v1,v2,flags)

#define QASubmitVerticesGouraud(drawContext,nVertices,vertices) \
                (drawContext)->submitVerticesGouraud(drawContext,nVertices,vertices)

#define QASubmitVerticesTexture(drawContext,nVertices,vertices) \
                (drawContext)->submitVerticesTexture(drawContext,nVertices,vertices)
```

```
#define QADrawTriMeshGouraud(drawContext,nTriangle,triangles) \
               (drawContext)->drawTriMeshGouraud (drawContext,nTriangle,triangles)

#define QADrawTriMeshTexture(drawContext,nTriangle,triangles) \
               (drawContext)->drawTriMeshTexture (drawContext,nTriangle,triangles)

#define QADrawVGouraud(drawContext,nVertices,vertexMode,vertices,flags) \
(drawContext)->drawVGouraud (drawContext,nVertices,vertexMode,vertices,flags)

#define QADrawVTexture(drawContext,nVertices,vertexMode,vertices,flags) \
(drawContext)->drawVTexture (drawContext,nVertices,vertexMode,vertices,flags)

#define QADrawBitmap(drawContext,v,bitmap) \
                           (drawContext)->drawBitmap (drawContext,v,bitmap)

#define QARenderStart(drawContext,dirtyRect,initialContext) \
           (drawContext)->renderStart (drawContext,dirtyRect,initialContext)

#define QARenderEnd(drawContext,modifiedRect) \
                           (drawContext)->renderEnd (drawContext,modifiedRect)

#define QARenderAbort(drawContext) (drawContext)->renderAbort (drawContext)

#define QAFlush(drawContext) (drawContext)->flush (drawContext)

#define QASync(drawContext) (drawContext)->sync (drawContext)

#define QAGetNoticeMethod(drawContext, method, completionCallBack, refCon) \
    (drawContext)->getNoticeMethod (drawContext, method, completionCallBack, refCon)

#define QASetNoticeMethod(drawContext, method, completionCallBack, refCon) \
    (drawContext)->setNoticeMethod (drawContext, method, completionCallBack, refCon)
```

Registering a Custom Drawing Engine

```
TQAError QARegisterEngine         (TQAEngineGetMethod engineGetMethod);

TQAError QARegisterDrawMethod     (TQADrawContext *drawContext,
                                   TQADrawMethodTag methodTag,
                                   TQADrawMethod method);
```

Public Draw Context Methods

```
typedef float (*TQAGetFloat)      (const TQADrawContext *drawContext, TQATagFloat tag);

typedef void (*TQASetFloat)       (TQADrawContext *drawContext,
                                   TQATagFloat tag,
                                   float newValue);

typedef unsigned long (*TQAGetInt)(const TQADrawContext *drawContext, TQATagInt tag);

typedef void (*TQASetInt)         (TQADrawContext *drawContext,
                                   TQATagInt tag,
                                   unsigned long newValue);

typedef void *(*TQAGetPtr)        (const TQADrawContext *drawContext, TQATagPtr tag);

typedef void (*TQASetPtr)         (TQADrawContext *drawContext,
                                   TQATagPtr tag,
                                   const void *newValue);

typedef void (*TQADrawPoint)      (const TQADrawContext *drawContext,
                                   const TQAVGouraud *v);

typedef void (*TQADrawLine)       (const TQADrawContext *drawContext,
                                   const TQAVGouraud *v0,
                                   const TQAVGouraud *v1);

typedef void (*TQADrawTriGouraud) (const TQADrawContext *drawContext,
                                   const TQAVGouraud *v0,
                                   const TQAVGouraud *v1,
                                   const TQAVGouraud *v2,
                                   unsigned long flags);

typedef void (*TQADrawTriTexture) (const TQADrawContext *drawContext,
                                   const TQAVTexture *v0,
                                   const TQAVTexture *v1,
                                   const TQAVTexture *v2,
                                   unsigned long flags);

typedef void (*TQASubmitVerticesGouraud) (
                                   const TQADrawContext *drawContext,
                                   unsigned long nVertices,
                                   const TQAVGouraud *vertices);
```

```
typedef void (*TQASubmitVerticesTexture) (
                                const TQADrawContext *drawContext,
                                unsigned long nVertices,
                                const TQAVTexture *vertices);

typedef void (*TQADrawTriMeshGouraud) (
                                const TQADrawContext *drawContext,
                                unsigned long nTriangles,
                                const TQAIndexedTriangle *triangles);

typedef void (*TQADrawTriMeshTexture) (
                                const TQADrawContext *drawContext,
                                unsigned long nTriangles,
                                const TQAIndexedTriangle *triangles);

typedef void (*TQADrawVGouraud)   (const TQADrawContext *drawContext,
                                unsigned long nVertices,
                                TQAVertexMode vertexMode,
                                const TQAVGouraud vertices[],
                                const unsigned long flags[]);

typedef void (*TQADrawVTexture)   (const TQADrawContext *drawContext,
                                unsigned long nVertices,
                                TQAVertexMode vertexMode,
                                const TQAVTexture vertices[],
                                const unsigned long flags[]);

typedef void (*TQADrawBitmap)     (const TQADrawContext *drawContext,
                                const TQAVGouraud *v,
                                TQABitmap *bitmap);

typedef void (*TQARenderStart)    (const TQADrawContext *drawContext,
                                const TQARect *dirtyRect,
                                const TQADrawContext *initialContext);

typedef TQAError (*TQARenderEnd)  (const TQADrawContext *drawContext,
                                const TQARect *modifiedRect);

typedef TQAError (*TQARenderAbort)(const TQADrawContext *drawContext);

typedef TQAError (*TQAFlush)      (const TQADrawContext *drawContext);

typedef TQAError (*TQASync)       (const TQADrawContext *drawContext);
```

```
typedef TQAError (*TQAGetNoticeMethod) (
                                    const TQADrawContext *drawContext,
                                    TQAMethodSelector method,
                                    TQANoticeMethod *completionCallBack,
                                    void **refCon);

typedef TQAError (*TQASetNoticeMethod) (
                                    const TQADrawContext *drawContext,
                                    TQAMethodSelector method,
                                    TQANoticeMethod completionCallBack,
                                    void *refCon);
```

Private Draw Context Methods

```
typedef TQAError (*TQADrawPrivateNew) (
                                    TQADrawContext *newDrawContext,
                                    const TQADevice *device,
                                    const TQARect *rect,
                                    const TQAClip *clip,
                                    unsigned long flags);

typedef void (*TQADrawPrivateDelete) (
                                    TQADrawPrivate *drawPrivate);

typedef TQAError (*TQAEngineCheckDevice) (
                                    const TQADevice *device);

typedef TQAError (*TQAEngineGestalt) (
                                    TQAGestaltSelector selector, void *response);
```

Color Lookup Table Methods

```
typedef TQAError (*TQAColorTableNew)(
                                    TQAColorTableType pixelType,
                                    void *pixelData,
                                    long transparentIndex,
                                    TQAColorTable **newTable);

typedef void (*TQAColorTableDelete) (
                                    TQAColorTable *colorTable);
```

Texture and Bitmap Methods

```
typedef TQAError (*TQATextureNew) (unsigned long flags,
                                   TQAImagePixelType pixelType,
                                   const TQAImage images[],
                                   TQATexture **newTexture);

typedef TQAError (*TQATextureDetach) (
                                   TQATexture *texture);

typedef TQAError (*TQATextureBindColorTable) (
                                   TQATexture *texture,
                                   TQAColorTable *colorTable);

typedef void (*TQATextureDelete)  (TQATexture *texture);

typedef TQAError (*TQABitmapNew)  (unsigned long flags,
                                   TQAImagePixelType pixelType,
                                   const TQAImage *image,
                                   TQABitmap **newBitmap);

typedef TQAError (*TQABitmapDetach) (
                                   TQABitmap *bitmap);

typedef TQAError (*TQABitmapBindColorTable) (
                                   TQABitmap *bitmap,
                                   TQAColorTable *colorTable);

typedef void (*TQABitmapDelete)   (TQABitmap *bitmap);
```

Method Reporting Methods

```
typedef TQAError (*TQAEngineGetMethod) (
                                   TQAEngineMethodTag methodTag,
                                   TQAEngineMethod *method);
```

Notice Methods

```
typedef void (*TQAStandardNoticeMethod)
                                   (const TQADrawContext *drawContext, void *refCon);

typedef void (*TQABufferNoticeMethod)
                                   (const TQADrawContext *drawContext,
                                    const TQADevice *buffer,
                                    const TQARect *dirtyRect,
                                    void *refCon);
```

Result Codes

`kQANoErr`	0	No error
`kQAError`	1	Generic error code
`kQAOutOfMemory`	2	Insufficient memory for requested operation
`kQANotSupported`	3	Requested feature is not supported
`kQAOutOfDate`	4	A newer drawing engine was registered
`kQAParamErr`	5	Invalid parameter
`kQAGestaltUnknown`	6	Requested Gestalt type isn't available
`kQADisplayModeUnsupported`	7	Engine cannot render to the display in its current mode

Bibliography

Farin, Gerald, *NURB Curves and Surfaces From Projective Geometry To Practical Use,* A.K. Peters, Wellesley, MA, 1995.

Foley, J., A. van Dam, S. Feiner, and J. Hughes, *Computer Graphics: Principles and Practice,* second edition, Addison-Wesley, Reading, MA, 1990.

Foley, J., A. van Dam, S. Feiner, J. Hughes, and R. Phillips, *Introduction to Computer Graphics,* Addison-Wesley, Reading, MA, 1994.

Fraleigh, John B., and R. A. Beauregard, *Linear Algebra,* Addison-Wesley, 1987.

Glassner, A.S. ed., *Graphics Gems,* Harcourt Brace Jovanovich, Boston, 1990 and following.

Hart, John C., G. Francis, and L. Kaufman, "Visualizing Quaternion Rotation," *ACM Transactions on Computer Graphics,* vol. 13, no. 3, July 1994, 256-276.

Hearn, Donald, and M. Pauline Baker, *Computer Graphics,* second edition, Prentice-Hall, Englewood Cliffs, NJ, 1986.

Kernighan, Brian W., and Dennis M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs, NJ, 1978.

Kernighan, Brian W., and Rob Pike, *The UNIX Programming Environment,* Prentice-Hall, Englewood Cliffs, NJ, 1984.

Rogers, David F., *Procedural Elements for Computer Graphics,* McGraw-Hill Publishing Company, New York, 1985.

Rogers, David F., and J. Alan Adams, *Mathematical Elements for Computer Graphics,* McGraw-Hill Publishing Company, New York, 1990.

Vince, John, *The Language of Computer Graphics,* Van Nostrand Reinhold, New York, 1990.

Watt, Alan, *3D Computer Graphics,* second edition, Addison-Wesley, Reading, MA, 1993.

Watt, Alan, and M. Watt, *Advanced Animation and Rendering Techniques,* Addison-Wesley, Wokingham, England, 1992.

Glossary

2D Two-dimensional. See also **planar.**

3D Three-dimensional. See also **spatial.**

3DMF See **QuickDraw 3D Object Metafile.**

3D pointing device Any physical device capable of controlling movements or specifying positions in three-dimensional space.

3D Viewer A shared library that you can use to display 3D objects and other data in a window and to allow users limited interaction with those objects. See also **viewer object.**

accelerator See **graphics accelerator.**

adjoint The transpose of a matrix in which each element has been replaced by its cofactor.

adjoint matrix See **adjoint.**

affine matrix A matrix that specifies an affine transform.

affine transform Any arbitrary concatenation of scale, translate, and rotate transforms. An affine transform preserves parallel lines in the objects transformed.

aliasing The jagged edges (or staircasing) that result from drawing an image on a raster device such as a computer screen. Compare **antialiasing.**

alpha channel A color component in some color spaces whose value represents the opacity of the color defined in the other components. Compare **ARGB color structure.**

ambient coefficient A measure of an object's level of reflection of ambient light.

ambient light An amount of light of a specific color that is added to the illumination of all surfaces in a model.

ambient reflection coefficient See **ambient coefficient.**

antialiasing The smoothing of jagged edges on a displayed shape by modifying the transparencies of individual pixels along the shape's edge. Compare **aliasing.**

antialiasing quality structure A data structure that specifies data for the antialiasing quality. Defined by the `TQ3AntialiasingQuality` data type.

API See **application programming interface.**

application coordinate system See **world coordinate system.**

application space See **world coordinate system.**

application programming interface (API) The total set of constants, data structures, routines, and other programming elements that allow developers to use some part of the system software.

area A rectangular section of a plane. Defined by the `TQ3Area` data type.

ARGB color space A color space whose components measure the intensity of red, green, and blue, together with the opacity (or alpha component) of the color thus defined.

ARGB color structure A data structure that contains information about a color and its opacity. Defined by the `TQ3ColorARGB` data type.

artifact Any oddity or unwanted feature of a rendered image. Compare **aliasing.**

aspect ratio The ratio of the width of an image or other rectangular area to its height.

aspect ratio camera A type of perspective camera defined in terms of a viewing angle and a horizontal-to-vertical aspect ratio.

aspect ratio camera data structure A data structure that contains basic information about an aspect ratio camera. Defined by the `TQ3ViewAngleAspectCameraData` data type.

attenuation The loss of light intensity over distance.

attribute See **attribute object.**

attribute metahandler A metahandler that defines methods for handling custom attribute data.

attribute object A type of QuickDraw 3D object that determines some of the characteristics of a model, such as the color of objects or parts of objects in the model, the transparency of objects, and so forth. An attribute is of type `TQ3Element`. See also **ambient coefficient, diffuse color, highlight state, normal vector, shading parameterization, specular color, specular reflection exponent, standard surface parameterization, surface shader, surface tangent, transparency color.**

attribute set A collection of zero or more different attribute types and their associated data.

axis See **coordinate axis.**

back clipping plane See **yon plane.**

backface culling Ignoring backfacing polygons during rendering. Backface culling can reduce the amount of time required to render a model. Compare **hidden surface removal.**

backfacing polygon Any polygon in a surface whose surface normal points away from a view's camera.

backfacing style A type of QuickDraw 3D object that determines whether or not a renderer draws shapes that face away from a scene's camera.

background shader A shader that applies an image to the yon (or far) clipping plane of a view. Compare **foreground shader.**

badge A visual element in a frame of a 3D model displayed by the 3D Viewer that distinguishes the frame from a static image.

base class See **parent class.**

big-endian Data formatting in which each field is addressed by referring to its most significant byte. See also **little-endian.**

binary file A file object whose data is a stream of raw binary data, the type of which is indicated by object type codes. Compare **text file.**

bitmap A two-dimensional array of values, each of which represents the state of one pixel**.** Defined by the `TQ3Bitmap` data type. See also **mipmap, pixmap, storage pixmap.**

bitmap marker A type of marker that uses a bitmap to specify the image that is to be drawn on top of a rendered scene at the specified location. Defined by the `TQ3MarkerData` data type. Compare **pixmap marker.**

bounding box A rectangular box, aligned with the coordinate axes, that completely encloses an object. Defined by the `TQ3BoundingBox` data type.

bounding loop A section of code in which all bounding box or sphere calculation takes place. A bounding loop begins with a call to the `Q3View_StartBoundingBox` (or `Q3View_StartBoundingSphere`) routine and should end when a call to `Q3View_EndBoundingBox` (or `Q3View_EndBoundingSphere`) returns some value other than `kQ3ViewStatusRetraverse`. A bounding loop is a type of submitting loop. See also **picking loop, rendering loop, writing loop.**

bounding sphere A sphere that completely encloses an object. Defined by the `TQ3BoundingSphere` data type.

bounding volume A bounding box or a bounding sphere.

bounds See **bounding volume.**

box A three-dimensional object defined by an origin (that is, a corner of the box) and three vectors that define the edges of the box meeting in that corner. Defined by the `TQ3BoxData` data type.

B-spline curve A curve that passes smoothly through a series of control points.

B-spline polynomial A parametric equation that defines a B-spline curve.

B-spline surface A surface that passes smoothly through a series of control points.

camera See **camera object.**

camera coordinate system The coordinate system defined by a view's camera. Also called the *view coordinate system*. Compare **local coordinate system**, **window coordinate system, world coordinate system**.

camera data structure A data structure that contains basic information about a camera. Defined by the `TQ3CameraData` data type.

camera location The position, in the world coordinate system, of a camera. Also called the *eye point*. Compare **camera placement structure**.

camera object A type of QuickDraw 3D object that you can use to define a point of view, a range of visible objects, and a method of projection for generating a two-dimensional image of those objects from a three-dimensional model. A camera object is an instance of the `TQ3CameraObject` class. See also **aspect ratio camera, orthographic camera, view plane camera.**

camera placement The location, orientation, and direction of a camera. See also **camera placement structure**.

camera placement structure A data structure that contains information about the placement (that is, the location, orientation, and direction) of a camera. Defined by the `TQ3CameraPlacement` data type.

camera range The spatial extent that lies between the hither and yon planes of a camera. See also **camera range structure.**

camera range structure A data structure that contains information about the hither and yon clipping planes for a camera. Defined by the `TQ3CameraRange` data type.

camera space See **camera coordinate system.**

camera vector See **viewing direction.**

camera viewpoint control A control in the controller strip of a viewer object that, when held down, causes a pop-up menu to appear listing the available cameras. Compare **distance button, move button, rotate button, zoom button.**

camera view port The rectangular portion of a view plane that is to be mapped onto the area specified by the current draw context.

camera view port structure A data structure that contains information about the view port of a camera. Defined by the `TQ3CameraViewPort` data type.

cap See **end cap.**

Cartesian coordinate system A system of assigning planar positions to objects in terms of their distances from two mutually perpendicular lines (the x and y coordinate axes), or of assigning spatial positions to objects in terms of their distances from three mutually perpendicular lines (the x, y, and z coordinate axes). Compare **polar coordinate system, spherical coordinate system.**

center of projection The point at which the projectors in a perspective projection intersect.

child class A class that is immediately below some other class (the parent class) in the QuickDraw 3D class hierarchy. For example, the light class is a child class of the shape class. A child class inherits all of the methods of its parent. Also called a *subclass.*

clamp For a shader effect, to replicate the boundaries of the effect across the portion of the mapped area that lies outside the valid range 0.0 to 1.0. Compare **wrap.**

class See **QuickDraw 3D class.**

class type See **object type.**

clipping plane Either of the two planes that limit the part of a model that is rendered. See also **hither plane, yon plane.**

closed Not open. Compare **open.**

color space A specification of a particular method for representing colors. Compare **RGB color space.**

complement The set of points that lie outside a given solid object. The complement of the object A is represented by the function $\neg A$. Compare **intersection, union.**

component See **mesh component.**

concave polygon A polygon with at least one interior angle greater than 180°. Compare **convex polygon.**

cone A three-dimensional object defined by an origin (that is, the center of the base) and three vectors that define the orientation and the major and minor radii of the cone. Defined by the `TQ3ConeData` data type.

conic See **conic section.**

conic section Any two-dimensional curve that is formed by the intersection of a plane with a right circular cone. The most common conic sections are ellipses, circles, parabolas, and hyperbolas. Compare **nonuniform rational B-spline (NURB).**

connected Said of a pair of mesh vertices if an unbroken path of edges exists linking one vertex to the other. Compare **mesh component.**

constant shading A method of shading surfaces in which the incident light color and intensity are calculated for a single point on a polygon and then applied to the entire polygon. Compare **Gouraud shading, Phong shading.**

constant subdivision A method of subdividing smooth curves and surfaces. In this method, the renderer subdivides a curve into some given number of polyline segments and a surface into a certain-sized mesh of polygons. Compare **screen-space subdivision, world-space subdivision.**

constructive solid geometry (CSG) A way of modeling solid objects constructed from the union, intersection, or difference of other solid objects.

container face The face in a mesh that contains a particular contour.

contour A list of vertices. In a mesh, a contour specifies a hole in a face. Compare **container face.**

controller See **controller object.**

controller channel Any piece of information sent from an application to an input device. Compare **controller value.**

controller data structure A data structure that contains information about a controller. Defined by the `TQ3ControllerData` data type.

controller object A QuickDraw 3D object that represents a 3D pointing device. A controller object is an instance of the `TQ3ControllerObject` class. See also **tracker object.**

controller state See **controller state object.**

controller state object A QuickDraw 3D object that represents the current channels and other settings of a controller. A controller state object is an instance of the `TQ3ControllerStateObject` class.

controller strip A rectangular area at the bottom of a viewer object that contains one or more controls (usually buttons). Compare **camera viewpoint control, distance button, move button, rotate button, zoom button.**

controller value Any piece of information sent from an input device to an application. Compare **controller channel.**

control point A geometric point used to control the curvature of a curve or surface. Compare **knot.**

convex polygon A polygon whose interior angles are all less than or equal to 180°. Compare **concave polygon.**

coordinate axis A line in a plane or in space that helps to define the position of geometric objects. See also ***x* axis, *y* axis, *z* axis.**

coordinates (1) See **coordinate system.** (2) See **tracker coordinates.**

coordinate space See **coordinate system.**

coordinate system Any system of assigning planar or spatial positions to objects. Compare **Cartesian coordinate system, polar coordinate system, spherical coordinate system.**

corner See **mesh corner.**

cross product The vector that is perpendicular to two given vectors and whose magnitude is the product of the magnitudes of those two vectors multiplied by the sine of the angle between them. The cross product of the vectors u and v is denoted $u \times v$. Compare **dot product.**

CSG See **constructive solid geometry.**

CSG equation A value that encodes which CSG operations are to be performed on a model's CSG objects.

CSG object ID A number, attached to an object as an attribute, that identifies an object for CSG operations.

C standard I/O library See **standard I/O library.**

C string object A QuickDraw 3D object that contains a standard C string (that is, an array of characters terminated by the null character).

culling See **backface culling.**

custom Supplied by your application, not by QuickDraw 3D.

custom surface parameterization A parameterization of a surface supplied by your application. Compare **natural surface parameterization, standard surface parameterization.**

cylinder A three-dimensional object defined by an origin (that is, the center of the base) and three mutually perpendicular vectors that define the orientation and the major and minor radii of the cylinder. Defined by the `TQ3CylinderData` data type.

database file A metafile in which all shared objects contained in the file are listed in the file's table of contents. See also **normal file, stream file.**

database mode The mode in which a database file is opened. See also **normal mode, stream mode.**

default surface parameterization See **standard surface parameterization.**

degrees of freedom (DOF) The number of dimensions that are independently specifiable by a particular input device. For example, a slider or a dial has one degree of freedom; a mouse typically has two degrees of freedom.

device coordinate system See **window coordinate system.**

device space See **window coordinate system.**

differential scaling A scale transform in which the scaling values d_x, d_y, and d_z are not all identical. Compare **uniform scaling.**

diffuse coefficient A measure of an object's level of diffuse reflection.

diffuse color The color of the light of a diffuse reflection.

diffuse reflection The type of reflection that is characteristic of light reflected from a dull, nonshiny surface. Also called *Lambertian reflection.* Compare **specular reflection.**

diffuse reflection coefficient See **diffuse coefficient.**

direct draw surface draw context data structure A data structure that contains information about a direct draw surface draw context. Defined by the `TQ3DDSurfaceDrawContextData` data type.

directional light A light source that emits parallel rays of light in a specific direction.

directional light data structure A data structure that contains information about a directional light. Defined by the `TQ3DirectionalLightData` data type.

dirty state A Boolean value that indicates whether an unknown object is preserved in its original state (`kQ3False`) or should be updated when written back to the file object from which it was originally read (`kQ3True`).

disk A two-dimensional surface defined by an origin (that is, the center of the disk) and two vectors that define the major and minor radii of the disk. Defined by the `TQ3DiskData` data type.

display group A type of group that contains drawable objects. See also **ordered display group, proxy display group.**

distance button A button in the controller strip of a viewer object that, when clicked, puts the cursor into trucking mode. Subsequent dragging up or down in the picture area causes the object to move farther away or closer. Compare **camera viewpoint control, move button, rotate button, zoom button.**

DOF See **degrees of freedom.**

dot product The floating-point number obtained by multiplying corresponding scalar components of two vectors and then adding together all those products. The dot product of the vectors u and v is denoted $u \cdot v$. Compare **cross product.**

drawable flag A group state flag that determines whether a group is to be drawn when it is passed to a view for rendering or picking. Compare **inline flag, picking flag.**

draw context See **draw context object.**

draw context coordinate system See **window coordinate system.**

draw context data structure A data structure that contains basic information about a draw context. Defined by the `TQ3DrawContextData` data type.

draw context object A QuickDraw 3D object that maintains information specific to a particular window system or drawing destination. A draw context object is an instance of the `TQ3DrawContextObject` class. See also **Macintosh draw context, pixmap draw context, Windows draw context, X draw context.**

draw context space See **window coordinate system.**

drawing completion callback routine An application-defined function that is called whenever the 3D Viewer has finished drawing any part of a viewer object.

drawing destination The window or other output destination for a rendered model.

edge A straight line that connects two vertices. See also **mesh edge.**

edge tolerance A measure of how close a point must be to a line for a hit to occur. Compare **vertex tolerance.**

edit index A long integer associated with a shared object that changes each time the object is edited**.**

element See **element object.**

element object Any QuickDraw 3D object that can be part of a set. An element object is an instance of the `TQ3ElementObject` class.

elevation projection A type of orthographic projection in which the view plane is perpendicular to one of the principal axes of the object being projected. See also **front elevation projection, side elevation projection, top elevation projection.** Compare **isometric projection.**

ellipse A two-dimensional curve defined by an origin (that is, the center of the ellipse) and two perpendicular vectors that define the major and minor radii of the ellipse. Defined by the `TQ3EllipseData` data type.

ellipsoid A three-dimensional object defined by an origin (that is, the center of the ellipsoid) and three mutually perpendicular vectors that define the orientation and the major and minor radii of the ellipsoid. Defined by the `TQ3EllipsoidData` data type.

end cap The delimiting boundary of some QuickDraw 3D geometric objects (for instance, the oval top or bottom of a cylinder). Compare **interior cap.**

error A nonrecoverable condition that causes the currently executing QuickDraw 3D routine to fail. See also **fatal error, notice, warning.**

Error Manager The part of QuickDraw 3D that you can use to handle any errors or other exceptional conditions that occur during the execution of QuickDraw 3D routines.

even-odd rule A method of determining which planar areas defined by an arbitrary list of vertices are inside a polygon. To determine whether a particular bounded region is inside or outside a polygon, shoot a ray from any point in that region in any direction that does not intersect any vertex. If the ray cuts an odd number of edges, that region is inside the polygon; if the ray cuts an even number of edges, that region is outside the polygon.

eye point See **camera location.**

face A closed figure that forms part of the surface of an object. Usually faces are planar, but mesh faces do not need to be planar. See also **mesh face.**

face attribute An attribute that defines a characteristic of a polygonal object.

face index In a mesh, a unique integer (between 0 the total number of faces in the mesh minus 1) associated with a face. Compare **vertex index.**

facet See **face.**

faceted shading See **constant shading.**

fall-off value A measure of the attenuation of a spot light's intensity from the edge of the hot angle to the edge of the outer angle. See also **hot angle, outer angle**.

fan A strip in which all the triangles share a common vertex.

far plane See **yon plane.**

fatal error An error whose effects persist even after the call that caused it has ended.

field of view The horizontal or vertical angular expanse visible through a camera. See also **aspect ratio camera.**

file See **file object.**

file idle method A callback routine that is called during lengthy file operations. Compare **view idle method.**

file mode A set of flags that determine which operations can be performed on a piece of storage.

file mode flag A value used to construct a file mode.

file object A type of QuickDraw 3D object that you can use to access disk- or memory-based data stored in a container. A file object is an instance of the `TQ3FileObject` class. See also **storage object.**

file status value A value returned by the `Q3File_EndWrite` function that indicates whether QuickDraw 3D has finished writing the model to a file object.

fill style A type of QuickDraw 3D object that determines whether an object is drawn as a solid filled object or is decomposed into its components (namely, into a set of edges or points).

flat shading See **constant shading.**

foreground shader A shader that applies an image to the hither (or near) clipping plane of a view. Compare **background shader.**

frame See **viewer pane.**

front clipping plane See **hither plane.**

front elevation projection A type of elevation projection in which the view plane is parallel to the front of the object being projected.

frustum A solid figure created by cutting a cone or pyramid with two parallel planes. Compare **view frustum.**

frustum coordinate system See **camera coordinate system.**

frustum space See **camera coordinate system.**

frustum-to-window transform A transform that defines the relationship between a frustum coordinate system and a

window coordinate system. Compare **local-to-world transform, world-to-frustum transform**.

general polygon A closed plane figure defined by one or more lists of vertices (that is, defined by one or more contours). Defined by the `TQ3GeneralPolygonData` data type. See also **simple polygon**.

generic renderer A renderer that you can use solely to collect state information. The generic renderer does not draw any image.

geometric object A type of QuickDraw 3D object that describes a particular kind of drawable shape, such as a triangle or a box. A geometric object is an instance of the `TQ3GeometryObject` class. See also **box, cone, cylinder, disk, ellipse, ellipsoid, general polygon, line, marker, mesh, NURB curve, NURB patch, point, polygon, polyhedron, polyline, torus, triangle, trigrid, trimesh.**

geometric primitive Any of the basic geometric objects defined by QuickDraw 3D.

geometry See **geometric object.**

geometry attribute An attribute that defines a characteristic of a nonpolygonal geometric object.

global coordinate system See **world coordinate system.**

global space See **world coordinate system.**

Gouraud shading A method of shading surfaces in which the incident light color and intensity are calculated for each vertex of a polygon and then interpolated linearly across the entire polygon. Compare **constant shading, Phong shading.**

graphics accelerator Any hardware device used by QuickDraw 3D to accelerate rendering.

group **See group object.**

group object A type of QuickDraw 3D object that you can use to collect objects together into hierarchical models. A group object is an instance of the `TQ3GroupObject` class.

group position A pointer to a data structure maintained internally by QuickDraw 3D that indicates the position of a group element in the group.

group state flag A value that indicates the state of some characteristic of a group.

group state value A set of group state flags that determine how a group is traversed during rendering or picking, or during computation of its bounding box or sphere.

handle storage object A storage object that represents a handle to a dynamically allocated block of RAM.

head The hot spot of a 3D cursor. Compare **tail.**

hidden line removal The process of removing any lines in a model that are hidden by opaque surfaces of objects.

hidden surface removal The process of removing any surfaces in a model that are hidden by opaque surfaces of objects. Compare **backface culling.**

hierarchy See **QuickDraw 3D class hierarchy**.

highlight state An attribute having data of type `TQ3Boolean` that determines whether a highlight style overrides the material attributes of an object (`kQ3True`) or not.

highlight style A type of QuickDraw 3D object that determines the material attributes of a geometric object (or a group of geometric objects) that override the normal attributes of the object (or group of objects).

high-order bit See **most significant bit.**

hit An object in a model that is close enough to the pick geometry. See also **hit list.**

hit data structure A data structure that contains information about a hit. Defined by the `TQ3HitData` data type.

hither plane The clipping plane closest to the camera.

hit information mask A value that indicates the type of information you want returned for the items in a hit list.

hit list A list of all objects in a model that are close to the pick geometry.

hit list sorting value A value that determines the kind of sorting that is to be done on a hit list.

hit path structure A data structure that contains information about the path through a model hierarchy to a specific picked object. Defined by the `TQ3HitPath` data type.

hit testing See **picking.**

hot angle The half-angle (specified in radians) from the center of a spot light's cone of light within which the light remains at constant full intensity. See also **fall-off value, outer angle**.

identity matrix Any $n \times n$ square matrix with elements a_{ij} such that $a_{ij} = 1$ if $i = j$ and $a_{ij} = 0$ otherwise. Compare **inverse.**

idle method See **file idle method, view idle method.**

illumination shader A shader that determines the effects of the view's group of lights on the objects in a model. Compare **Lambert illumination shader, Phong illumination shader.**

image The two-dimensional product of rendering.

image plane structure A data structure that contains information about an image plane. Defined by the `TQ3ImagePlane` data type.

immediate mode A mode of defining and rendering a model in which the application maintains the only copy of the model data. See also **retained mode.**

immediate object An object that is rendered in immediate mode. See also **retained object.**

indexed vertex A three-dimensional vertex specified by its index into an array of three-dimensional points. Defined by the `TQ3IndexedVertex3D` data type. (Polyhedra are specified using indexed vertices.) See also **mesh vertex, vertex.**

infinite light See **directional light.**

information group A group that contains one or more strings (and no other types of QuickDraw 3D objects).

inherit To have the data and methods of a parent class apply to a child class. Compare **override.**

inheritance The property of the QuickDraw 3D class hierarchy whereby a child class inherits the data and methods of its parent class.

initial line See **polar axis.**

inline A method of executing groups that does not push and pop the graphics state stack before and after it is executed.

inline flag A group state flag that determines whether or not a group should be executed inline. Compare **drawable flag, picking flag.**

inner product See **dot product**.

input/output (I/O) The parts of a computer system that transfer data to or from peripheral devices.

instantiable class A class of which instances can be created. All leaf classes are instantiable, and many parent classes are instantiable as well. (For example, both the class `TQ3AttributeSet` and its parent class `TQ3SetObject` are instantiable.)

interacting The process of selecting and manipulating objects in a model.

interactive renderer A renderer that uses a fast and accurate algorithm for drawing solid, shaded surfaces. See also **wireframe renderer**.

interior cap The delimiting boundary of the cutaway section of a partial solid.

interpolated shading See **Gouraud shading**.

interpolation style A type of QuickDraw 3D object that determines the method of interpolation a renderer uses when applying lighting or other shading effects to a surface.

intersection The set of points that lie inside both of two given solid objects. The intersection of the objects A and B is represented by the function $A \cap B$. Compare **complement, union.**

inverse For an $n \times n$ square matrix A with a nonzero determinant, the matrix B such that $AB = BA = I$, where I is the $n \times n$ identity matrix.

inverse matrix See **inverse.**

I/O See **input/output.**

I/O proxy display group A display group that contains several representations of a single geometric object.

isometric projection A type of orthographic projection in which the view plane is not perpendicular to any of the principal axes of the object being projected but makes equal angles with each of those axes. Compare **elevation projection.**

iterative construction Constructing a mesh by building it face-by-face, instead of filling in a data structure and constructing it from the data structure all at once.

join point See **knot.**

knot A point on a curve that joins two segments of the curve.

knot vector An array of numbers that defines a curve's knots.

Lambertian reflection See **diffuse reflection.**

Lambert illumination A method of calculating the illumination of a point on a surface based on diffuse reflection. Compare **null illumination, Phong illumination.**

Lambert illumination shader An illumination shader that implements a Lambert illumination model. Compare **null illumination shader, Phong illumination shader.**

leaf class A class that has no children.

leaf object An instance of a leaf class.

leaf type The object type of a leaf object.

least significant bit (LSB) The bit contributing the least value in a string of bits. Same as *low-order bit.* Compare **most significant bit.**

left-handed coordinate system A coordinate system that obeys the left-hand rule. In a left-handed coordinate system, positive rotations of an axis are clockwise. Compare **right-handed coordinate system.**

left-hand rule A method of determining the direction of the positive *z* axis (and thereby the front of a planar surface). According to the left-hand rule, if the thumb of the left hand points in the direction of the positive *x* axis and the index finger points in the direction of the positive *y* axis, then the middle finger points in the direction of the positive *z* axis. Compare **right-hand rule.**

light See **light object.**

light attenuation See **attenuation.**

light data structure A data structure that contains basic information about a light. Defined by the `TQ3LightData` data type.

light fall-off See **fall-off value.**

light group A group that contains one or more lights (and no other types of QuickDraw 3D objects).

light object A type of QuickDraw 3D object that you can use to illuminate the surfaces in a model. A light object is an instance of the `TQ3LightObject` class. See also **ambient light, directional light, point light, spot light.**

line A straight segment in three-dimensional space defined by its two endpoints, with an optional set of attributes. Defined by the `TQ3LineData` data type.

line of projection See **projector.**

little-endian Data formatting in which each field is addressed by referring to its least significant byte. See also **big-endian.**

local coordinate system The coordinate system in which an individual geometric objects is defined. Also called the *object coordinate system* or the *modeling coordinate system*. Compare **camera coordinate system**, **window coordinate system, world coordinate system**.

local space See **local coordinate system.**

local-to-world transform A transform that defines the relationship between an object's local coordinate system and the world

coordinate system. Compare **frustum-to-window transform, world-to-frustum transform**.

low-order bit See **least significant bit.**

LSB See **least significant bit.**

luminance The intensity of light in a color.

Macintosh draw context A draw context that is associated with a Macintosh window.

Macintosh draw context data structure A data structure that contains information about a Macintosh draw context. Defined by the `TQ3MacDrawContextData` data type.

Macintosh FSSpec storage object A storage object that represents the data fork of a Macintosh file using a file system specification structure (of type `FSSpec`).

Macintosh storage object A storage object that represents the data fork of a Macintosh file using a file reference number. Compare **Macintosh FSSpec storage object.**

mapping The process of transforming one coordinate space into another.

marker A two-dimensional object typically used to indicate the position of an object (or part of an object) in a window. See **bitmap marker, pixmap marker.**

matrix A rectangular array of numbers. QuickDraw 3D defines 3-by-3 and 4-by-4 matrices using the `TQ3Matrix3x3` and `TQ3Matrix4x4` data types.

matrix transform Any transform specified by an affine, invertible 4-by-4 matrix.

memory storage object A storage object that represents a dynamically allocated block of RAM. Compare **handle storage object.**

mesh A collection of vertices, faces, and edges that represent a topological polyhedron. Defined by the `TQ3Mesh` data type.

mesh component A collection of connected vertices in a mesh. Defined by the `TQ3MeshComponent` data type.

mesh corner A mesh face together with one of its vertices. You can associate a set of attributes with a mesh corner. The attributes in a corner override any existing attributes of the associated vertex.

mesh edge A line that connects two mesh vertices. A mesh edge is part of one or more mesh faces. Defined by the `TQ3MeshEdge` data type.

mesh face A closed figure that forms part of a mesh. Unlike the faces of other geometric objects, mesh faces do not need to be planar. Defined by the `TQ3MeshFace` data type.

mesh iterator structure A data structure used by QuickDraw 3D to maintain information when iterating through parts of a mesh. Defined by the `TQ3MeshIterator` data type.

mesh part See **mesh part object.**

mesh part object A distinguishable part of a mesh. A mesh part object is an instance of the `TQ3MeshPartObject` class. See also **mesh edge part object, mesh face part object, mesh vertex part object.**

mesh vertex A vertex (that is, a three-dimensional point) that is contained in a mesh. Defined by the `TQ3MeshVertex` data type.

metafile A file format (that is, a description of the format of a kind of file). See also **QuickDraw 3D Object Metafile.**

metafile object A basic unit contained in a file that conforms to the QuickDraw 3D Object Metafile.

metahandler An application-defined function that QuickDraw 3D calls to build a method table for a custom object type. Compare **attribute metahandler.**

method An item of data associated with a particular object class. The data is usually a function pointer or other information used by the object class.

metric pick See **metric pick object.**

metric pick object A pick object whose pick geometry has a pick origin.

mipmap An array of pixel images of varying pixel depths**.** Defined by the `TQ3Mipmap` data type. See also **bitmap, pixmap, storage pixmap.**

mipmapping A method of storing texture maps in an array of pixel images of varying pixel depths. The first element in the array must be the mipmap page having the highest resolution, with a width and height that are even powers of 2. Each subsequent pixel image in the array should have a width and height that are half those of the previous image.

mipmap texture object A texture object in which the texture is defined by a mipmap.

model A collection of synthetic three-dimensional geometric objects and groups of geometric objects. A model represents a prototype.

modeling The process of creating a representation of real or abstract objects.

modeling coordinate system See **local coordinate system.**

modeling space See **local coordinate system.**

most significant bit (MSB) The bit contributing the greatest value in a string of bits. Same as *high-order bit.* Compare **least significant bit.**

move button A button in the controller strip of a viewer object that, when clicked, puts the cursor into move mode. Subsequent dragging on an object in the picture area causes the object to be moved to a new location. Compare **camera viewpoint control, distance button, rotate button, zoom button.**

MSB See **most significant bit.**

natural attribute An attribute that can naturally be contained in a set of attributes of a specific type.

natural surface parameterization A parameterization of a surface that can be derived directly from the definition of the surface. Compare **custom surface parameterization, standard surface parameterization.**

near plane See **hither plane.**

nonuniform rational B-spline (NURB) A curve defined by nonuniform parametric ratios of B-spline polynomials. NURB curves can be used to define very complex curves and surfaces, as well as very common geometric objects (for instance, the conic sections). See also **control point, knot, NURB curve, NURB patch.**

normal (a.) Perpendicular. (n.) A normal vector.

normal file A metafile in which the specification of an object in the file never occurs more than once. In other words, a file object that contains a table of contents that lists all multiply-referenced objects in the file. See also **normal file, stream file.**

normalized vector A vector whose length is 1.

normal mode The mode in which a normal file is opened. See also **database mode, stream mode.**

normal vector A vector that is normal (that is perpendicular) to a surface or planar object at a specific point.

notice A condition that is less severe than a warning, and that will likely not cause problems. See also **error, warning.**

notify function See **tracker notify function.**

null illumination A method of calculating the illumination of a point on a surface that depends only on the diffuse color of the point. Compare **Lambert illumination, Phong illumination.**

null illumination shader An illumination shader that implements a null illumination model. Compare **Lambert illumination shader, Phong illumination shader.**

NURB See **nonuniform rational B-spline.**

NURB curve A three-dimensional curve represented by a NURB equation. Defined by the `TQ3NURBCurveData` data type.

NURB patch A three-dimensional surface represented by a NURB equation. Defined by the `TQ3NURBPatchData` data type.

object (1) See **QuickDraw 3D object.** (2) See **metafile object.**

object coordinate system See **local coordinate system.**

object space See **local coordinate system.**

object type The identifier of the class of which a QuickDraw 3D object is an instance. Also called the *class type.*

oblique projection A type of parallel projection in which the view plane is not perpendicular to the viewing direction. Compare **orthographic projection.**

off-axis viewing A method of perspective projection in which the center of the projected object on the view plane is not on the camera vector.

opaque (1) For a data structure, not publicly defined. You must use QuickDraw 3D functions to get and set values in an opaque data structure. For an object, having data and methods that are not publicly defined. (2) For a geometric object, not allowing light to pass though.

open Said of a storage object whenever its associated storage is in use—for example, when an application is reading data from a file object attached to the storage object.

order For a NURB curve or patch, one more than the highest degree equation used to define the curve or patch. For example, the order of a NURB curve defined by cubic polynomial equations is 4.

ordered display group A display group in which the objects in the group are sorted by their type.

orientation style A type of QuickDraw 3D object that determines which side of a planar surface is considered to be the "front" side.

origin In Cartesian coordinates, the point (0, 0) or (0, 0, 0). The coordinate axes intersect at the origin.

original QuickDraw See **QuickDraw.**

orthogonal Perpendicular.

orthographic camera A type of camera that uses orthographic projection.

orthographic camera data structure A data structure that contains basic information about an orthographic camera. Defined by the `TQ3OrthographicCameraData` data type.

orthographic projection A type of parallel projection in which the view plane is perpendicular to the viewing direction. Compare **oblique projection.** See also **elevation projection, isometric projection.**

outer angle The half-angle (specified in radians) from the center of a spot light's cone to the edge of the cone. See also **fall-off value, hot angle.**

outer product See **cross product**.

override To define class data or methods that replace those of the parent class. Compare **inherit.**

parallel projection A method of projecting a model onto a viewing plane that uses parallel projectors. See also **oblique projection, orthographic projection.** Compare **perspective projection.**

parameterization A parametric function that picks out all points on a geometric object, such as a pixmap or a surface. Compare **surface parameterization.**

parametric curve Any curve whose points are described by one or more parametric functions. A two-dimensional parametric curve can be described by the parametric functions $x = x(t)$ and $y = y(t)$. A three-dimensional parametric curve is described by the parametric functions $x = x(t)$, $y = y(t)$, and $z = z(t)$. Compare **B-spline polynomial, nonuniform rational B-spline (NURB).**

parametric equation See **parametric function.**

parametric function A function of one or more parameters (often denoted by s and t or u and v).

parametric point A position in two- or three-dimensional space picked out by a parametric function. Defined by the `TQ3Param2D` and `TQ3Param3D` data types. Compare **point, point object, rational point.**

parent class The class (if any) of which a given class is a subclass. In other words, a class' parent class is the class immediately above that class in the QuickDraw 3D class

hierarchy. For example, the shape class is the parent class of the light class. Also called a *base class* or a *superclass.*

partial solid A solid object whose `uMin` field is greater than 0.0 or whose `uMax` field is less than 1.0.

patch A portion of a surface defined by a set of points. Compare **NURB patch.**

perspective foreshortening A feature of perspective projections wherein the size of a projected object varies inversely with the distance of the object from the center of projection.

perspective projection A method of projecting a model onto a viewing plane that uses nonparallel projectors. Compare **parallel projection.**

Phong illumination A method of calculating the illumination of a point on a surface based on both diffuse reflection and specular reflection. Compare **Lambert illumination, null illumination.**

Phong illumination shader An illumination shader that implements a Phong illumination model. Compare **Lambert illumination shader, null illumination shader.**

Phong shading A method of shading surfaces in which the incident light color and intensity are calculated for a series of points along each edge of a polygon and then interpolated across the entire polygon. Compare **constant shading, Gouraud shading.**

pick (n.) See **pick object.** (v.) To determine whether a specified object is close enough to a pick geometry for a hit to be recorded.

pick data structure A data structure that contains basic information about a pick object. Defined by the `TQ3PickData` data type.

pick detail See **hit information mask.**

pick geometry The geometric object used in any picking method.

pick hit See **hit.**

pick hit list See **hit list.**

picking The process of identifying the objects in a view that are close to a specified geometric object.

picking flag A binary flag in a group state value that determines whether a group is eligible for picking. Compare **drawable flag, inline flag.**

picking ID An arbitrary 32-bit value that you can use to determine which object was selected by a pick operation.

picking ID style A type of style object that determines the picking ID of an object or group of objects in a model.

picking loop A section of code in which all picking takes place. A picking loop begins with a call to the `Q3View_StartPicking` routine and should end when a call to `Q3View_EndPicking` returns some value other than `kQ3ViewStatusRetraverse`. A picking loop is a type of submitting loop. See also **bounding loop, rendering loop, writing loop.**

picking parts style A type of QuickDraw 3D object that determines which parts of a geometric object (for instance, a mesh) are eligible for inclusion in a hit list.

pick object A QuickDraw 3D object that is used to select geometric objects in a model that are close to a specified geometric object. A pick object is an instance of the `TQ3PickObject` class.

pick origin A point in space that determines the origin of sorting hits. Compare **metric pick object.**

pick parts mask A value that indicates the kinds of objects placed in a hit list.

picture area The portion of a window occupied by a viewer object that contains the displayed image.

pixel image See **pixmap**

pixel map See **pixmap**

pixmap A two-dimensional array of values, each of which represents the color of one pixel. Defined by the `TQ3Pixmap` data type. See also **bitmap, mipmap, storage pixmap.**

pixmap draw context A draw context that is associated with a pixmap.

pixmap draw context data structure A data structure that contains information about a pixmap draw context. Defined by the `TQ3PixmapDrawContextData` data type.

pixmap marker A type of marker that uses a pixmap to specify the image that is to be drawn on top of a rendered scene at the specified location. Defined by the `TQ3PixmapMarkerData` data type. Compare **bitmap marker.**

pixmap texture object A texture object in which the texture is defined by a pixmap.

planar Contained completely in two dimensions (as, for example, a circle). See also **spatial.**

plane constant The value d in the plane equation $ax+by+cz+d = 0$.

plan elevation projection See **top elevation projection.**

plane equation An equation that defines a plane. A plane equation can always be reduced to the form $ax+by+cz+d = 0$. Defined by the `TQ3PlaneEquation` data type.

point A dimensionless position in two- or three-dimensional space. Defined by the `TQ3Point2D` and `TQ3Point3D` data types. Compare **parametric point, point object, rational point.**

point light A light source that emits rays of light in all directions from a specific location.

point light data structure A data structure that contains information about a point light. Defined by the `TQ3PointLightData` data type.

point object A dimensionless position in three-dimensional space, with an optional set of attributes. Defined by the `TQ3PointData` data type.

point of interest The point in world space at which a camera is aimed. The point of interest and the camera location determine the viewing direction.

point pick object See **window-point pick object.**

polar coordinate system A system of assigning planar positions to objects in terms of their distances (r) from a point (the

polar origin, or pole) along a ray that forms a given angle (θ) with a coordinate line (the polar axis). The polar origin has the polar coordinates (0, θ), for any angle θ. Compare **Cartesian coordinate system, spherical coordinate system.**

polar axis A fixed ray that radiates from the polar origin, in terms of which polar coordinates are determined. Also called the *initial line*.

polar origin The point in a plane from which the polar axis radiates. Also called the *pole*.

polar point A point in a plane described using polar coordinates.

pole See **polar origin.**

polygon A closed plane figure. See **general polygon, simple polygon.**

polygon mesh A mesh whose faces are composed of polygons.

polyhedral primitive A three-dimensional surface composed of polygonal faces that share edges and vertices with other faces. See **mesh, polyhedron, trigrid, trimesh.**

polyhedron (1) A polyhedral primitive whose faces are triangular. Defined by the `TQ3PolyhedronData` data type. (2) Any polyhedral primitive.

polyhedron data structure A data structure that contains information about a polyhedron. Defined by the `TQ3PolyhedronData` data type.

polyhedron edge data structure A data structure that contains information about an edge in a polyhedron. Defined by the `TQ3PolyhedronEdgeData` data type.

polyhedron triangle data structure A data structure that contains information about a triangle (that is, a face) in a polyhedron. Defined by the `TQ3PolyhedronTriangleData` data type.

polyline A collection of n lines defined by the n+1 points that define the endpoints of each line segment. Defined by the `TQ3PolyLineData` data type.

postmultiplied A term that describes the order in which matrices are multiplied. Matrix [A] is postmultiplied by matrix [B] if matrix [A] is replaced by [A] × [B]. Compare **premultiplied.**

premultiplied A term that describes the order in which matrices are multiplied. Matrix [A] is premultiplied by matrix [B] if matrix [A] is replaced by [B] × [A]. Compare **postmultiplied.**

primitive See **geometric primitive.**

private See **opaque.**

projection (1) A method of mapping three-dimensional objects into two dimensions. See also **parallel projection, perspective projection.** Compare **camera object.** (2) The image on the view plane that results from mapping three-dimensional objects into two dimensions.

projection plane See **view plane.**

projective transform See **frustum-to-window transform.**

projector A ray that intersects both a point on an object in a model and the view plane, thereby projecting the object in the model onto the view plane.

prototype The object (or collection of objects) represented in a model. Compare **model, synthetic.**

prototypical Of or pertaining to a prototype. Compare **model, synthetic.**

proxy display group See **I/O proxy display group.**

quaternion A quadruple of floating-point numbers that obeys the laws of quaternion arithmetic. Defined by the `TQ3Quaternion` data type.

quaternion transform A type of transform that rotates and twists an object according to the mathematical properties of quaternions.

QuickDraw A collection of system software routines on Macintosh computers that perform two-dimensional drawing on the user's screen.

QuickDraw 3D A graphics library developed by Apple Computer, Inc., that you can use to create, configure, render, and interact with models of three-dimensional objects. You can also use QuickDraw 3D to read and write 3D data.

QuickDraw 3D class A structure for the data that characterize QuickDraw 3D objects, together with a set of methods that operate on that data. Compare **QuickDraw 3D object.** See also **child class, leaf class, parent class.**

QuickDraw 3D class hierarchy The hierarchical arrangement of QuickDraw 3D object classes.

QuickDraw 3D object Any instance of a QuickDraw 3D class. See also **object type.**

QuickDraw 3D Object Metafile (3DMF) An extensible file format defined by Apple Computer, Inc., for storing 3D data and interchanging 3D data between applications.

QuickDraw 3D Pointing Device Manager A set of functions that you can use to manage three-dimensional pointing devices.

QuickDraw 3D shading architecture An environment in which shaders can be applied at various stages in the imaging pipeline.

radius vector The ray that radiates from the polar origin and that forms a given angle with the polar axis (or two given angles with the *x* and *z* axes). A polar or spherical point lies at a given distance along the radius vector. See also **polar coordinate system, spherical coordinate system.**

rasterization The process of determining values for the pixels in a rendered image. Also called *scan conversion*.

rational point A dimensionless position in two- or three-dimensional space together with a floating-point weight. Defined by the `TQ3RationalPoint3D` and `TQ3RationalPoint4D` data types. Compare **point.**

ray A point of origin and a direction. Defined by the `TQ3Ray3D` data type.

real See **prototypical.**

rectangle pick object See **window-rectangle pick object.**

reference count The number of times a shared object is being accessed.

render To create an image (on the screen or some other medium) of a model.

renderer See **renderer object.**

renderer feature flags A set of flags that encode information about a specific renderer.

renderer object A QuickDraw 3D object that you can use to render a model—that is, to create an image from a view and a model. A renderer object is an instance of the `TQ3RendererObject` class.

rendering The process of creating an image (on the screen or some other medium) of a model. See also **rasterization.**

rendering loop A section of code in which all rendering takes place. A rendering loop begins with a call to the `Q3View_StartRendering` routine and should end when a call to `Q3View_EndRendering` returns some value other than `kQ3ViewStatusRetraverse`. A rendering loop is a type of submitting loop. See also **bounding loop, picking loop, writing loop.**

retained mode A mode of defining and rendering a model in which the graphics library (for instance, QuickDraw 3D) maintains a copy of the model. See also **immediate mode.**

retained object An object that is defined and rendered in retained mode. See also **immediate object.**

RGB color space A color space whose three components measure the intensity of red, green, and blue.

RGB color structure A data structure that contains information about a color. Defined by the `TQ3ColorRGB` data type.

right-handed coordinate system A coordinate system that obeys the right-hand rule. In a right-handed coordinate system, positive rotations of an axis are counterclockwise. Compare **left-handed coordinate system.**

right-hand rule A method of determining the direction of the positive *z* axis (and thereby the front of a planar surface). According to the right-hand rule, if the thumb of the right hand points in the direction of the positive *x* axis and the index finger points in the direction of the positive *y* axis, then the middle finger points in the direction of the positive *z* axis. Compare **left-hand rule.**

rotate To reposition an object by revolving (or turning) each point of the object by the same angle around a point or axis.

rotate-about-axis transform A type of transform that rotates an object about an arbitrary axis in space by a specified number of radians at an arbitrary point in space.

rotate-about-axis transform data structure A data structure that contains information on a rotate transform about an arbitrary axis in space at an arbitrary point in space. Defined by the `TQ3RotateAboutAxisTransformData` data type.

rotate-about-point transform A type of transform that rotates an object about the *x*, *y*, or *z* axis by a specified number of radians at an arbitrary point in space.

rotate-about-point transform data structure A data structure that contains information on a rotate transform about an arbitrary point in space. Defined by the `TQ3RotateAboutPointTransformData` data type.

rotate button A button in the controller strip of a viewer object that, when clicked, puts the cursor into rotate mode. Subsequent dragging of the cursor in the picture area causes the displayed object to rotate in the direction in which the cursor is dragged. Compare **camera viewpoint control, distance button, move button, zoom button.**

rotate transform A type of transform that rotates an object about the *x*, *y*, or *z* axis at the origin by a specified number of radians.

rotate transform data structure A data structure that contains information about a rotate transform. Defined by the `TQ3RotateTransformData` data type.

rotation A transform that causes an object to revolve around a point or an axis. Compare **rotate-about-axis transform, rotate-about-point transform, rotate transform.**

scalar product See **dot product**.

scale To reposition and resize an object by multiplying the *x*, *y*, and *z* coordinates of each of its points by values d_x, d_y, and d_z. Compare **differential scaling, uniform scaling.**

scale transform A type of transform that scales an object along the *x*, *y*, and *z* axes by specified values.

scan conversion See **rasterization.**

scene A combination of objects, lights, and draw context.

screen coordinate system See **window coordinate system.**

screen space See **window coordinate system.**

screen-space picking The process of testing whether the projections of three-dimensional objects onto the screen intersect or are close enough to a specified two-dimensional object on the screen.

screen-space subdivision A method of subdividing smooth curves and surfaces. In this method, the renderer subdivides a curve (or surface) into polylines (or polygons) whose sides have a maximum length of some specified number of pixels. Compare **constant subdivision, world-space subdivision.**

serpentine Said of a trigrid in which quadrilaterals are divided into triangles in an alternating fashion.

set See **set object.**

set object A collection of zero or more elements, each of which has both an element type and some associated element data. A set object is an instance of the `TQ3SetObject` class.

shader See **shader object.**

shader object A type of QuickDraw 3D object that you can use to manipulate visual effects that depend on the illumination provided by a view's group of lights, the color and other material properties (such as the reflectance and texture) of surfaces in a model, and the position and orientation of

the lights and objects in a model. A shader object is an instance of the `TQ3ShaderObject` class.

shading parameterization A surface *uv* parameterization used when shading a surface.

shadow-receiving style A type of QuickDraw 3D object that determines whether or not objects in a model receive shadows when obscured by other objects in the model.

shape See **shape object.**

shape hint A data item associated with a general polygon that that specifies the shape of the general polygon.

shape object A type of QuickDraw 3D object that affects how and where a renderer renders an object in a view. A shape object is an instance of the `TQ3ShapeObject` class.

shape part See **shape part object.**

shape part object A distinguishable part of a shape object. A shape part object is an instance of the `TQ3ShapePartObject` class. See also **mesh part object.**

shared object A QuickDraw 3D object that may be referenced by many objects or the application at the same time. A shared object is an instance of the `TQ3SharedObject` class.

side elevation projection A type of elevation projection in which the view plane is parallel to a side of the object being projected.

simple polygon A closed plane figure defined by a list of vertices (that is, defined by a single contour). Defined by the `TQ3PolygonData` data type. See also **general polygon**.

smooth shading See **Gouraud shading, Phong shading**.

space (1) See **coordinate system.** (2) The two- or three-dimensional extent defined by a coordinate system.

spatial Contained completely in three dimensions (as, for example, a box). See also **planar.**

specular coefficient A measure of an object's level of specular reflection.

specular color The color of the light of a specular reflection.

specular control See **specular reflection exponent.**

specular exponent See **specular reflection exponent.**

specular highlight A bright area on an object's surface caused by specular reflection.

specular reflection The type of reflection that is characteristic of light reflected from a shiny surface. Compare **diffuse reflection.**

specular reflection coefficient See **specular coefficient.**

specular reflection exponent A value that determines how quickly the specular reflection diminishes as the viewing direction moves away from the direction of reflection.

spherical coordinate system A system of assigning spatial positions to objects in terms of their distances from the origin (ρ) along a ray that forms a given angle (θ) with the x axis and another angle (ϕ) with the z axis. Compare **Cartesian coordinate system, polar coordinate system.**

spherical point A point in space described using spherical coordinates.

spot light A light source that emits a circular cone of light in a specific direction from a specific location. See also **fall-off value, hot angle, outer angle**.

spot light data structure A data structure that contains information about a spot light. Defined by the `TQ3SpotLightData` data type.

standard I/O library A collection of functions that provide character I/O and file manipulation services for C programs. Compare **UNIX storage object.**

standard surface parameterization A parametric function that maps the unit square to an object's surface. Compare **custom surface parameterization, natural surface parameterization.**

storage object A QuickDraw 3D object that represents any piece of storage in a computer (for example, a file on disk, an area of memory, or some data on the Clipboard). A storage object is an instance of the `TQ3StorageObject` class.

storage pixmap A two-dimensional array of values contained in a storage object, each of which represents the color of one pixel. Defined by the `TQ3StoragePixmap` data type. See also **bitmap, mipmap, pixmap.**

stream file A metafile that contains no internal references. In other words, a file object that does not contain a table of contents and in which any references to objects are simply copies of the objects themselves. See also **normal file, stream file.**

stream mode The mode in which a stream file is opened. See also **database mode, normal mode.**

string See **string object.**

string object A QuickDraw 3D object that contains a sequence of characters. A string object is an instance of the `TQ3StringObject` class. See also **C string object.**

strip A surface composed of triangles that are ordered sequentially (that is, each triangle has one edge in common with the previous neighboring triangle, a second edge in common with the next neighboring triangle, and the remaining edge in common with no other triangle). See also **fan.**

style See **style object.**

style object A type of QuickDraw 3D object that determines some of the basic characteristics of the renderer used to render the curves and surfaces in a scene. A style object is an instance of the `TQ3StyleObject` class.

subclass See **child class.**

subdivision method A method of subdividing smooth curves and surfaces. See **constant subdivision, screen-space subdivision, world-space subdivision.**

subdivision method specifier An indicator of the number of parts into which a smooth curve or surface is to be subdivided.

subdivision style A type of QuickDraw 3D object that determines how a renderer decomposes smooth curves and surfaces into polylines and polygonal meshes for display purposes.

subdivision style data structure A data structure that contains information about the type of subdivision of curves and surfaces used by a renderer. Defined by the `TQ3SubdivisionStyleData` data type.

submit To make an object (or group of objects) eligible for drawing, picking, writing, or bounding box or sphere calculation. Compare **submitting loop.**

submitting loop A section of code in which all submitting takes place. Compare **bounding loop, picking loop, rendering loop, writing loop.**

superclass See **parent class.**

surface-based shader A shader that affects the surfaces of geometric objects based on their material properties, position, and orientation (and other factors). Compare **view-based shader.**

surface parameterization A parametric function that picks out all points on a surface. See also **custom surface parameterization, natural surface parameterization, standard surface parameterization.**

surface normal See **normal vector.**

surface shader A shader that is applied when calculating the appearance of a surface. Compare **texture shader.**

surface tangent A pair of vectors that indicate the directions of changing u and v parameters on a surface. Defined by the `TQ3Tangent2D` data type.

surrounding light See **ambient light.**

synthetic Not real, as for example the objects in a model. Compare **prototypical.**

synthetic camera See **camera object.**

tail The part of a 3D cursor at the point (1, 0, 0) in the local coordinates of the cursor. Compare **head.**

tangent A line or plane that intersects a curve or surface at a single point. Compare **surface tangent.**

tessellate To decompose a curve or surface into polygonal faces.

text file A file object whose data is a stream of ASCII characters with meaningful labels for each type of object contained in the file. Compare **binary file.**

texture See **texture object.**

texture mapping A technique wherein a predefined image (the texture) is mapped onto the surface of an object in a model.

texture object A type of QuickDraw 3D object used to perform texture mapping. Compare **mipmap texture object, pixmap texture object.**

texture parameterization A parametric function that maps the unit square to a texture.

texture shader A type of surface shader that applies textures to surfaces.

tolerance See **edge tolerance, vertex tolerance.**

top elevation projection A type of elevation projection in which the view plane is parallel to the top of the object being projected. Also called *plan elevation projection.*

topological modification The process of adding and deleting vertices, faces, edges, and other components in a geometric object, such as a mesh.

torus A three-dimensional object formed by the rotation of an ellipse about an axis in the plane of the ellipse that does not cut the ellipse. Defined by the `TQ3TorusData` data type.

tracker See **tracker object.**

tracker coordinates The current settings (that is, position and orientation) of a tracker.

tracker notify function A function that is called whenever the coordinates of a tracker change by more than a specified amount.

tracker object A QuickDraw 3D object that represents the position and orientation of a single element in your application's user interface. A tracker object is an instance of the `TQ3TrackerObject` class. See also **controller object.**

tracker serial number A unique number that changes every time the coordinates of a tracker are updated by a controller.

tracker threshold The amount by which a tracker's coordinates must change for the tracker notify function to be called.

transform See **transform object.**

transform object A type of QuickDraw 3D object that you can use to modify or transform the appearance or behavior of a QuickDraw 3D object. A transform object is an instance of the `TQ3TransformObject` class.

translate To reposition an object by adding values d_x, d_y, and d_z to the x, y, and z coordinates of each of its points.

translate transform A type of transform that translates an object along the x, y, and z axes by specified values.

transparency The ability of an object to allow light to pass through it.

transparency color A color of type `TQ3ColorRGB` that determines the amount of light that can pass through a surface. The color (0, 0, 0) indicates complete transparency, and (1, 1, 1) indicates complete opacity.

transpose (n.) For an $m \times n$ matrix with elements a_{ij}, the $n \times m$ matrix with elements b_{ij} such that $b_{ij} = a_{ji}$. (v.) To form the transpose of a given matrix.

transpose matrix See **transpose.**

triangle A closed plane figure defined by three edges. Defined by the `TQ3TriangleData` data type.

triangular mesh See **trimesh.**

trigrid A grid composed of triangular facets. Defined by the `TQ3TriGridData` data type.

trimesh A collection of vertices, edges, and faces in which all faces are triangular. (In other words, a trimesh is simply a mesh composed entirely of triangles.) Defined by the `TQ3TriMeshData` data type.

trimesh attributes data structure A data structure that contains information about the attributes of a trimesh vertex, edge, or face. Defined by the `TQ3TriMeshAttributeData` data type.

trimesh data structure A data structure that contains information about a trimesh. Defined by the `TQ3TriMeshData` data type.

trimesh edge data structure A data structure that contains information about an edge in a trimesh. Defined by the `TQ3TriMeshEdgeData` data type.

trimesh triangle data structure A data structure that contains information about a triangle (that is, a face) in a trimesh. Defined by the `TQ3TriMeshTriangleData` data type.

type See **object type.**

under-color shader A shader associated with some other shader that supplies an under color for surfaces shaded by that shader.

uniform scaling A scale transform in which the scaling values d_x, d_y, and d_z are all identical. Compare **differential scaling.**

union The set of points that lie inside either of two given solid objects. The union of the objects A and B is represented by the function $A \cup B$. Compare **complement, intersection.**

unit cube A box whose three defining edges have a length of 1.

unit vector See **normalized vector.**

UNIX path name storage object A storage object that represents a file using a path name.

UNIX storage object A storage object that represents a file using a structure of type `FILE` (defined in the standard I/O library). Compare **UNIX path name storage object.**

unknown object A type of QuickDraw 3D object that is created when QuickDraw 3D encounters data it doesn't recognize while reading a metafile. An unknown object is an instance of the `TQ3UnknownObject` class.

up vector A vector that indicates which direction is up. A camera has an up vector that defines its orientation. Compare **camera placement.**

user interface view See **user interface view object.**

user interface view notify function A function that is called whenever one of your user interface views needs to be redrawn.

user interface view object A type of view that allows the user to interact (using interface elements such as a 3D cursor or widgets) with the 3D objects displayed in the view. A user interface view object is an instance of the `TQ3UIViewObject` class.

valid range The range of u and v parametric values for a standard surface parameterization. For QuickDraw 3D, the valid range is the closed interval [0.0, 1.0].

vector A pair or triple of floating-point numbers that obeys the laws of vector arithmetic. Defined by the `TQ3Vector2D` and `TQ3Vector3D` data types. Compare **cross product**, **dot product, normal.**

vector-normal interpolation shading See **Phong shading**.

vector product See **cross product**.

vertex A dimensionless position in three- or four-dimensional space at which two or more lines (for instance, edges) intersect, with an optional set of vertex attributes. Defined by the `TQ3Vertex3D` and `TQ3Vertex4D` data types. See also **indexed vertex, mesh vertex.**

vertex attribute An attribute that defines a characteristic of a vertex of a polygonal object.

vertex index In a mesh, a unique integer (between 0 the total number of vertices in the mesh minus 1) associated with a vertex. Compare **face index.**

vertex tolerance A measure of how close two points must be for a hit to occur. Compare **edge tolerance.**

view See **view object.**

view attribute An attribute that defines a characteristic of a view object.

view-based shader A shader that operates independently of the material properties or orientation of objects (in other words, that operates solely on aspects of the view, such as the camera position). Compare **surface-based shader.**

viewing box The rectangular box defined by an orthographic camera and the hither and yon clipping planes. Compare **viewing frustum.**

view coordinate system See **camera coordinate system.**

viewer See **viewer object.**

Viewer See **3D Viewer.**

viewer badge See **badge.**

viewer controller strip See **controller strip.**

viewer flags A set of bit flags that specify information about the appearance and behavior of a viewer object.

viewer frame See **viewer pane.**

viewer object An instance of the 3D Viewer. A viewer object is of type `ViewerObject`.

viewer pane The portion of a window occupied by a viewer object. The pane includes the picture area and the controller strip.

viewer state flags A set of bit flags returned by the `Q3ViewerGetState` function that specify information about the current state of a viewer object.

view hints object An object in a metafile that gives hints about how to render a scene.

view idle method A callback routine that is called during lengthy rendering operations. Compare **file idle method.**

view information structure A data structure that contains information about a view. Defined by the `TQ3ViewInfo` data type.

viewing box A rectangle defined by a perspective camera and the hither and yon clipping planes. Compare **viewing frustum.**

viewing direction The direction of a view's camera. Also called the *camera vector* or the *viewing vector.*

viewing frustum A nonrectangular frustum defined by a perspective camera and the hither and yon clipping planes. Compare **viewing box.**

viewing vector See **viewing direction.**

view mapping matrix A matrix maintained by QuickDraw 3D that transforms the viewing frustum into a standard rectangular solid. The world-to-frustum transform is the product of the transforms specified by the view orientation matrix and the view mapping matrix. Compare **view orientation matrix.**

view object A type of QuickDraw 3D object used to collect state information that controls the appearance and position of objects at the time of rendering. A view object is an instance of the `TQ3ViewObject` class.

view orientation matrix A matrix maintained by QuickDraw 3D that rotates and translates a view's camera so that it is pointing down the negative *z* axis. The world-to-frustum transform is the product of the transforms specified by the view orientation matrix and the view mapping matrix. Compare **view mapping matrix.**

view plane The plane onto which a model is projected. Also called the *projection plane.*

view plane camera A type of perspective camera defined in terms of an arbitrary view plane.

view plane camera data structure A data structure that contains basic information about a view plane camera. Defined by the `TQ3ViewPlaneCameraData` data type.

view plane coordinate system The two-dimensional coordinate system whose origin is the point at which the viewing direction intersects the view plane and whose positive *y* axis is parallel to the camera's up vector.

view port See **camera view port.**

view space See **camera coordinate system.**

view status value A value returned by the `Q3View_EndRendering` function that indicates whether the renderer has finished processing the model.

view volume The part of world space that is projected onto the view plane during rendering. See also **view box, view frustum.**

virtual See **synthetic.**

virtual camera See **camera object.**

visual line determination See **hidden line removal**.

visual surface determination See **hidden surface removal**.

warning A condition that, though less severe than an error, might cause an error if your application continues execution without handling the warning. See also **error, notice.**

widget An element of an application's 3D user interface.

window coordinate system The coordinate system defined by a window. Also called the *screen coordinate system* or the *draw context coordinate system*. Compare **camera coordinate system**, **local coordinate system, world coordinate system**.

window picking See **screen-space picking.**

window-point pick data structure A data structure that contains information about a window-point pick object. Defined by the `TQ3WindowPointPickData` data type.

window-point pick object A pick object that tests for closeness between a point in a window and the screen projections of the objects in the model.

window-rectangle pick data structure A data structure that contains information about a window-rectangle pick object. Defined by the `TQ3WindowRectPickData` data type.

window-rectangle pick object A pick object that tests for closeness between a rectangle in a window and the screen projections of the objects in the model.

Windows 32 draw context A draw context that is associated with a window in a Windows computer.

Windows 32 draw context data structure A data structure that contains information about a Windows 32 draw context. Defined by the `TQ3Win32DCDrawContextData` data type.

window space See **window coordinate system.**

wireframe renderer A renderer that creates line drawings of models. See also **interactive renderer**.

world coordinate system The coordinate system that defines the locations of all geometric objects as they exist at rendering or picking time, with all applicable transforms acting on them. Also called the *global coordinate system* or the *application coordinate system*. Compare **camera coordinate system**, **local coordinate system, window coordinate system**.

world space See **world coordinate system.**

world-space subdivision A method of subdividing smooth curves and surfaces according to which the renderer subdivides a curve (or surface) into polylines (or polygons) whose sides have a world-space length that is at most as large as a given value. Compare **constant subdivision, screen-space subdivision.**

world-to-frustum transform A transform that defines the relationship between the world coordinate system and the frustum coordinate system. Compare **frustum-to-window transform**, **local-to-world transform**.

wrap For a shader effect, to replicate the entire effect across the mapped area. Compare **clamp.**

writing loop A section of code in which all writing takes place. A writing loop begins with a call to the `Q3View_StartWriting` routine and should end when a call to `Q3View_EndWriting` returns some value other than `kQ3ViewStatusRetraverse`. A writing

loop is a type of submitting loop. See also **bounding loop, picking loop, rendering loop.**

***x* axis** In Cartesian coordinates, the horizontal axis.

X color map data structure A data structure that contains information about an X color map. Defined by the `TQ3XColormapData` data type.

X draw context A draw context that is associated with a window in an X windows display.

X draw context data structure A data structure that contains information about an X draw context. Defined by the `TQ3XDrawContextData` data type.

***y* axis** In Cartesian coordinates, the vertical axis.

yon plane The clipping plane farthest away from the camera.

***z* axis** In Cartesian coordinates, the axis that represents depth.

zoom button A button in the controller strip of a viewer object that, when clicked, puts the cursor into zooming mode. Subsequent dragging up or down in the picture area causes the camera's field of view to increase or decrease. Compare **camera viewpoint control, distance button, move button, rotate button.**

Index

Symbols

Numerals

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Line art was created using Adobe Illustrator™ and Adobe Photoshop™. PostScript™, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Adobe Letter Gothic.

WRITER
George Towner

DEVELOPMENTAL EDITOR
Jeanne Woodward

ILLUSTRATORS
Ruth Anderson and Sandee Karr

ONLINE PRODUCTION
Bill Harris

Special thanks to Kent Davidson, Tracey Davis, Robert Dierkes, Pablo Fernicola, Julian Gómez, Mark Halstead, Mike Kelley, Eiichiro Mikami, Tim Monroe, Brent Pease, Philip Schneider, Klaus Strelau, Nick Thompson, David Vasquez, Dan Venolia, Ingrid Voss, and Kevin Wu.

Acknowledgments to George Corrick, Joe Flesch, Vicky Kaiser, Pete Litwinowicz, Charles Loop, Malcom MacFail, Fábio Pettinati, Brian Rowe, Steve Rubin, Melissa Sleeter, Ken Turkowski, and John Wang.