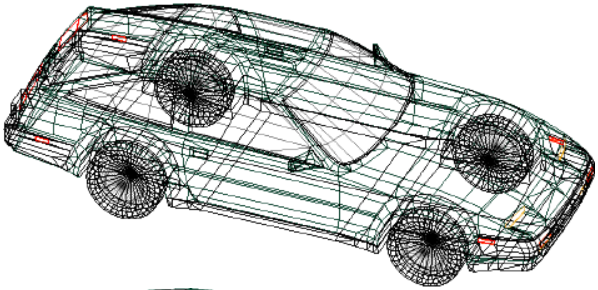
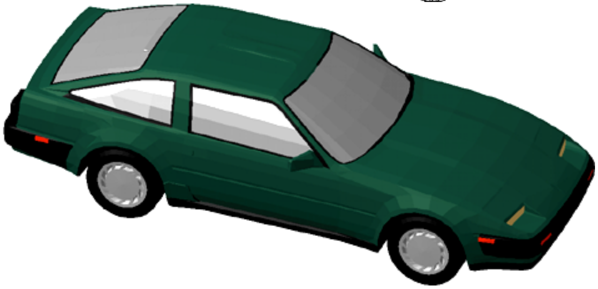
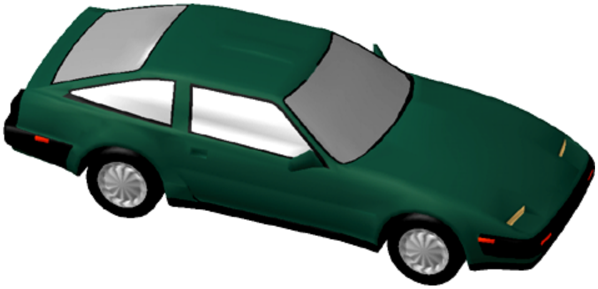
Plate 1 Wireframe and rendered images



Wireframe renderer



Interactive flat
shading renderer



Interactive renderer
per-vertex interpolation

Plate 2 RGB color space

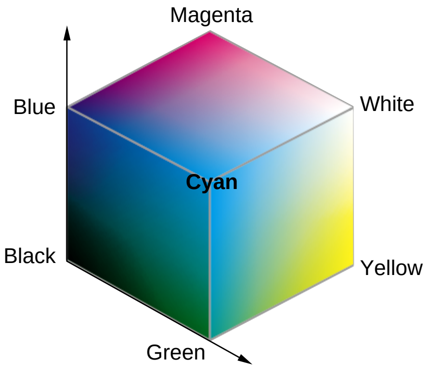
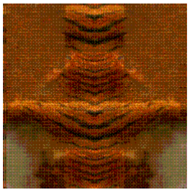
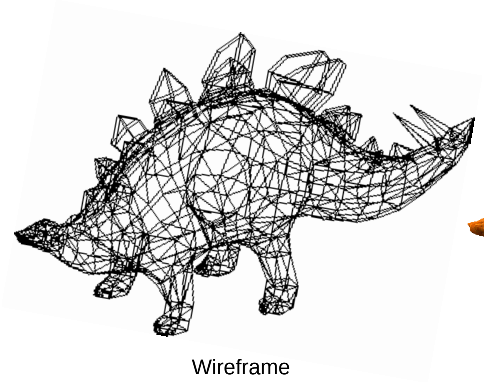


Plate 3 A texture applied to a geometry



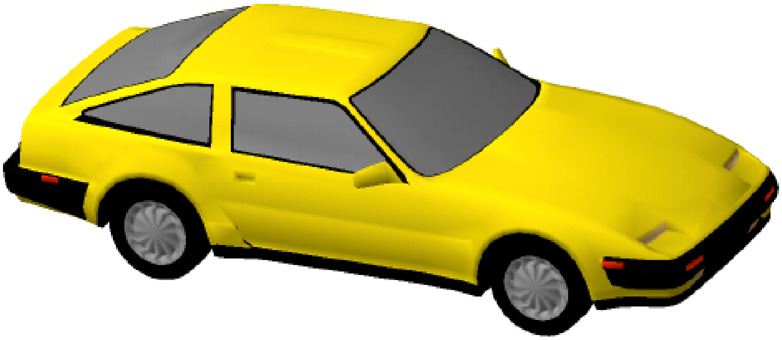
Texture



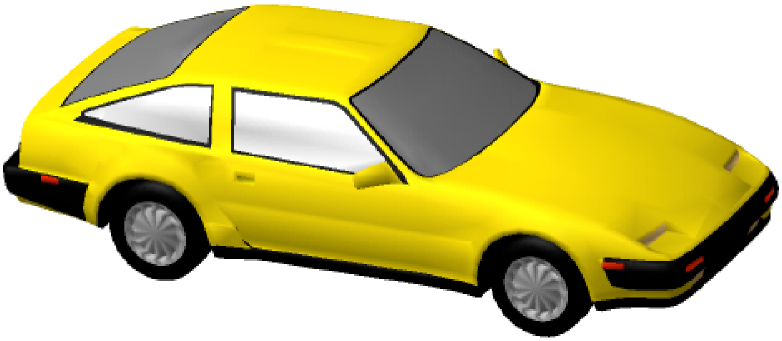
Wireframe

Rendered image

Plate 4 Illumination shaders



Lambert Illumination



Phong Illumination

3D Graphics Programming With QuickDraw 3D

With Reference Sections

Addison-Wesley Publishing Company

Reading, Massachusetts Menlo Park, California New York
Don Mills, Ontario Wokingham, England Amsterdam Bonn
Sydney Singapore Tokyo Madrid San Juan
Paris Seoul Milan Mexico City Taipei

 Apple Computer, Inc.
© 1995 Apple Computer, Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Computer, Inc. Printed in the United States of America.

No licenses, express or implied, are granted with respect to any of the technology described in this book. Apple retains all intellectual property rights associated with the technology described in this book.

Every effort has been made to ensure that the information in this manual is accurate. Apple is not responsible for printing or clerical errors.

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

Apple, the Apple logo, APDA, HyperCard, LaserWriter, Macintosh, Macintosh Quadra, MPW, and PowerBook are trademarks of Apple Computer, Inc., registered in the United States and other countries.

QuickDraw, QuickDraw 3D, and QuickTime are trademarks of Apple Computer, Inc.

Adobe Illustrator and PostScript are trademarks of Adobe Systems Incorporated, which may be registered in certain jurisdictions.

America Online is a registered service mark of America Online, Inc.

CompuServe is a registered service mark of CompuServe, Inc.

Dinosaur and car datasets provided in QuickDraw 3D metafile format courtesy of Viewpoint Data Labs, Intl., Orem, UT.

FrameMaker is a registered trademark of Frame Technology Corporation.

Helvetica and Palatino are registered trademarks of Linotype Company.

Internet is a trademark of Digital Equipment Corporation.

ITC Zapf Dingbats is a registered trademark of International Typeface Corporation.

Silicon Graphics is a registered trademark and OpenGL is a trademark of Silicon Graphics, Inc.

UNIX is a registered trademark of Novell, Inc. in the United States and other countries, licensed exclusively through X/Open Company, Ltd.

X Window System is a trademark of the Massachusetts Institute of Technology.

Simultaneously published in the United States and Canada.

ISBN 0-201-48926-0
1 2 3 4 5 6 7 8 9-MA-9998979695
First Printing, June 1995

Library of Congress Cataloging-in-Publication Data

3D graphics programming with QuickDraw 3D : using QuickDraw 3D / [Apple Computer, Inc.].
p. cm.
Includes index.
ISBN 0-201-48926-0
1. Macintosh (Computer)—Programming. 2. Computer graphics.
3. QuickDraw 3D. I. Apple Computer, Inc.
QA76.8.M3I427 1995
006.6—dc20 95-19363
CIP

LIMITED WARRANTY ON MEDIA AND REPLACEMENT

ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though Apple has reviewed this manual, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Chapter 4 **Geometric Objects** 4-1

Chapter 12 Draw Context Objects 12-1

Chapter 13 View Objects 13-1

Chapter 15 Pick Objects 15-1

Chapter 16 Storage Objects 16-1

Chapter 17 File Objects 17-1

Chapter 19 Error Manager 19-1

Chapter 20 QuickDraw 3D Mathematical Utilities 20-1

Figures, Tables, and Listings

Color Plates

About This Book

This book, *3D Graphics Programming With QuickDraw 3D,* describes QuickDraw 3D, 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 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 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 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 book describes the application programming interfaces that you can use to develop applications and other software using QuickDraw 3D. Although QuickDraw 3D 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 attributes) to those provided by QuickDraw 3D.

To use this book, you should be generally familiar with computer graphics and with 3D modeling and rendering techniques. This book 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 implements the standard techniques for 3D modeling, rendering, and interaction. You can consult the Bibliography near the end of this book 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 is object oriented in the sense that many of its capabilities are accessed by creating and manipulating QuickDraw 3D objects. In addition, QuickDraw 3D classes (of which QuickDraw 3D 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 book by reading the chapter “Introduction to QuickDraw 3D.” That chapter describes the basic capabilities provided by QuickDraw 3D and the QuickDraw 3D 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 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, you can use the 3D Viewer supplied with QuickDraw 3D. 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, you can read the remaining chapters in this book for more information on any particular topic. For example, for complete information on the types of lights provided by QuickDraw 3D, see the chapter “Light Objects.”

Format of a Typical Chapter

Almost all chapters in this book 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 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 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. Note, however, that this section is not included in the printed version of this book; it is available only online, on the enclosed CD-ROM.
- "Summary of Attribute Objects." This section provides the C interfaces for the constants, data structures, routines, and result codes associated with attribute objects.

Conventions Used in This Book

This book 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 Courier (`this is Courier`).

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 book.

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 1-4.) ◆

IMPORTANT
A note like this contains information that is essential for an understanding of the main text. (An example appears on page 1-14.) ▲

▲ **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 16-8.) ▲

Development Environment

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

All code listings in this book are shown in C. They show methods 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

APDA is Apple's worldwide source for over three hundred development tools, technical resources, training products, and information for anyone interested in developing applications on Apple platforms. Customers receive the quarterly *APDA Tools Catalog* featuring all current versions of Apple and the most popular third-party development tools. Ordering is easy; there are no membership fees, and application forms are not required for most of our products. APDA offers convenient payment and shipping options, including site licensing.

To order products or to request a complimentary copy of the *APDA Tools Catalog*, contact

APDA
Apple Computer, Inc.
P.O. Box 319
Buffalo, NY 14207-0319

Telephone	800-282-2732 (United States) 800-637-0029 (Canada) 716-871-6555 (International)
Fax	716-871-6511
AppleLink	APDA
America Online	APDAorder
CompuServe	76666,2405
Internet	APDA@applelink.apple.com

If you provide commercial products and services, call 408-974-4897 for information on the developer support programs available from Apple.

For information of registering signatures, file types, and other technical information, contact

Macintosh Developer Technical Support
Apple Computer, Inc.
1 Infinite Loop, M/S 303-2T
Cupertino, CA 95014

Introduction to QuickDraw 3D

Contents

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.

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 1-14 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 1-34, 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

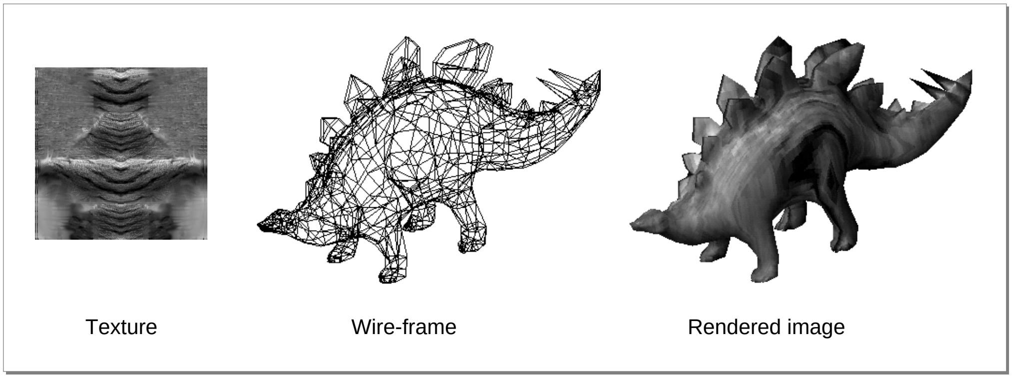
QuickDraw 3D is a graphics library developed by Apple Computer that you can use to create, configure, and render three-dimensional objects. It is specifically designed to be useful to a wide range of software developers, from those with very little knowledge of 3D modeling concepts and rendering techniques 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 some other 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. See also Color Plate 3 at the beginning of this book.

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

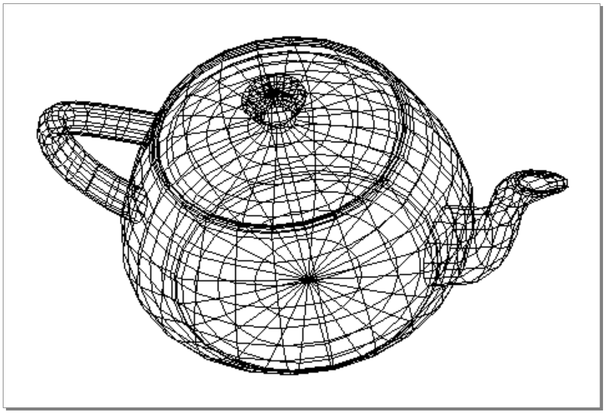
- determining what kinds of *pointing devices* are available on a particular computer and possibly configuring one or more of those devices to control items in a 3D model (such as a camera or a light).
- identifying the objects in a model that are close to the cursor when the user clicks or drags in 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 "QuickDraw 3D Pointing Device Manager" and "Pick Objects."

Extending QuickDraw 3D

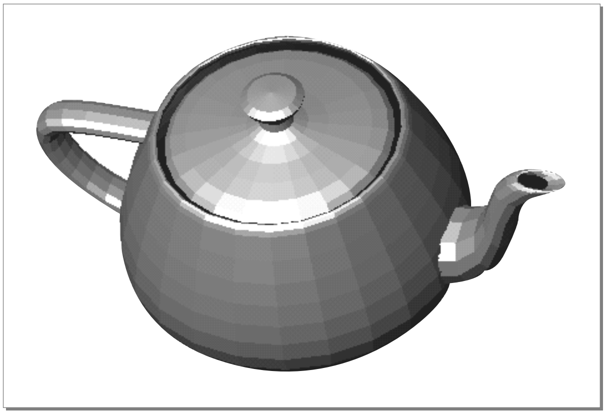
QuickDraw 3D is designed to be easily extensible, so that you can, if necessary, add capabilities that are not part of the basic QuickDraw 3D feature set. For instance, you've already seen that QuickDraw 3D supplies two types of renderers, the wireframe and interactive renderers. The wireframe renderer 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



QuickDraw 3D is extensible:

- You can define *custom attributes* and assign them to shapes or sets.

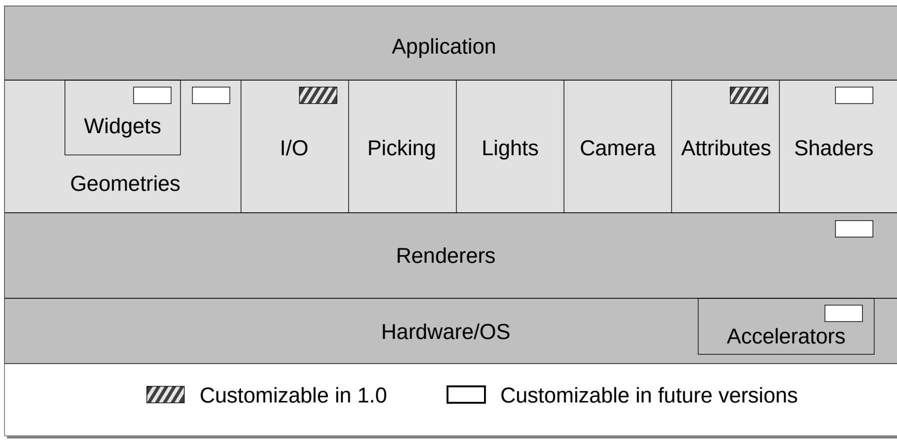
In addition, QuickDraw 3D is designed to be portable to other software platforms and to support a variety of hardware accelerators:

- QuickDraw 3D is *cross-platform.* It is available on the Macintosh Operating System and will be available on other operating systems that use alternative window systems as well. 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.

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 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 page 3-34 for a definition 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. Note, however, that when using immediate mode, you need to know exactly what types of objects you're drawing and hardcode 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 only on PowerPC-based Macintosh computers. It exists as a shared library, in two forms. A debugging version is available for use by developers while writing their applications or other software products. An optimized version of the QuickDraw 3D shared library is available for end users of those applications and other 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 Macintosh Operating System, 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((Boolean) 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 Macintosh Operating System, 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.");
}
```

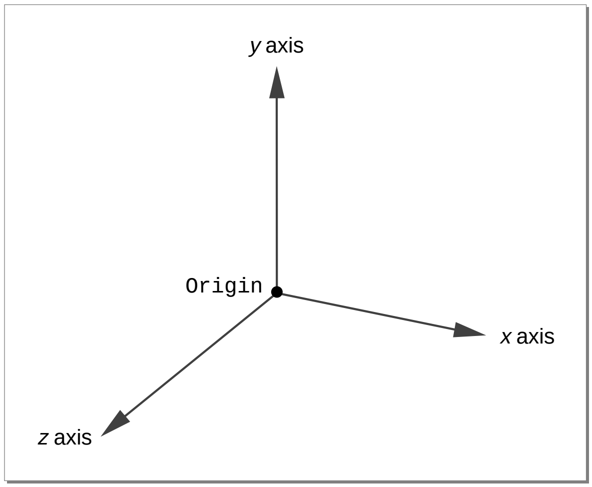
Creating a Model

As you learned earlier (in "Modeling and Rendering" on page 1-4), 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 you'll see later (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 on page 1-20*/
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 1-31 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, as 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);
} 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 1-20), using the view created and configured in Listing 1-9 (page 1-30). 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 CSG.

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 "Summary of QuickDraw 3D," beginning on page 1-52 for a list of the basic data types 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.

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" on page 3-17 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 types of sets currently supported by QuickDraw 3D are defined by constants:

```
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.

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). A string object is an object that contains a sequence of characters.

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.

Summary of QuickDraw 3D

C Summary

Constants

Gestalt Selector and Response Values

```
enum {
    gestaltQD3D                         = 'qd3d',
    gestaltQD3DNotPresent               = 0,
    gestaltQD3DAvailable                = 1
}
```

Basic Constants

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

typedef enum TQ3Status {
    kQ3Failure,
    kQ3Success
} TQ3Status;

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

QuickDraw 3D Routines

Initializing and Terminating QuickDraw 3D

```
TQ3Status Q3Initialize        (void);

TQ3Status Q3Exit              (void);

TQ3Boolean Q3IsInitialized    (void);
```

Getting Version Information

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

Managing Sets

```
TQ3SetObject Q3Set_New        (void);

TQ3ObjectType Q3Set_GetType   (TQ3SetObject set);

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

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

TQ3Boolean Q3Set_Contains     (TQ3SetObject set, TQ3ElementType type);

TQ3Status Q3Set_GetNextElementType (
                               TQ3SetObject set, TQ3ElementType *type);

TQ3Status Q3Set_Empty         (TQ3SetObject target);

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

Managing Shapes

```
TQ3ObjectType Q3Shape_GetType (TQ3ShapeObject shape);

TQ3Status Q3Shape_GetSet      (TQ3ShapeObject shape, TQ3SetObject *set);

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

Managing Strings

```
TQ3ObjectType Q3String_GetType(TQ3StringObject stringObj);

TQ3StringObject Q3CString_New (const char *string);

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

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

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

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

Errors, Warnings, and Notices

```
kQ3ErrorInternalError
kQ3ErrorNoRecovery
kQ3ErrorNotInitialized
kQ3ErrorAlreadyInitialized
kQ3ErrorUnimplemented
kQ3ErrorRegistrationFailed
kQ3ErrorMemoryLeak
kQ3ErrorOutOfMemory
kQ3ErrorNULLParameter
kQ3ErrorParameterOutOfRange
kQ3ErrorInvalidParameter
kQ3ErrorInvalidData
kQ3ErrorAcceleratorAlreadySet
kQ3ErrorInvalidObject
kQ3ErrorInvalidObjectType
```

```
kQ3ErrorInvalidObjectName
kQ3ErrorObjectClassInUse
kQ3ErrorAccessRestricted
kQ3ErrorMetaHandlerRequired
kQ3ErrorNeedRequiredMethods
kQ3ErrorNoSubClassType
kQ3ErrorUnknownElementType
kQ3ErrorNotSupported
kQ3ErrorNoExtensionsFolder
kQ3ErrorExtensionError
kQ3ErrorPrivateExtensionError
kQ3ErrorBadStringType

kQ3WarningInternalException
kQ3WarningNoObjectSupportForDuplicateMethod
kQ3WarningNoObjectSupportForWriteMethod
kQ3WarningNoObjectSupportForReadMethod
kQ3WarningNoObjectSupportForDrawMethod
kQ3WarningUnknownElementType
kQ3WarningTypeAndMethodAlreadyDefined
kQ3WarningTypeIsOutOfRange
kQ3WarningTypeHasNotBeenRegistered
kQ3WarningInvalidSubObjectForObject
kQ3WarningInvalidHexString
kQ3WarningUnknownObject
kQ3WarningInvalidTableOfContents
kQ3WarningUnresolvableReference
kQ3WarningNoAttachMethod
kQ3WarningInconsistentData
kQ3WarningLowMemory
kQ3WarningPossibleMemoryLeak

kQ3NoticeDataAlreadyEmpty
kQ3NoticeMethodNotSupported
kQ3NoticeObjectAlreadySet
```

3D Viewer

Contents

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.

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” earlier in this book.

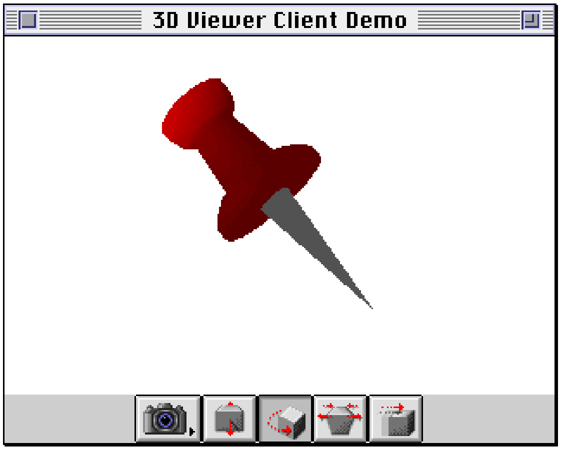
IMPORTANT

The 3D Viewer allows you to display 3D data in metafiles (or in memory) with minimal programming effort. It is analogous to the movie controller provided with QuickTime, which allows you, also with minimal programming effort, to display and allow users to control movies. If your application needs more advanced rendering or interaction capabilities, or if you want to allow users to create and manipulate objects dynamically, you should use the lower-level QuickDraw 3D application programming interfaces instead of the higher-level 3D Viewer programming interfaces. ▲

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 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.

Note
The 3D Viewer is currently available only on the Macintosh Operating System. ◆

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.

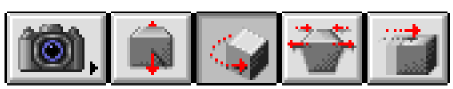
The part of the window that contains the picture area and the controller strip (if present) is the **viewer pane** (or **viewer frame**). In Figure 2-1, the viewer pane entirely fills the window's content region. Alternatively, you can place the viewer pane in part of the window; you would do this to 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 angle button.** This control allows the user to view the model from a different camera angle. Holding down the camera angle button causes a pop-up menu to appear, listing the available cameras.
- 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 **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 **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.

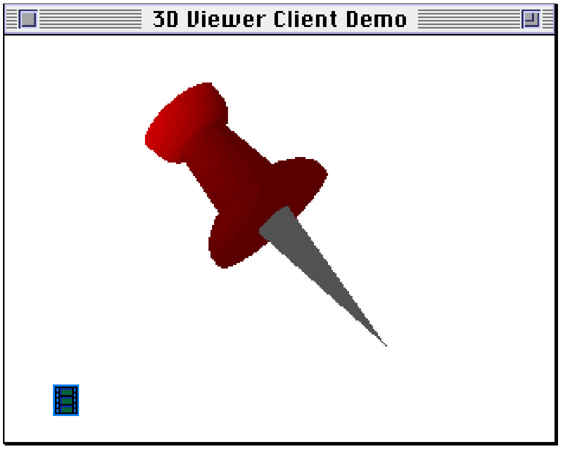
- The **move button.** This control allows the user to move an object. Clicking the move button and then dragging on the object in the picture area causes the object to be moved to a new location.

Your application controls which of these buttons are displayed in a viewer object's controller strip at the time you create the viewer object, by appropriately setting a viewer's flags. See Listing 2-2 on page 2-9 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.

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" on page 2-12 for more information.

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

```
Boolean MyEnvironmentHas3DViewer (void)
{
   return((Boolean)Q3ViewerNew != kUnresolvedSymbolAddress);
}
```

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 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 {
    gestaltQuickDraw3DViewer            = 'q3vc',
    gestaltQ3ViewerNotAvailable         = 0,
    gestaltQ3ViewerAvailable            = 1
}
```

You should ensure that the value `gestaltQ3ViewerAvailable` is returned before calling any 3D Viewer routines.

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

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 2-2 illustrates one way to call `Q3ViewerNew`. The function `MyCreateViewer` defined in Listing 2-2 creates a viewer pane that occupies the entire content region of the window whose address is passed to it as a parameter.

Listing 2-2 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 2-2, 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 2-12 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.

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`.

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 {
    gestaltQuickDraw3DViewer              = 'q3vc'
}
```

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

```
enum {
   gestaltQ3ViewerNotAvailable         = 0,
   gestaltQ3ViewerAvailable            = 1
}
```

Constant descriptions

`gestaltQ3ViewerNotAvailable`
The 3D Viewer is not available.

`gestaltQ3ViewerAvailable`
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.

```
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,
   kQ3ViewerDefault               = (kQ3ViewerViewerActive |
                                     kQ3ViewerControllerVisible |
                                     kQ3ViewerButtonCamera |
                                     kQ3ViewerButtonTruck |
                                     kQ3ViewerButtonOrbit)
};
```

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” on page 2-6 for complete details on when the badge appears and disappears. If this flag is clear, no badge is displayed.

`kQ3ViewerActive`
: If this flag is set, the viewer object is active.

`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 is visible whenever the badge is not displayed.

`kQ3ViewerDrawFrame`
: If this flag is set, a frame is drawn around the viewer pane. If this flag is clear, no frame is drawn around the viewer pane.

`kQ3ViewerDraggingOff`
: If this flag is set, dragging is turned off in the viewer pane.

`kQ3ViewerButtonCamera`
: If this flag is set, the camera angle button in the controller strip is displayed and is active.

`kQ3ViewerButtonTruck`
: If this flag is set, the distance button in the controller strip is displayed and is active.

`kQ3ViewerButtonOrbit`
: If this flag is set, the rotate button in the controller strip is displayed and is active.

`kQ3ViewerButtonZoom`
: If this flag is set, the zoom button in the controller strip is displayed and is active.

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

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

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:**

```
enum {
   kQ3ViewerEmpty                 = 0,
   kQ3ViewerHasModel              = 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.

3D Viewer Routines

This section describes the routines that you can use to create and manage 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

Note

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 2-7. ◆

Creating and Configuring Viewers

This section describes the routines you can use to create and configure new viewer objects. See "Creating a Viewer" on page 2-8 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.

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

`port` A pointer to a color graphics port that specifies the window with which the new viewer is to be associated.

`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.

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" on page 2-12 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.

Q3ViewerDispose

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

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

`theViewer` A viewer object.

DESCRIPTION

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

Q3ViewerUseFile

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

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

`theViewer` A viewer object.

`refnum` The file reference number of 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.

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

`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.

Q3ViewerDraw

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

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

`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 by calling `Q3ViewerDraw`.

Q3ViewerGetView

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

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

`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.

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

`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.

Q3ViewerGetFlags

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

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

`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.

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

`theViewer` A viewer object.

`flags` A set of viewer flags. See "Viewer Flags" on page 2-12 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.

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

`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.

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

`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. ▲

Q3ViewerGetPort

You can use the `Q3ViewerGetPort` function to get the color graphics port associated with a viewer object.

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

`theViewer` A viewer object.

DESCRIPTION

The `Q3ViewerGetPort` function returns, as its function result, a pointer to the color graphics port currently associated with the viewer object specified by the `theViewer` parameter.

Q3ViewerSetPort

You can use the `Q3ViewerSetPort` function to set the color graphics port associated with a viewer object.

```
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.

DESCRIPTION

The `Q3ViewerSetPort` function sets the color 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.

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

`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.

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

`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.

Q3ViewerGetBackgroundColor

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

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

`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.

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

`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.

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

`theViewer` A viewer object.

`refnum` The file reference number of an open file.

DESCRIPTION

The `Q3ViewerWriteFile` function writes the 3D data currently associated with the viewer object specified by the `theViewer` parameter to the file specified by the `refnum` parameter.

Q3ViewerWriteData

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

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

`theViewer` A viewer object.

`data` A pointer to the beginning of a block of data in memory.

DESCRIPTION

The `Q3ViewerWriteData` function writes the 3D data currently associated with the viewer object specified by the `theViewer` parameter to the memory location specified by the `data` parameter.

Handling Viewer Events

Viewer objects support several routines for handling events that occur in a viewer pane.

Q3ViewerEvent

You can use the `Q3ViewerEvent` function to give the 3D Viewer an opportunity to handle events involving a viewer object.

```
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 2-31.

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.

Q3ViewerAdjustCursor

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

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

`theViewer` A viewer object.

`pt` The location 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. `Q3ViewerAdjustCursor` returns a Boolean value that indicates whether the shape of the cursor was changed (`True`) or not (`False`).

Getting Viewer Information

The 3D Viewer provides routines that you can use to get information about a viewer object.

Q3ViewerGetState

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

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

`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
};
```

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. You can use this information to determine whether Edit menu commands such as Cut, Clear, and Copy should be enabled or disabled.

Q3ViewerGetPict

You can use the `Q3ViewerGetPict` function to get a picture representation of the image currently displayed by a viewer object.

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

`theViewer` A viewer object.

DESCRIPTION

The `Q3ViewerGetPict` function returns, as its function result, a handle to a `Picture` 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 picture when you're done using it.

Q3ViewerGetButtonRect

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

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

`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
```

Q3ViewerGetCurrentButton

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

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

`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:

```
kQ3ViewerButtonCamera
kQ3ViewerButtonTruck
kQ3ViewerButtonOrbit
kQ3ViewerButtonZoom
kQ3ViewerButtonDolly
```

Q3ViewerSetCurrentButton

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

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

`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:

```
kQ3ViewerButtonCamera
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.

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

`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.

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.

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

`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.

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

`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.

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

`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 2-27).

Q3ViewerClear

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

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

`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.

Summary of the 3D Viewer

C Summary

Constants

Gestalt Selector and Response Values

```
enum {
    gestaltQuickDraw3DViewer                 = 'q3vc',
    gestaltQ3ViewerNotAvailable              = 0,
    gestaltQ3ViewerAvailable                 = 1
}
```

Viewer Flags

```
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,
    kQ3ViewerDefault                 =
                                     (kQ3ViewerActive |
                                      kQ3ViewerControllerVisible |
```

```
                                    kQ3ViewerButtonCamera |
                                    kQ3ViewerButtonTruck |
                                    kQ3ViewerButtonOrbit)
};
```

Viewer State Flags

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

Data Types

```
typedef void                                        *TQ3ViewerObject;
```

3D Viewer Routines

Creating and Configuring Viewers

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

OSErr Q3ViewerDispose          (TQ3ViewerObject theViewer);

OSErr Q3ViewerUseFile          (TQ3ViewerObject theViewer, long refNum);

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

OSErr Q3ViewerDraw             (TQ3ViewerObject theViewer);

TQ3ViewObject Q3ViewerGetView  (TQ3ViewerObject theViewer);

OSErr Q3ViewerRestoreView      (TQ3ViewerObject theViewer);

unsigned long Q3ViewerGetFlags (TQ3ViewerObject theViewer);
```

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

OSErr Q3ViewerGetBounds       (TQ3ViewerObject theViewer, Rect *bounds);

OSErr Q3ViewerSetBounds       (TQ3ViewerObject theViewer, Rect *bounds);

CGrafPtr Q3ViewerGetPort      (TQ3ViewerObject theViewer);

OSErr Q3ViewerSetPort         (TQ3ViewerObject theViewer, CGrafPtr port);

TQ3GroupObject Q3ViewerGetGroup (
                               TQ3ViewerObject theViewer);

OSErr Q3ViewerUseGroup        (TQ3ViewerObject theViewer,
                               TQ3GroupObject group);

OSErr Q3ViewerGetBackgroundColor (
                               TQ3ViewerObject theViewer,
                               TQ3ColorARGB *color);

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

Updating Viewer Data

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

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

Handling Viewer Events

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

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

Getting Viewer Information

```
unsigned long Q3ViewerGetState(TQ3ViewerObject theViewer);

PicHandle Q3ViewerGetPict     (TQ3ViewerObject theViewer);
```

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

unsigned long Q3ViewerGetCurrentButton (
                               TQ3ViewerObject theViewer);

OSErr Q3ViewerSetCurrentButton(TQ3ViewerObject theViewer,
                               unsigned long button);

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

Handling Edit Commands

```
OSErr Q3ViewerCut             (TQ3ViewerObject theViewer);

OSErr Q3ViewerCopy            (TQ3ViewerObject theViewer);

OSErr Q3ViewerPaste           (TQ3ViewerObject theViewer);

OSErr Q3ViewerClear           (TQ3ViewerObject theViewer);
```

QuickDraw 3D Objects

Contents

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 3-14 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 3-18 describes the most basic routines associated with the QuickDraw 3D class hierarchy. These routines allow you to manage objects and shared 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.

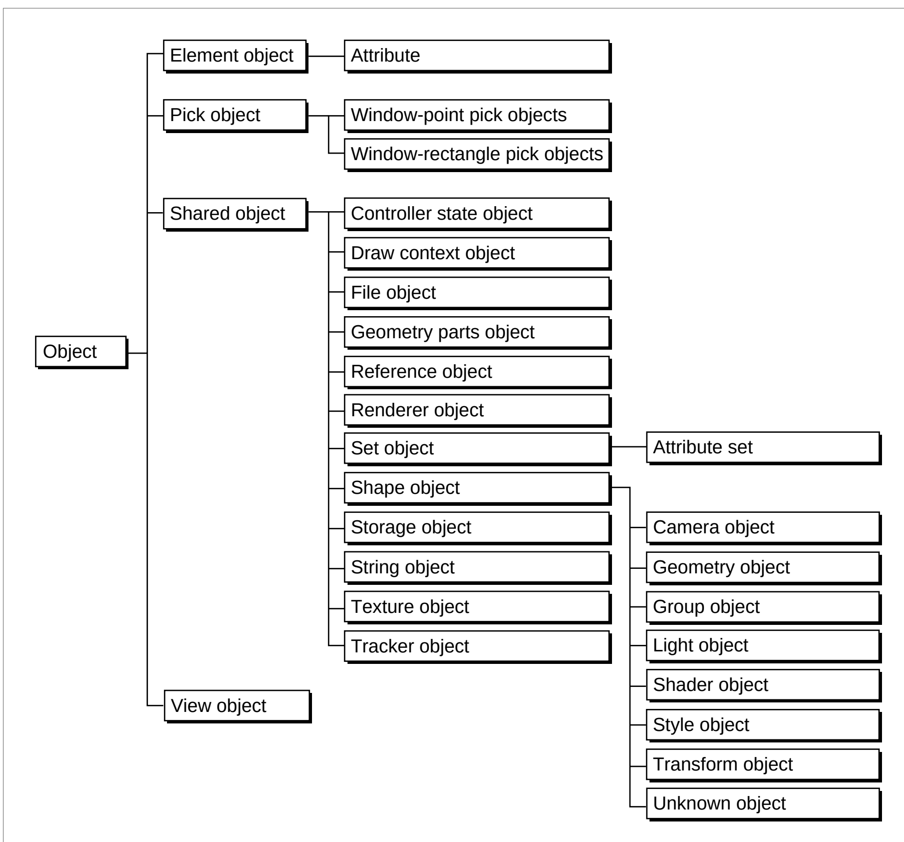
Note
Currently, only C language interfaces are available for creating and manipulating QuickDraw 3D objects. ◆

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. 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.** Figure 3-1 illustrates the top levels of the QuickDraw 3D class hierarchy.

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



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

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" on page 3-11 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" on page 3-14 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" on page 1-31 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" on page 3-11. ◆

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                TQ3ControllerStateObject;
typedef TQ3SharedObject                TQ3DrawContextObject;
typedef TQ3SharedObject                TQ3FileObject;
typedef TQ3SharedObject                TQ3ReferenceObject;
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;
```

Controller state objects and tracker objects are used to support user interaction with the objects in a model. See the chapter "QuickDraw 3D 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 **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.

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" on page 3-9 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                  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.

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 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.

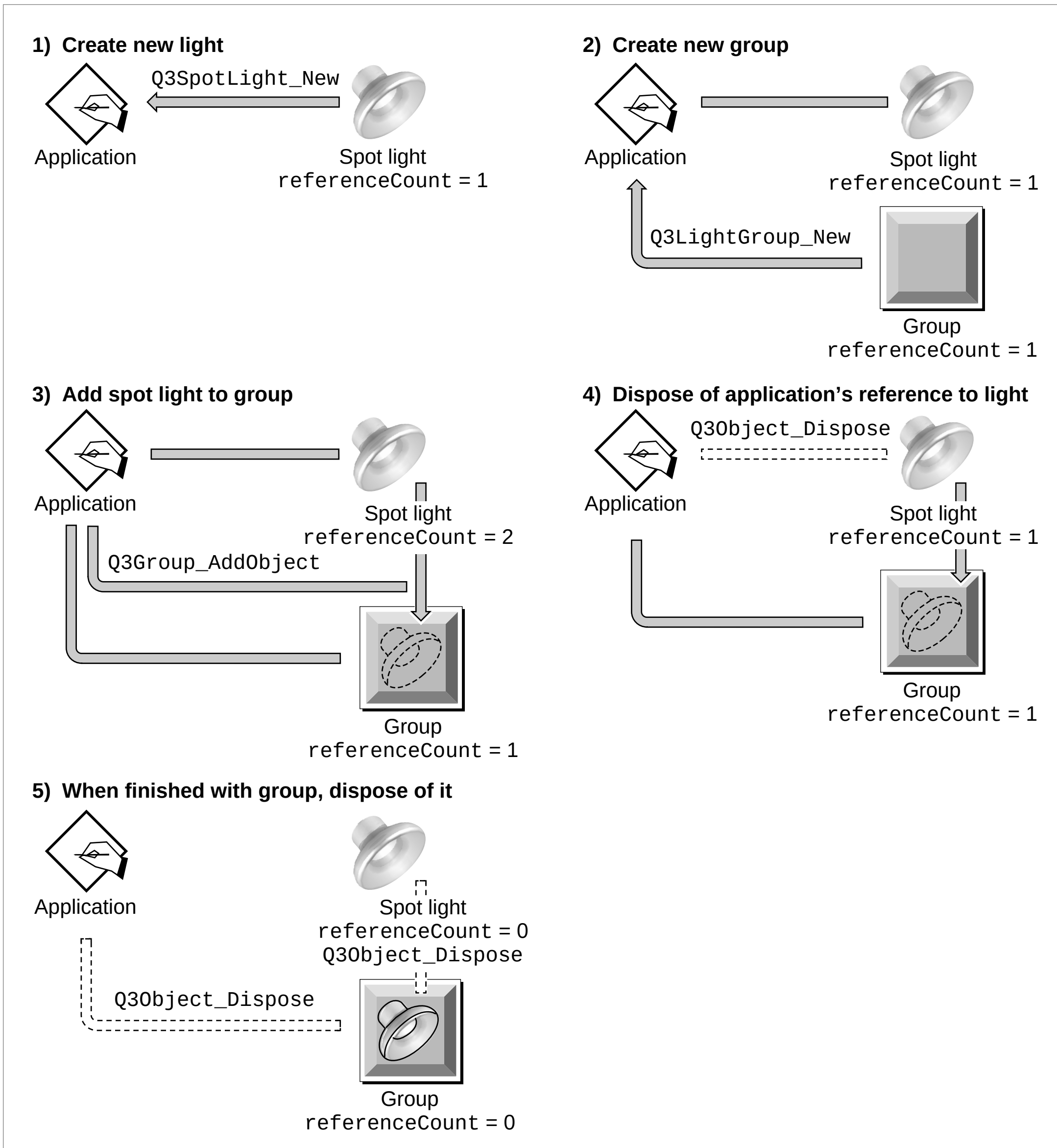
```
typedef TQ3ShapeObject                    TQ3SurfaceShaderObject;
typedef TQ3ShapeObject                    TQ3IlluminationShaderObject;
```

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 earlier (in “QuickDraw 3D Object Subclasses” on page 3-6), 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 on page 3-12 illustrates a series of operations involving a spot light and a group.

Figure 3-2 Incrementing and decrementing reference counts



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`.

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

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 on page 1-25 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 provides source code examples that show how you can

- determine the type of a QuickDraw 3D object
- define a simple object metahandler to support a custom attribute 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.

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.

Your object metahandler is an application-defined function that returns the addresses of the methods associated with the custom object type.

QuickDraw 3D supports a large number of object methods. All custom objects should support this method:

```
kQ3MethodTypeObjectUnregister
```

Note
See "Application-Defined Routines," beginning on page 3-28 for more information on defining custom object methods. ◆

Custom objects that are to be read from and written to files should support these I/O methods:

```
kQ3MethodTypeObjectTraverse
kQ3MethodTypeObjectWrite
kQ3MethodTypeObjectReadData
```

Note
See the chapter "File Objects" for more information on defining custom I/O methods. ◆

Custom attribute types should support these methods:

```
kQ3MethodTypeAttributeCopyInherit
kQ3MethodTypeAttributeInherit
```

Note
See the chapter "Attribute Objects" for more information on defining custom attribute types. ◆

Custom element types should support these methods:

```
kQ3MethodTypeElementCopyAdd
kQ3MethodTypeElementCopyReplace
kQ3MethodTypeElementCopyGet
kQ3MethodTypeElementCopyDuplicate
kQ3MethodTypeElementDelete
```

Note
See "Defining Custom Elements," beginning on page 3-17 for more information on defining custom element types. ◆

Listing 3-1 defines a simple attribute metahandler.

Listing 3-1 Reporting custom object methods

```
TQ3FunctionPointer MyObjectMetaHandler (TQ3MethodType methodType)
{
    switch (methodType) {
        case kQ3MethodTypeObjectUnregister:
            return (TQ3FunctionPointer) MyObject_Unregister;
        default:
            return (NULL);
    }
}
```

As you can see, the `MyObjectMetaHandler` metahandler simply returns the appropriate function address, or `NULL` if the metahandler does not implement a particular method type.

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`.

To define a custom element type, you need to define and register (using your element metahandler) custom element methods. Currently, QuickDraw 3D supports five element methods, corresponding to these constants:

```
kQ3MethodTypeElementCopyAdd
kQ3MethodTypeElementCopyReplace
kQ3MethodTypeElementCopyGet
kQ3MethodTypeElementCopyDuplicate
kQ3MethodTypeElementDelete
```

The four copy methods are called to add a new element of your custom type to a set, to replace an existing element of your custom type, to get the data associated with an element of your custom type, and to duplicate the data associated with an element of your custom type. Note that the data you maintain internally for a custom element type can differ from the data you return to an application when it calls `Q3Set_Get` or `Q3AttributeSet_Get`.

See "Application-Defined Routines," beginning on page 3-28 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 Classes

QuickDraw 3D provides a routine that you can use to unregister custom object classes.

Q3ObjectClass_Unregister

You can use the `Q3ObjectClass_Unregister` function to remove an application-defined object class.

```
TQ3Status Q3ObjectClass_Unregister (TQ3ObjectClass objectClass);
```

`objectClass` An object class.

DESCRIPTION

The `Q3ObjectClass_Unregister` unregisters the custom object class specified by the `objectClass` parameter. For example, you can call `Q3ObjectClass_Unregister` to unregister a custom attribute type you registered by calling the function `Q3AttributeClass_Register`.

You should dispose of all instances of the custom object class you want to unregister before calling `Q3ObjectClass_Unregister`.

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_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.

ERRORS

`kQ3ErrorInvalidObject`

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.

ERRORS

```
kQ3ErrorInvalidObject
kQ3ErrorOutOfMemory
kQ3ErrorUnimplemented
```

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

```
kQ3ErrorInvalidObject
kQ3ErrorOutOfMemory
kQ3ErrorUnimplemented
```

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`).

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).

Determining Object Types

QuickDraw 3D provides routines that you can use to determine the type of a QuickDraw 3D object.

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.

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`.

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(myObject, kQ3ShapeTypeGeometry))
   return MyDoGeometry(object);
```

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.

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:

```
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`.

Registering Custom Elements

You can add a custom element type by calling the `Q3ElementClass_Register` function. If necessary, you get the size of an application-defined element type by calling the `Q3ElementType_GetElementSize` function.

Q3ElementClass_Register

You can use the `Q3ElementClass_Register` function to register an application-defined element class.

```
TQ3ObjectClass Q3ElementClass_Register (
                    TQ3ElementType elementType,
                    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 3-15 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.

SEE ALSO

See page 3-29 for information on writing copy and dispose methods for a custom element type.

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.

Application-Defined Routines

This section describes the methods you can implement to handle a custom object type. Your custom methods are reported to QuickDraw 3D by your metahandler. This section also describes the methods you can implement to handle custom element types. Your custom element methods are also reported to QuickDraw 3D by your metahandler.

Note
For information about defining custom object methods associated with reading and writing file data, see the chapter "File Objects." ◆

TQ3MetaHandler

You can define an object metahandler to specify methods for custom object types or custom element types.

```
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.)

SEE ALSO

See "Defining an Object Metahandler," beginning on page 3-15 for a sample metahandler.

TQ3ObjectUnregisterMethod

You can define a method to unregister your custom object class.

```
typedef TQ3Status (*TQ3ObjectUnregisterMethod) (
                    TQ3ObjectClass objectClass);
```

`objectClass` An object class.

DESCRIPTION

Your `TQ3ObjectUnregisterMethod` function should perform whatever operations are necessary to unregister the object class specified by the `objectClass` parameter. If you have local data associated with that object class, you should define an unregistration method. You must not call the `Q3ObjectClass_Unregister` function within this method.

RESULT CODES

Your `TQ3ObjectUnregisterMethod` function should return `kQ3Success` if it is successful and `kQ3Failure` otherwise.

TQ3ElementCopyAddMethod

You can define a method to copy the data of your custom element type when an element of that type is added element to a set.

```
typedef TQ3Status (*TQ3ElementCopyAddMethod) (
                    const void *fromAPIElement,
                    void *toInternalElement);
```

`fromAPIElement`
A pointer to the element data associated with an element having your custom element type.

`toInternalElement`
On entry, a pointer to an uninitialized block of memory large enough to contain the element data associated with an element having your custom element type.

DESCRIPTION

Your `TQ3ElementCopyAddMethod` function should copy the element data pointed to by the `fromAPIElement` parameter into the location pointed to by the `toInternalElement` parameter. This method is called whenever the `Q3Set_Add` or `Q3AttributeSet_Add` function is used to add an element of your custom type to a set. The `fromAPIElement` parameter contains the same data pointer that was passed to `Q3Set_Add` or `Q3AttributeSet_Add`.

RESULT CODES

Your `TQ3ElementCopyAddMethod` function should return `kQ3Success` if it is successful and `kQ3Failure` otherwise.

TQ3ElementCopyDuplicateMethod

You can define a method to copy the data of your custom element type when an element of that type is in a set being duplicated.

```
typedef TQ3Status (*TQ3ElementCopyDuplicateMethod) (
                    const void *fromInternalElement,
                    void *toInternalElement);
```

`fromInternalElement`
A pointer to the element data associated with an element having your custom element type.

`toInternalElement`
On entry, a pointer to an empty, zeroed block of memory large enough to contain the element data associated with an element having your custom element type.

DESCRIPTION

Your `TQ3ElementCopyDuplicateMethod` function should copy the element data pointed to by the `fromInternalElement` parameter into the location pointed to by the `toInternalElement` parameter. This method is called whenever the `Q3Object_Duplicate` function is used to duplicate a set or an attribute set that contains an element of your custom type.

RESULT CODES

Your `TQ3ElementCopyDuplicateMethod` function should return `kQ3Success` if it is successful and `kQ3Failure` otherwise.

TQ3ElementCopyGetMethod

You can define a method to copy the data of your custom element types when that data is being retrieved from a set.

```
typedef TQ3Status (*TQ3ElementCopyGetMethod) (
                  const void *fromInternalElement,
                  void *toAPIElement);
```

`fromInternalElement`
A pointer to the element data associated with an element having your custom element type.

`toAPIElement`
On entry, a pointer to an empty, zeroed block of memory large enough to contain the element data associated with an element having your custom element type.

DESCRIPTION

Your TQ3ElementCopyGetMethod function should copy the element data pointed to by the fromInternalElement parameter into the location pointed to by the toAPIElement parameter. This method is called whenever the Q3Set_Get or Q3AttributeSet_Get function is used to get the data of an element of your custom type in a set. The toAPIElement parameter contains the same data pointer that was passed to Q3Set_Get or Q3AttributeSet_Get.

RESULT CODES

Your TQ3ElementCopyGetMethod function should return kQ3Success if it is successful and kQ3Failure otherwise.

TQ3ElementCopyReplaceMethod

You can define a method to copy the data of your custom element type when an element of that type is being replaced by another element of that type.

```
typedef TQ3Status (*TQ3ElementCopyReplaceMethod) (
                    const void *fromAPIElement,
                    void *ontoInternalElement);
```

fromAPIElement
: A pointer to the element data associated with an element having your custom element type.

ontoInternalElement
: On entry, a pointer to an empty, zeroed block of memory large enough to contain the element data associated with an element having your custom element type.

DESCRIPTION

Your TQ3ElementCopyReplaceMethod function should copy the element data pointed to by the fromAPIElement parameter into the location pointed to by the toInternalElement parameter. This method is called whenever the Q3Set_Add or Q3AttributeSet_Add function is used to replace an element of your custom type in a set. The fromAPIElement parameter contains the same data pointer that was passed to `Q3Set_Add` or `Q3AttributeSet_Add`. The `ontoInternalElement` parameter is a pre-existing block initialized by your `TQ3ElementCopyAddMethod` or `TQ3ElementCopyDuplicateMethod` method.

RESULT CODES

Your `TQ3ElementCopyReplaceMethod` function should return `kQ3Success` if it is successful and `kQ3Failure` otherwise.

TQ3ElementDeleteMethod

You can define a method to delete (that is, dispose of) your custom element types.

```
typedef TQ3Status (*TQ3ElementDeleteMethod) (
                    void *internalElement);
```

`internalElement`
: A pointer to the element data associated with an element having your custom element type.

DESCRIPTION

Your `TQ3ElementDeleteMethod` function should perform whatever operations are necessary to dispose of the element data specified by the `internalElement` parameter.

RESULT CODES

Your `TQ3ElementDeleteMethod` function should return `kQ3Success` if it is successful and `kQ3Failure` otherwise.

Summary of QuickDraw 3D Objects

C Summary

Constants

System-Wide Macros

```
#define Q3_FOUR_CHARACTER_CONSTANT(a,b,c,d)       ((const unsigned long)   \
                              ((const unsigned long) (a) << 24)          \
                            | ((const unsigned long) (b) << 16)          \
                            | ((const unsigned long) (c) << 8)           \
                            | ((const unsigned long) (d)))

#define Q3_OBJECT_TYPE(a,b,c,d)                                          \
                     ((TQ3ObjectType) Q3_FOUR_CHARACTER_CONSTANT(a,b,c,d))

#define Q3_METHOD_TYPE(a,b,c,d)                                          \
                     ((TQ3MethodType) Q3_FOUR_CHARACTER_CONSTANT(a,b,c,d))
```

Core Object Types

```
#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
```

Shared Types

```
#define kQ3SharedTypeControllerState         Q3_OBJECT_TYPE('c','t','s','t')
#define kQ3SharedTypeDrawContext             Q3_OBJECT_TYPE('d','c','t','x')
#define kQ3SharedTypeFile                    Q3_OBJECT_TYPE('f','i','l','e')
#define kQ3SharedTypeReference               Q3_OBJECT_TYPE('r','f','r','n')
#define kQ3SharedTypeRenderer                Q3_OBJECT_TYPE('r','d','d','r')
#define kQ3SharedTypeSet                     Q3_OBJECT_TYPE('s','e','t',' ')
#define kQ3SharedTypeShape                   Q3_OBJECT_TYPE('s','h','a','p')
#define kQ3SharedTypeShapePart               Q3_OBJECT_TYPE('s','p','r','t')
#define kQ3SharedTypeStorage                 Q3_OBJECT_TYPE('s','t','r','g')
#define kQ3SharedTypeString                  Q3_OBJECT_TYPE('s','t','r','n')
#define kQ3SharedTypeTexture                 Q3_OBJECT_TYPE('t','x','t','r')
#define kQ3SharedTypeTracker                 Q3_OBJECT_TYPE('t','r','k','r')
#define kQ3SharedTypeViewHints               Q3_OBJECT_TYPE('v','w','h','n')
```

Shape Types

```
#define kQ3ShapeTypeCamera                   Q3_OBJECT_TYPE('c','m','r','a')
#define kQ3ShapeTypeGeometry                 Q3_OBJECT_TYPE('g','m','t','r')
#define kQ3ShapeTypeGroup                    Q3_OBJECT_TYPE('g','r','u','p')
#define kQ3ShapeTypeLight                    Q3_OBJECT_TYPE('l','g','h','t')
#define kQ3ShapeTypeShader                   Q3_OBJECT_TYPE('s','h','d','r')
#define kQ3ShapeTypeStyle                    Q3_OBJECT_TYPE('s','t','y','l')
#define kQ3ShapeTypeTransform                Q3_OBJECT_TYPE('x','f','r','m')
#define kQ3ShapeTypeUnknown                  Q3_OBJECT_TYPE('u','n','k','n')
```

Element Types

```
#define kQ3ElementTypeAttribute              Q3_OBJECT_TYPE('e','a','t','t')
#define kQ3ElementTypeNone                   0
#define kQ3ElementTypeUnknown                32
```

Set Types

```
#define kQ3SetTypeAttribute                  Q3_OBJECT_TYPE('a','t','t','r')
```

String Types

```
#define kQ3StringTypeCString                    Q3_OBJECT_TYPE('s','t','r','c')
```

Method Types

```
#define kQ3MethodTypeObjectFileVersion          Q3_METHOD_TYPE('v','e','r','s')
#define kQ3MethodTypeObjectReadData             Q3_METHOD_TYPE('r','d','d','t')
#define kQ3MethodTypeObjectTraverse             Q3_METHOD_TYPE('t','r','v','s')
#define kQ3MethodTypeObjectUnregister           Q3_METHOD_TYPE('u','n','r','g')
#define kQ3MethodTypeObjectWrite                Q3_METHOD_TYPE('w','r','i','t')

#define kQ3MethodTypeElementCopyAdd             Q3_METHOD_TYPE('e','c','p','a')
#define kQ3MethodTypeElementCopyDuplicate       Q3_METHOD_TYPE('e','c','p','d')
#define kQ3MethodTypeElementCopyGet             Q3_METHOD_TYPE('e','c','p','g')
#define kQ3MethodTypeElementCopyReplace         Q3_METHOD_TYPE('e','c','p','r')
#define kQ3MethodTypeElementDelete              Q3_METHOD_TYPE('e','d','e','l')
```

Data Types

Objects

```
typedef long                                    TQ3ObjectType;

typedef struct TQ3ObjectPrivate                 *TQ3Object;

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

Shared Objects

```
typedef TQ3SharedObject                         TQ3ControllerStateObject;
typedef TQ3SharedObject                         TQ3DrawContextObject;
typedef TQ3SharedObject                         TQ3FileObject;
typedef TQ3SharedObject                         TQ3ReferenceObject;
```

```
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;
```

Sets

```
typedef TQ3SetObject                                       TQ3AttributeSet;

typedef long                                               TQ3ElementType;
```

Shapes

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

Groups

```
typedef TQ3GroupObject                                     TQ3DisplayGroupObject;
```

Shaders

```
typedef TQ3ShaderObject                                    TQ3SurfaceShaderObject;
typedef TQ3ShaderObject                                    TQ3IlluminationShaderObject;
```

Other Basic Types

```
typedef struct TQ3GroupPositionPrivate          *TQ3GroupPosition;

typedef struct TQ3ObjectClassPrivate            *TQ3ObjectClass;

typedef unsigned long                           TQ3MethodType;

typedef void                                    (*TQ3FunctionPointer)(void);
```

QuickDraw 3D Objects Routines

Managing Objects Classes

```
TQ3Status Q3ObjectClass_Unregister (
                                TQ3ObjectClass objectClass);
```

Managing Objects

```
TQ3Status Q3Object_Dispose    (TQ3Object object);

TQ3Object Q3Object_Duplicate  (TQ3Object object);

TQ3Object Q3Object_Submit     (TQ3Object object, TQ3ViewObject view);

TQ3Boolean Q3Object_IsDrawable(TQ3Object object);

TQ3Boolean Q3Object_IsWritable(TQ3Object object);
```

Determining Object Types

```
TQ3ObjectType Q3Object_GetLeafType (
                                TQ3Object object);

TQ3ObjectType Q3Object_GetType(TQ3Object object);

TQ3Boolean Q3Object_IsType    (TQ3Object object, TQ3ObjectType type);
```

Managing Shared Objects

```
TQ3SharedObject Q3Shared_GetReference (
                              TQ3SharedObject sharedObject);

TQ3ObjectType Q3Shared_GetType (
                              TQ3SharedObject sharedObject);
```

Registering Custom Elements

```
TQ3ObjectClass Q3ElementClass_Register (
                              TQ3ElementType elementType,
                              char *name,
                              unsigned long sizeOfElement,
                              TQ3MetaHandler metaHandler);

TQ3Status Q3ElementType_GetElementSize (
                              TQ3ElementType elementType,
                              unsigned long *sizeOfElement);
```

Application-Defined Routines

Method Metahandler

```
typedef TQ3FunctionPointer (*TQ3MetaHandler) (
                              TQ3MethodType methodType);
```

Object Methods

```
typedef TQ3Status (*TQ3ObjectUnregisterMethod) (
                              TQ3ObjectClass objectClass);
```

Set Methods

```
typedef TQ3Status (*TQ3ElementCopyAddMethod) (
                              const void *fromAPIElement,
                              void *toInternalElement);
```

```
typedef TQ3Status (*TQ3ElementCopyDuplicateMethod) (
                              const void *fromInternalElement,
                              void *toInternalElement);

typedef TQ3Status (*TQ3ElementCopyGetMethod) (
                              const void *fromInternalElement,
                              void *toAPIElement);

typedef TQ3Status (*TQ3ElementCopyReplaceMethod) (
                              const void *fromAPIElement,
                              void *ontoInternalElement);

typedef TQ3Status (*TQ3ElementDeleteMethod) (
                              void *internalElement);
```

Geometric Objects

Contents

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 4-23 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 "QuickDraw 3D 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. As you've seen, 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
kQ3GeometryTypeGeneralPolygon
kQ3GeometryTypeLine
kQ3GeometryTypeMarker
kQ3GeometryTypeMesh
kQ3GeometryTypeNURBCurve
kQ3GeometryTypeNURBPatch
kQ3GeometryTypePoint
kQ3GeometryTypePolygon
kQ3GeometryTypePolyLine
kQ3GeometryTypeTriangle
kQ3GeometryTypeTriGrid
```

These objects are described in detail later in this chapter, beginning on page 4-23. 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 three of these types of geometric objects—meshes, NURB curves, and NURB patches—need special discussion. See "Meshes," beginning on page 4-6 for a description of meshes and "NURB Curves and Patches," beginning on page 4-10 for a description of NURB curves and patches.

Note
You can determine a geometric object's type by calling the `Q3Geometry_GetType` function, described later in this chapter. ◆

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.

Note
See the chapter "Attribute Objects" for complete information on the types of attributes defined by QuickDraw 3D and on defining custom attribute types. That chapter also shows how to create attribute sets. ◆

You can associate a set of attributes with most parts of a geometric object. As you've seen, you can associate a set of attributes with the face of a triangle. You can also associate a set of attributes with one or more of the triangle's vertices. Similarly, a box can have a set of attributes that apply to 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;
```

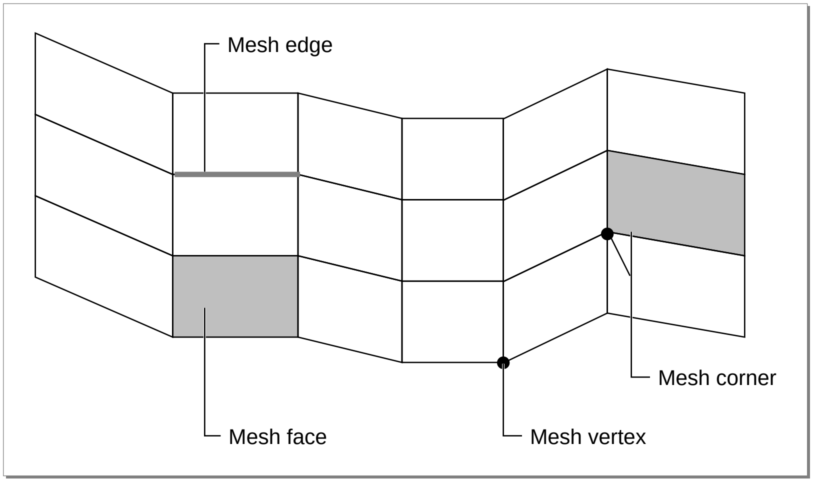
```
    TQ3AttributeSet                     *faceAttributeSet;
    TQ3AttributeSet                     boxAttributeSet;
} TQ3BoxData;
```

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.

Meshes

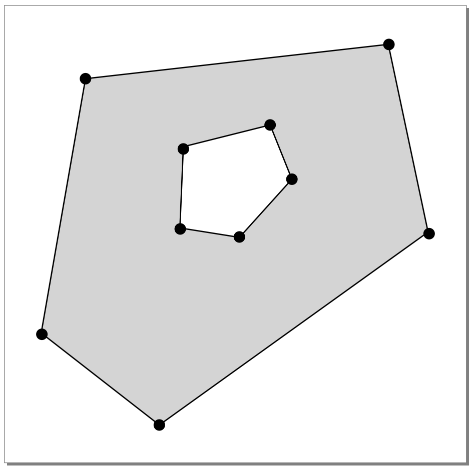
A **mesh** is a collection of vertices, faces, and edges that represent a topological polyhedron (that is, a solid figure composed of polygonal faces). The polyhedra represented by QuickDraw 3D meshes do not need to be closed, so that the meshes may have boundaries. Figure 4-1 illustrates a mesh.

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. You create a basic geometric object by filling in a 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 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 4-140, 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 4-19 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.

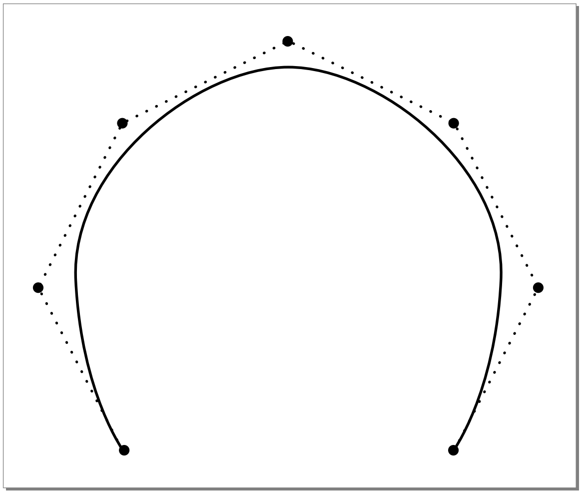
Note also that a vertex, edge, or face might be shifted from one component to another during a change in the topology of the mesh. To be safe, you should bracket all changes to the mesh topology by calls to `Q3Mesh_DelayUpdates` and `Q3Mesh_ResumeUpdates`, and you should not assume that mesh component 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. ◆

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.** 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. Figure 4-3 shows a sample NURB curve.

Figure 4-3 A NURB curve



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

$$M_{j,1}(v) = \begin{cases} 1 & if\, y_j \le v < y_{j+1} \\ 0 & otherwise \end{cases}$$

$$M_{j,k}(v) = \frac{(v - y_j)\, M_{j,\,l-1}(v)}{y_{j+l-1} - y_j} + \frac{(y_{j+l} - v)\, M_{j+1,\,l-1}(v)}{y_{j+l} - y_{j+1}}$$

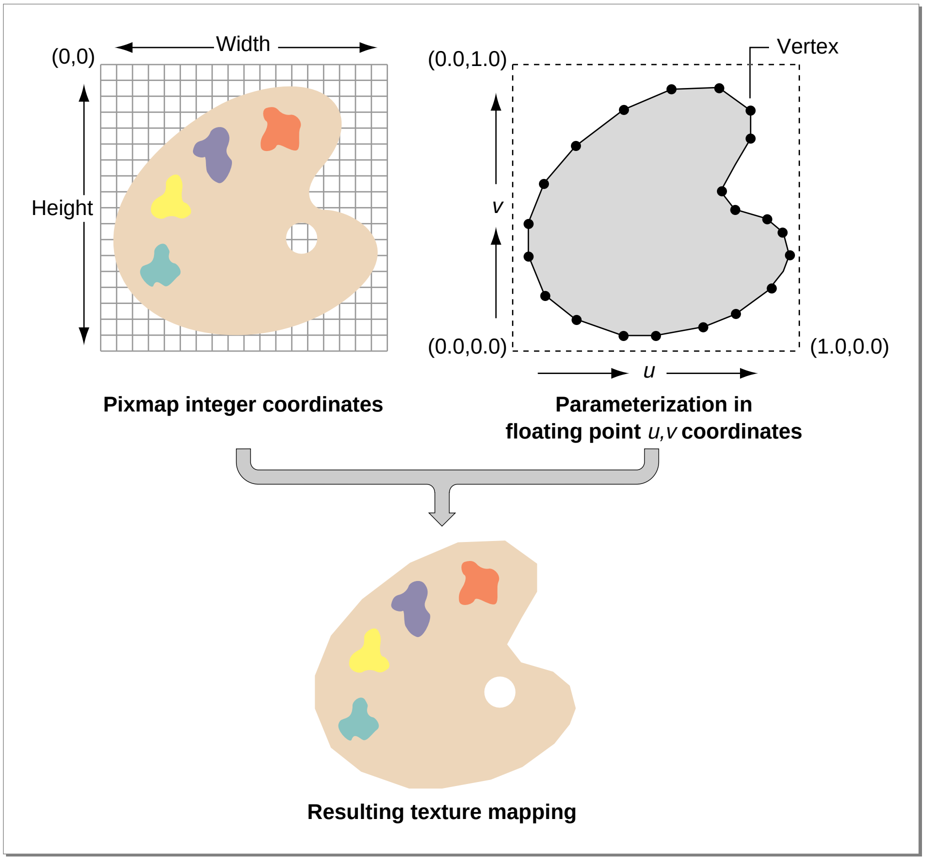
In these equations, the factors $B_{i,j}$ are, algebraically, the coefficients of the polynomial representing the surface. Geometrically, they are the (x, y, z) coordinates of the control points that define the surface. The factors $w_{i,j}$ are the weights of those control points. The factors x_i and 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 on page 4-14 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.

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

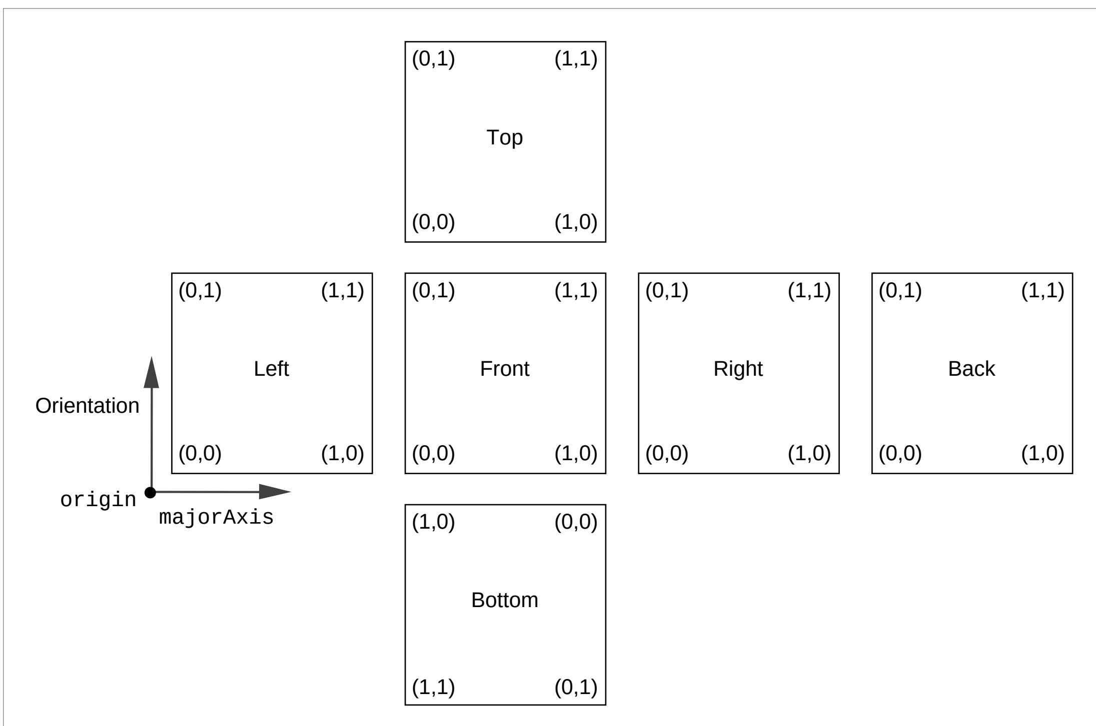
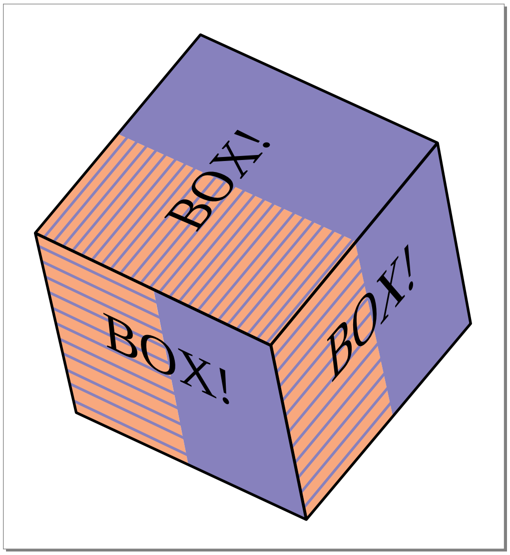


Figure 4-6 shows the result of mapping the texture shown in Figure 4-4 onto the front face of a box.

Figure 4-6 A texture mapped onto a box



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 4-3 on page 4-19 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);
```

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);
```

Because you didn't create any retained entity, you do not need to dispose of the immediate box.

Creating a Mesh

As you saw earlier (in "Meshes," beginning on page 4-6), 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 4-3 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 4-3 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 a large number of functions that you can use to traverse a mesh by iterating through the various distinguishable parts of the mesh (that is, through the faces, vertices, edges, contours, or components in the mesh). 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 by calling `Q3Mesh_NextMeshFace` to get each successive face in the mesh. 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 a mesh. You must pass that same mesh iterator structure to `Q3Mesh_NextMeshFace` when you get successive faces in the mesh. Listing 4-4 illustrates how to use these routines to operate on all faces in a mesh.

Listing 4-4 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. For example, you can use the `Q3ForEachMeshFace` macro, defined like this:

```
#define Q3ForEachMeshFace(m,f,i)                                  \
   for ( (f) = Q3Mesh_FirstMeshFace((m),(i));                     \
      (f);                                                        \
      (f) = Q3Mesh_NextMeshFace((i)) )
```

Listing 4-5 shows how to use two of these macros to attach a corner to each vertex or each face of a mesh.

Listing 4-5 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);
}
```

Geometric Objects Reference

This section describes the data structures provided by QuickDraw 3D that define the QuickDraw 3D geometric objects. It also describes the routines you can use to create and manipulate those objects.

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
- NURB curves
- NURB patches
- 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.

```
typedef struct TQ3RationalPoint3D {
    float        x;
    float        y;
    float        w;
} TQ3RationalPoint3D;

typedef struct TQ3RationalPoint4D {
    float        x;
    float        y;
    float        z;
    float        w;
} TQ3RationalPoint4D;
```

Field descriptions

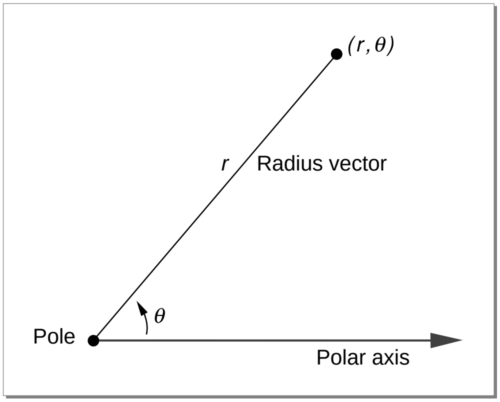
`x`	The *x* coordinate (abscissa) of a rational point.
`y`	The *y* coordinate (ordinate) of a rational point.
`z`	The *z* coordinate of a rational point.
`w`	The weight of a rational point.

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 4-7, 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 4-7 A planar point described with polar coordinates




```
typedef struct TQ3PolarPoint {
   float        r;
   float        theta;
} TQ3PolarPoint;
```

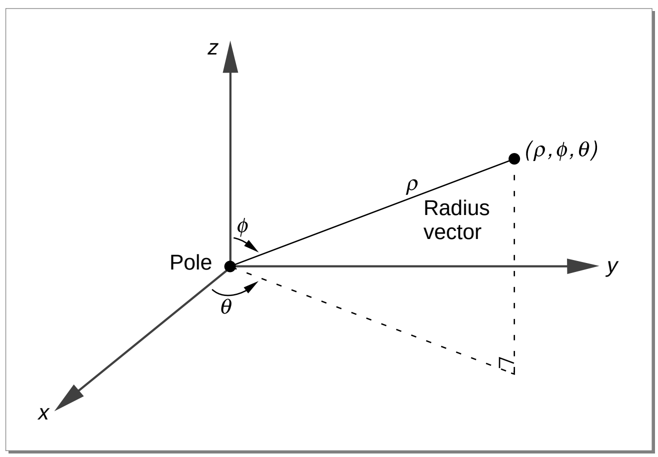
Field descriptions

`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 4-8, 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 4-8 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 and the articles cited in that article. ◆

```
typedef struct TQ3Quaternion {
   float        w;
   float        x;
   float        y;
   float        z;
} TQ3Quaternion;
```

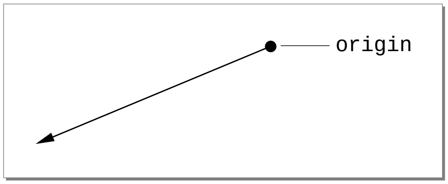
Field descriptions

`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 4-9 shows a ray.

Figure 4-9 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. This field must contain one of these constants:

```
typedef enum TQ3Endian {
   kQ3EndianBig,
   kQ3EndianLittle
} TQ3Endian;
```

The constant `kQ3EndianBig` indicates that the bits are addressed in a big-endian manner. The constant `kQ3EndianLittle` indicates that the bits are addressed in a little-endian manner.

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. This field must contain one of these constants (which must match the size specified in the `pixelSize` field):

```
typedef enum TQ3PixelType {
   kQ3PixelTypeRGB32      /*32 bits per pixel*/
} TQ3PixelType;
```

`bitOrder` 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;
```

The constant `kQ3EndianBig` indicates that the bits are addressed in a big-endian manner. The constant `kQ3EndianLittle` indicates that the bits are addressed in a little-endian manner.

`byteOrder` The order in which the bytes in a word are addressed. This field must contain `kQ3EndianBig` or `kQ3EndianLittle`.

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. This field must contain one of these constants (which must match the size specified in the `pixelSize` field):

```
typedef enum TQ3PixelType {
   kQ3PixelTypeRGB32      /*32 bits per pixel*/
} TQ3PixelType;
```

`bitOrder` 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;
```

The constant `kQ3EndianBig` indicates that the bits are addressed in a big-endian manner. The constant `kQ3EndianLittle` indicates that the bits are addressed in a little-endian manner.

`byteOrder` The order in which the bytes in a word are addressed. This field must contain `kQ3EndianBig` or `kQ3EndianLittle`.

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 4-59 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

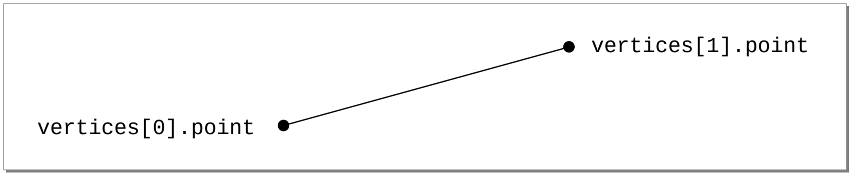
`point` A three-dimensional point.

`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 4-63 for a description of the routines you can use to create and edit lines. Figure 4-10 on page 4-38 shows a line.

Figure 4-10 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.

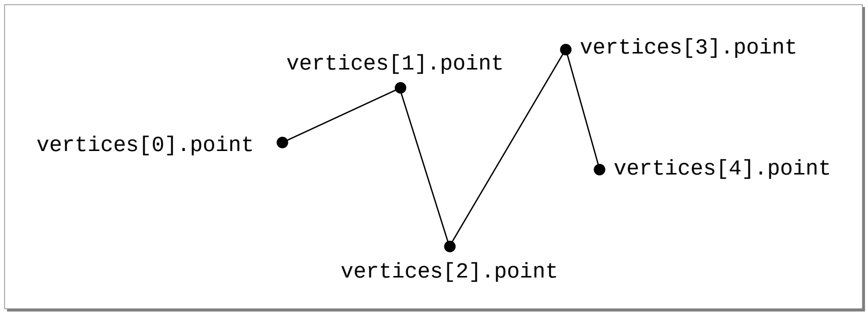
Polylines

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 4-68 for a description of the routines you can use to create and edit polylines. Figure 4-11 on page 4-39 shows a polyline.

IMPORTANT

A polyline is not closed. The last point should not be connected to the first. ▲

Figure 4-11 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.

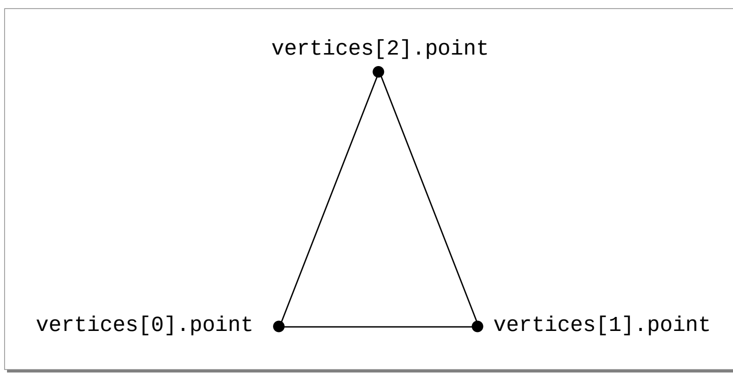
`segmentAttributeSet`
A pointer to an array of segment attribute sets. If no segments in the polyline are to have attributes, this field should contain the value `NULL`. If any of the segments have attributes, this field should contain a pointer to an array (containing `numVertices` – 1 elements) of 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 4-76 for a description of the routines you can use to create and edit triangles. Figure 4-12 shows a triangle.

Figure 4-12 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.

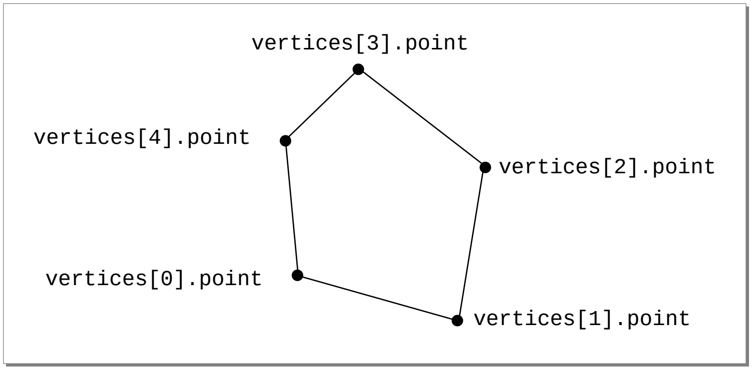
Simple Polygons

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 4-81 for a description of the routines you can use to create and edit simple polygons. Figure 4-13 shows a simple polygon.

Figure 4-13 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

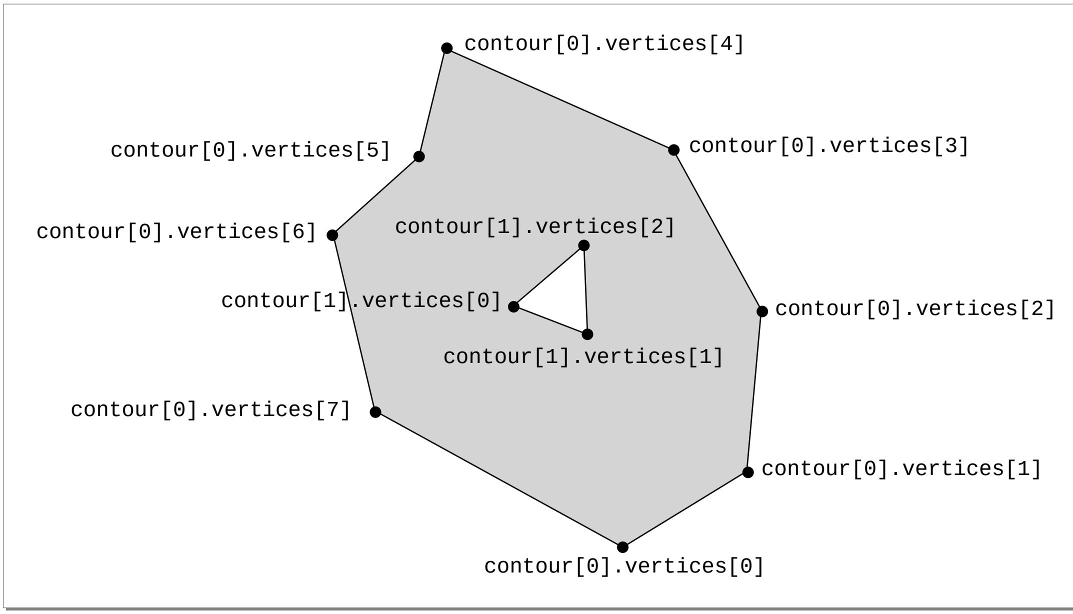
A general polygon is a closed plane figure defined by one or more lists of vertices. (In other words, a general polygon is a polygon defined by one or more contours.) Each contour may be concave or convex, and contours may be nested. In addition, a general polygon's contours may overlap or be disjoint. 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.

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 4-87 for a description of the routines you can use to create and edit general polygons. Figure 4-14 on page 4-43 shows a general polygon.

Figure 4-14 A general polygon




```
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. You can use the following constants for shape hints:

```
typedef enum TQ3GeneralPolygonShapeHint {
    kQ3GeneralPolygonShapeHintComplex,
    kQ3GeneralPolygonShapeHintConcave,
    kQ3GeneralPolygonShapeHintConvex
} TQ3GeneralPolygonShapeHint;
```

The constant `kQ3GeneralPolygonShapeHintComplex` indicates that the general polygon consists of more than one contour, is self-intersecting, or is not known to be either concave or convex. For a general polygon with exactly one contour, the constant `kQ3GeneralPolygonShapeHintConcave` indicates that the polygon is concave, and the constant `kQ3GeneralPolygonShapeHintConvex` indicates that the polygon is convex.

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 4-95 for a description of the routines you can use to create and edit boxes. Figure 4-15 shows a box.

Figure 4-15 A box

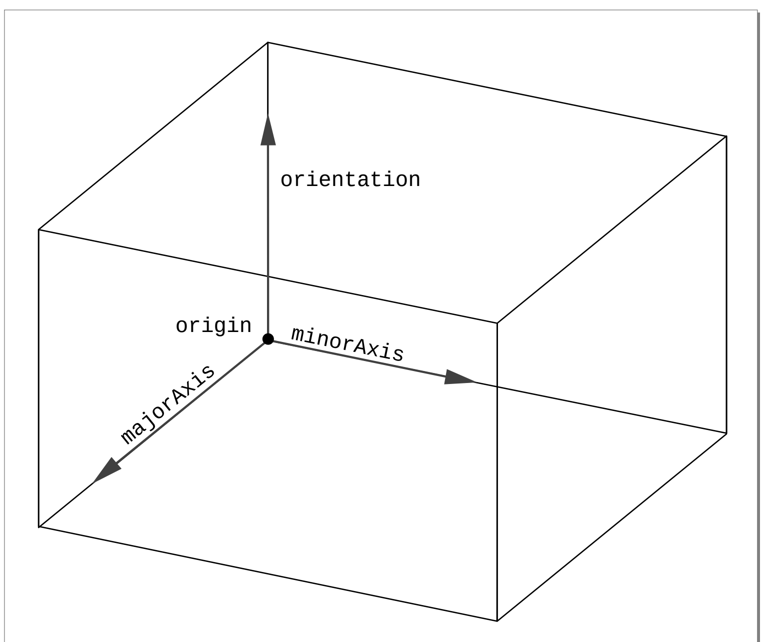
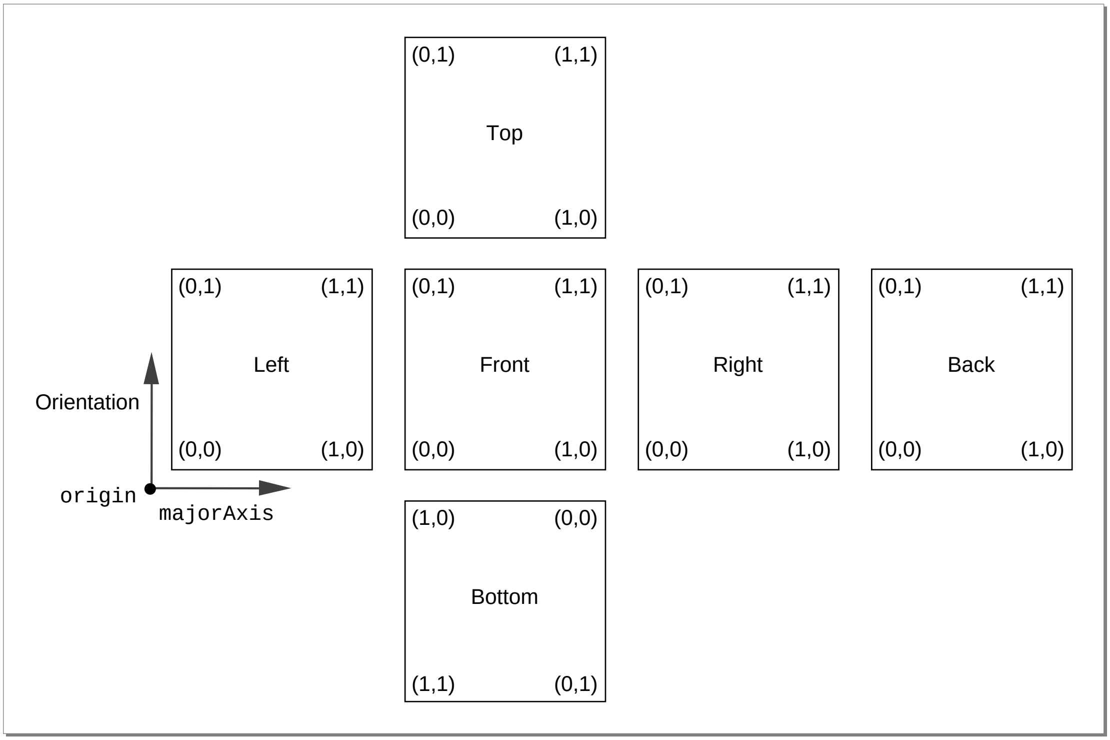


Figure 4-16 shows the standard surface parameterization of a box.

Figure 4-16 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.

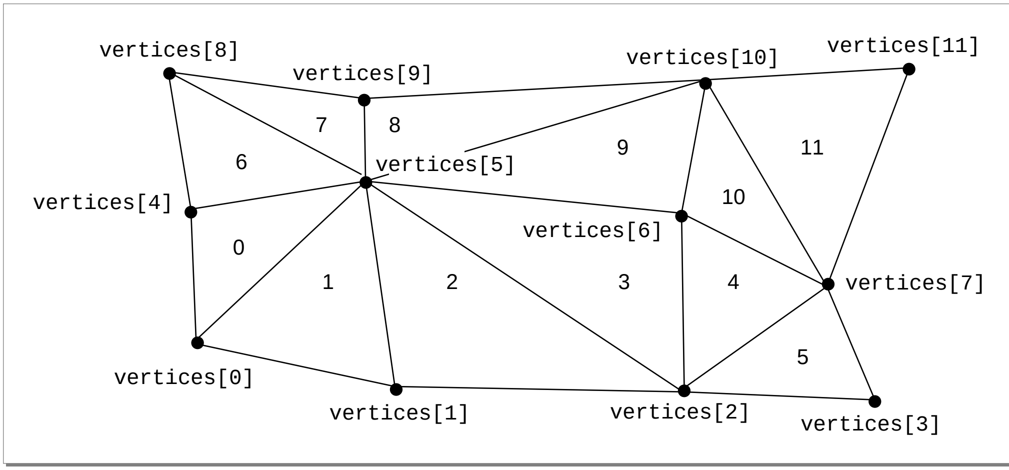
Trigrids

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.

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 4-103 for a description of the routines you can use to create and edit trigrids. Figure 4-17 on page 4-48 shows a trigrid.

Figure 4-17 A trigrid




```
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. You create a new empty mesh by calling `Q3Mesh_New`, and you can add vertices and faces by calling `Q3Mesh_VertexNew` and `Q3Mesh_FaceNew`.

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. ▲

A mesh can have a set of attributes attached to it; you call the `Q3Geometry_SetAttributeSet` function to attach an attribute set to a mesh. In addition, each mesh vertex, face, edge, and corner can have a set of attributes attached to it.

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;
```

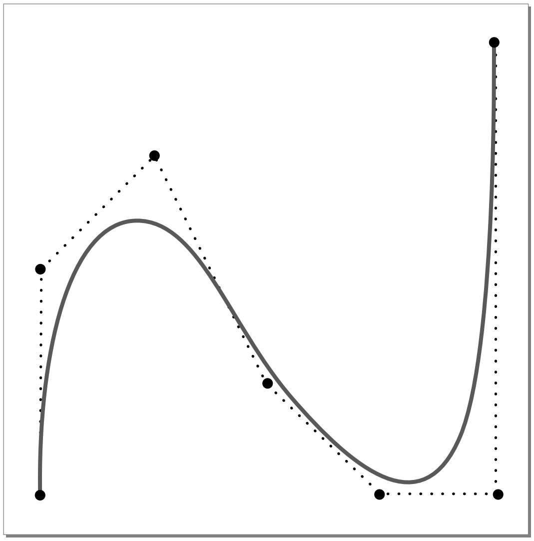
Field descriptions

`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.

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 4-160 for a description of the routines you can use to create and edit NURB curves. Figure 4-18 shows a NURB curve.

Figure 4-18 A NURB curve



```
typedef struct TQ3NURBCurveData {
    unsigned long               order;
    unsigned long               numPoints;
    TQ3RationalPoint4D          *controlPoints;
    float                       *knots;
    TQ3AttributeSet             curveAttributeSet;
} TQ3NURBCurveData;
```

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.

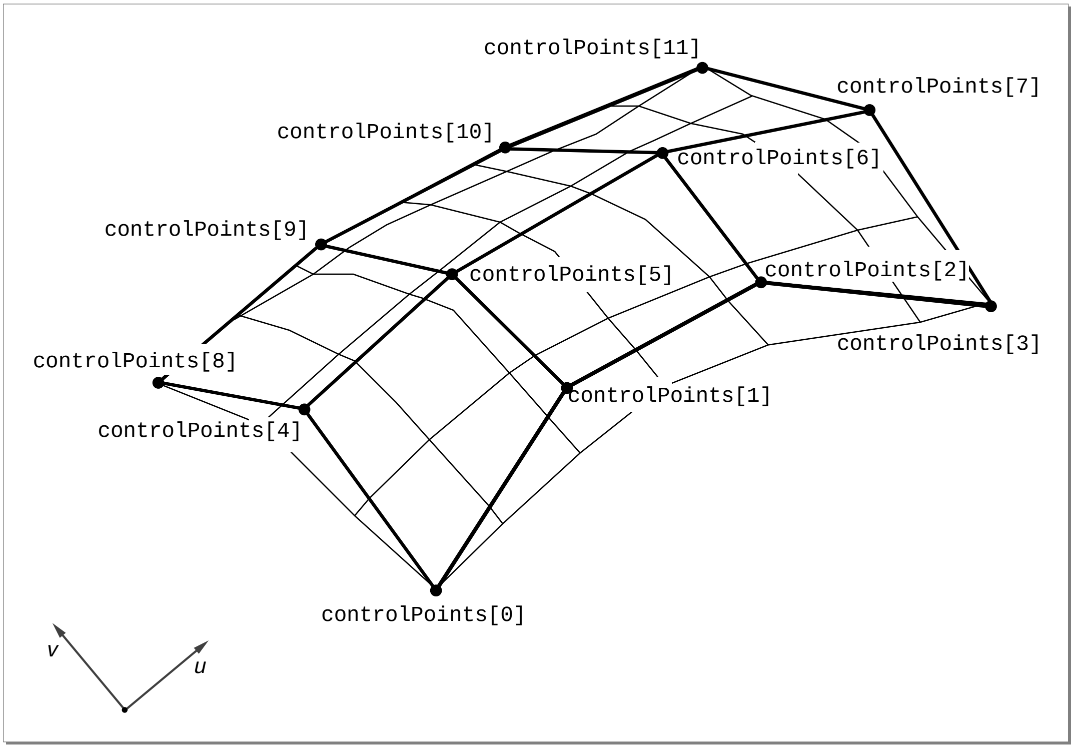
`knots` A pointer to an array of knots that define the NURB curve. The number of knots in a NURB curve is the sum of the values in the `order` and `numPoints` fields. The values in this array must be nondecreasing (but successive values may be equal, up to a multiplicity equivalent to the order of the 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 4-166 for a description of the routines you can use to create and edit NURB patches. Figure 4-19 on page 4-52 shows a NURB patch.

Figure 4-19 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 curve. The number of *u* knots in a NURB curve is the sum of the values in the `uOrder` and `numRows` 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 curve. The number of *v* knots in a NURB curve is the sum of the values in the `vOrder` and `numColumns` 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 the value 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 greater than 1.

`numPoints` — The number of control points that define the NURB trim curve. The value in this field must be greater than 2.

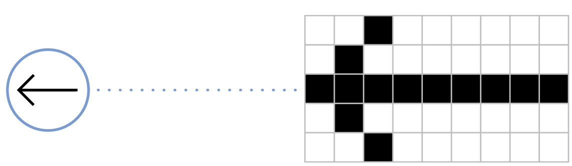
`controlPoints` A pointer to an array of three-dimensional rational control points that define the NURB trim curve.

`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).

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. A marker is defined by the `TQ3MarkerData` data type, which contains a bitmap and a location, together with an optional set of attributes. The bitmap 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 4-20 shows a marker.

Figure 4-20 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.

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
kQ3GeometryTypeGeneralPolygon
kQ3GeometryTypeLine
kQ3GeometryTypeMarker
kQ3GeometryTypeMesh
kQ3GeometryTypeNURBCurve
kQ3GeometryTypeNURBPatch
kQ3GeometryTypePoint
kQ3GeometryTypePolygon
kQ3GeometryTypePolyLine
kQ3GeometryTypeTriangle
kQ3GeometryTypeTriGrid
```

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" on page 4-37 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" on page 4-37 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" on page 4-38 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” on page 4-40 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" on page 4-41 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" on page 4-42 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" on page 4-42 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” on page 4-42 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” on page 4-45 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" on page 4-47 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" on page 4-49 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.

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 4-137.) 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 specified 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 NURB Curves

QuickDraw 3D provides routines that you can use to create and manipulate NURB curves. See "NURB Curves" on page 4-50 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 of knots 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" on page 4-51 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 Markers

QuickDraw 3D provides routines that you can use to create and manipulate markers. See "Markers" on page 4-55 for the definition of a 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`.

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.

Summary of Geometric Objects

C Summary

Constants

```
#define kQ3NURBCurveMaxOrder          16  /*maximum order for NURB curves*/
#define kQ3NURBPatchMaxOrder          11  /*maximum order for NURB patches*/

#define kQ3GeometryTypeBox                   Q3_OBJECT_TYPE('b','o','x',' ')
#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 kQ3GeometryTypePoint                 Q3_OBJECT_TYPE('p','n','t',' ')
#define kQ3GeometryTypePolygon               Q3_OBJECT_TYPE('p','l','y','g')
#define kQ3GeometryTypePolyLine              Q3_OBJECT_TYPE('p','l','y','l')
#define kQ3GeometryTypeTriangle              Q3_OBJECT_TYPE('t','r','n','g')
#define kQ3GeometryTypeTriGrid               Q3_OBJECT_TYPE('t','r','i','g')

typedef enum TQ3PixelType {
   kQ3PixelTypeRGB32                         /*32 bits per pixel*/
} TQ3PixelType;

typedef enum TQ3Endian {
   kQ3EndianBig,
   kQ3EndianLittle
} TQ3Endian;
```

```
typedef enum TQ3GeneralPolygonShapeHint {
    kQ3GeneralPolygonShapeHintComplex,
    kQ3GeneralPolygonShapeHintConcave,
    kQ3GeneralPolygonShapeHintConvex
} TQ3GeneralPolygonShapeHint;

typedef enum TQ3EndCapMasks {
    kQ3EndCapNone                   = 0,
    kQ3EndCapMaskTop                = 1 << 0,
    kQ3EndCapMaskBottom             = 1 << 1
} TQ3EndCapMasks;
```

Data Types

```
typedef unsigned long                               TQ3EndCap;
```

Points

```
typedef struct TQ3Point2D {
    float           x;
    float           y;
} TQ3Point2D;

typedef struct TQ3Point3D {
    float           x;
    float           y;
    float           z;
} TQ3Point3D;
```

Rational Points

```
typedef struct TQ3RationalPoint3D {
    float           x;
    float           y;
    float           w;
} TQ3RationalPoint3D;
```

```
typedef struct TQ3RationalPoint4D {
    float         x;
    float         y;
    float         z;
    float         w;
} TQ3RationalPoint4D;
```

Polar and Spherical Points

```
typedef struct TQ3PolarPoint {
    float         r;
    float         theta;
} TQ3PolarPoint;
```

```
typedef struct TQ3SphericalPoint {
    float         rho;
    float         theta;
    float         phi;
} TQ3SphericalPoint;
```

Vectors

```
typedef struct TQ3Vector2D {
    float         x;
    float         y;
} TQ3Vector2D;
```

```
typedef struct TQ3Vector3D {
    float         x;
    float         y;
    float         z;
} TQ3Vector3D;
```

Quaternions

```
typedef struct TQ3Quaternion {
    float           w;
    float           x;
    float           y;
    float           z;
} TQ3Quaternion;
```

Rays

```
typedef struct TQ3Ray3D {
    TQ3Point3D                      origin;
    TQ3Vector3D                     direction;
} TQ3Ray3D;
```

Parametric Points

```
typedef struct TQ3Param2D {
    float           u;
    float           v;
} TQ3Param2D;
```

```
typedef struct TQ3Param3D {
    float           u;
    float           v;
    float           w;
} TQ3Param3D;
```

Tangents

```
typedef struct TQ3Tangent2D {
    TQ3Vector3D                     uTangent;
    TQ3Vector3D                     vTangent;
} TQ3Tangent2D;
```

```
typedef struct TQ3Tangent3D {
    TQ3Vector3D                     uTangent;
    TQ3Vector3D                     vTangent;
    TQ3Vector3D                     wTangent;
} TQ3Tangent3D;
```

Vertices

```
typedef struct TQ3Vertex3D {
    TQ3Point3D                      point;
    TQ3AttributeSet                 attributeSet;
} TQ3Vertex3D;
```

Matrices

```
typedef struct TQ3Matrix3x3 {
    float                           value[3][3];
} TQ3Matrix3x3;

typedef struct TQ3Matrix4x4 {
    float                           value[4][4];
} TQ3Matrix4x4;
```

Bitmaps and Pixel Maps

```
typedef struct TQ3Bitmap {
    unsigned char                   *image;
    unsigned long                   width;
    unsigned long                   height;
    unsigned long                   rowBytes;
    TQ3Endian                       bitOrder;
} TQ3Bitmap;

typedef struct TQ3Pixmap {
    void                            *image;
    unsigned long                   width;
    unsigned long                   height;
```

```
    unsigned long                   rowBytes;
    unsigned long                   pixelSize;
    TQ3PixelType                    pixelType;
    TQ3Endian                       bitOrder;
    TQ3Endian                       byteOrder;
} TQ3Pixmap;

typedef struct TQ3StoragePixmap {
    TQ3StorageObject                image;
    unsigned long                   width;
    unsigned long                   height;
    unsigned long                   rowBytes;
    unsigned long                   pixelSize;
    TQ3PixelType                    pixelType;
    TQ3Endian                       bitOrder;
    TQ3Endian                       byteOrder;
} TQ3StoragePixmap;
```

Areas and Plane Equations

```
typedef struct TQ3Area {
    TQ3Point2D                      min;
    TQ3Point2D                      max;
} TQ3Area;

typedef struct TQ3PlaneEquation {
    TQ3Vector3D                     normal;
    float                           constant;
} TQ3PlaneEquation;
```

Point Objects

```
typedef struct TQ3PointData {
    TQ3Point3D                      point;
    TQ3AttributeSet                 pointAttributeSet;
} TQ3PointData;
```

Lines

```
typedef struct TQ3LineData {
    TQ3Vertex3D                         vertices[2];
    TQ3AttributeSet                     lineAttributeSet;
} TQ3LineData;
```

Polylines

```
typedef struct TQ3PolyLineData {
    unsigned long                       numVertices;
    TQ3Vertex3D                         *vertices;
    TQ3AttributeSet                     *segmentAttributeSet;
    TQ3AttributeSet                     polyLineAttributeSet;
} TQ3PolyLineData;
```

Triangles

```
typedef struct TQ3TriangleData {
    TQ3Vertex3D                         vertices[3];
    TQ3AttributeSet                     triangleAttributeSet;
} TQ3TriangleData;
```

Simple Polygons

```
typedef struct TQ3PolygonData {
    unsigned long                       numVertices;
    TQ3Vertex3D                         *vertices;
    TQ3AttributeSet                     polygonAttributeSet;
} TQ3PolygonData;
```

General Polygons

```
typedef struct TQ3GeneralPolygonContourData {
    unsigned long                       numVertices;
    TQ3Vertex3D                         *vertices;
} TQ3GeneralPolygonContourData;
```

```
typedef struct TQ3GeneralPolygonData {
    unsigned long                       numContours;
    TQ3GeneralPolygonContourData        *contours;
    TQ3GeneralPolygonShapeHint          shapeHint;
    TQ3AttributeSet                     generalPolygonAttributeSet;
} TQ3GeneralPolygonData;
```

Boxes

```
typedef struct TQ3BoxData {
    TQ3Point3D                      origin;
    TQ3Vector3D                     orientation;
    TQ3Vector3D                     majorAxis;
    TQ3Vector3D                     minorAxis;
    TQ3AttributeSet                 *faceAttributeSet;
    TQ3AttributeSet                 boxAttributeSet;
} TQ3BoxData;
```

Trigrids

```
typedef struct TQ3TriGridData {
    unsigned long                   numRows;
    unsigned long                   numColumns;
    TQ3Vertex3D                     *vertices;
    TQ3AttributeSet                 *facetAttributeSet;
    TQ3AttributeSet                 triGridAttributeSet;
} TQ3TriGridData;
```

Meshes

```
typedef struct TQ3MeshComponentPrivate              *TQ3MeshComponent;
typedef struct TQ3MeshVertexPrivate                 *TQ3MeshVertex;
typedef struct TQ3MeshVertexPrivate                 *TQ3MeshFace;
typedef struct TQ3MeshEdgeRepPrivate                *TQ3MeshEdge;
typedef struct TQ3MeshContourPrivate                *TQ3MeshContour;
```

```
typedef struct TQ3MeshIterator {
    void                            *var1;
    void                            *var2;
    void                            *var3;
    struct {
        void                        *field1;
        char                        field2[4];
    } var4;
} TQ3MeshIterator;
```

NURB Curves

```
typedef struct TQ3NURBCurveData {
    unsigned long                   order;
    unsigned long                   numPoints;
    TQ3RationalPoint4D              *controlPoints;
    float                           *knots;
    TQ3AttributeSet                 curveAttributeSet;
} TQ3NURBCurveData;
```

NURB Patches

```
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;
```

```
typedef struct TQ3NURBPatchTrimLoopData {
    unsigned long                               numTrimCurves;
    TQ3NURBPatchTrimCurveData                   *trimCurves;
} TQ3NURBPatchTrimLoopData;

typedef struct TQ3NURBPatchTrimCurveData {
    unsigned long                               order;
    unsigned long                               numPoints;
    TQ3RationalPoint3D                          *controlPoints;
    float                                       *knots;
} TQ3NURBPatchTrimCurveData;
```

Markers

```
typedef struct TQ3MarkerData {
    TQ3Point3D                    location;
    long                          xOffset;
    long                          yOffset;
    TQ3Bitmap                     bitmap;
    TQ3AttributeSet               markerAttributeSet;
} TQ3MarkerData;
```

Geometric Objects Routines

Managing Geometric Objects

```
TQ3ObjectType Q3Geometry_GetType (
                                TQ3GeometryObject geometry);

TQ3Status Q3Geometry_GetAttributeSet (
                                TQ3GeometryObject geometry,
                                TQ3AttributeSet *attributeSet);

TQ3Status Q3Geometry_SetAttributeSet (
                                TQ3GeometryObject geometry,
                                TQ3AttributeSet attributeSet);

TQ3Status Q3Geometry_Submit    (TQ3GeometryObject geometry,
                                TQ3ViewObject view);
```

Creating and Editing Points

```
TQ3GeometryObject Q3Point_New (const TQ3PointData *pointData);

TQ3Status Q3Point_Submit      (const TQ3PointData *pointData,
                               TQ3ViewObject view);

TQ3Status Q3Point_GetData     (TQ3GeometryObject point,
                               TQ3PointData *pointData);

TQ3Status Q3Point_SetData     (TQ3GeometryObject point,
                               const TQ3PointData *pointData);

TQ3Status Q3Point_EmptyData   (TQ3PointData *pointData);

TQ3Status Q3Point_GetPosition (TQ3GeometryObject point,
                               TQ3Point3D *position);

TQ3Status Q3Point_SetPosition (TQ3GeometryObject point,
                               const TQ3Point3D *position);
```

Creating and Editing Lines

```
TQ3GeometryObject Q3Line_New  (const TQ3LineData *lineData);

TQ3Status Q3Line_Submit       (const TQ3LineData *lineData,
                               TQ3ViewObject view);

TQ3Status Q3Line_GetData      (TQ3GeometryObject line,
                               TQ3LineData *lineData);

TQ3Status Q3Line_SetData      (TQ3GeometryObject line,
                               const TQ3LineData *lineData);

TQ3Status Q3Line_GetVertexPosition (
                               TQ3GeometryObject line,
                               unsigned long index,
                               TQ3Point3D *position);

TQ3Status Q3Line_SetVertexPosition (
                               TQ3GeometryObject line,
                               unsigned long index,
                               const TQ3Point3D *position);
```

```
TQ3Status Q3Line_GetVertexAttributeSet (
                              TQ3GeometryObject line,
                              unsigned long index,
                              TQ3AttributeSet *attributeSet);

TQ3Status Q3Line_SetVertexAttributeSet (
                              TQ3GeometryObject line,
                              unsigned long index,
                              TQ3AttributeSet attributeSet);

TQ3Status Q3Line_EmptyData    (TQ3LineData *lineData);
```

Creating and Editing Polylines

```
TQ3GeometryObject Q3PolyLine_New (
                              const TQ3PolyLineData *polyLineData);

TQ3Status Q3PolyLine_Submit   (const TQ3PolyLineData *polyLineData,
                              TQ3ViewObject view);

TQ3Status Q3PolyLine_GetData  (TQ3GeometryObject polyLine,
                              TQ3PolyLineData *polyLineData);

TQ3Status Q3PolyLine_SetData  (TQ3GeometryObject polyLine,
                              const TQ3PolyLineData *polyLineData);

TQ3Status Q3PolyLine_EmptyData(TQ3PolyLineData *polyLineData);

TQ3Status Q3PolyLine_GetVertexPosition (
                              TQ3GeometryObject polyLine,
                              unsigned long index,
                              TQ3Point3D *position);

TQ3Status Q3PolyLine_SetVertexPosition (
                              TQ3GeometryObject polyLine,
                              unsigned long index,
                              const TQ3Point3D *position);
```

```
TQ3Status Q3PolyLine_GetVertexAttributeSet (
                                TQ3GeometryObject polyLine,
                                unsigned long index,
                                TQ3AttributeSet *attributeSet);

TQ3Status Q3PolyLine_SetVertexAttributeSet (
                                TQ3GeometryObject polyLine,
                                unsigned long index,
                                TQ3AttributeSet attributeSet);

TQ3Status Q3PolyLine_GetSegmentAttributeSet (
                                TQ3GeometryObject polyLine,
                                unsigned long index,
                                TQ3AttributeSet *attributeSet);

TQ3Status Q3PolyLine_SetSegmentAttributeSet (
                                TQ3GeometryObject polyLine,
                                unsigned long index,
                                TQ3AttributeSet attributeSet);
```

Creating and Editing Triangles

```
TQ3GeometryObject Q3Triangle_New (
                                const TQ3TriangleData *triangleData);

TQ3Status Q3Triangle_Submit   (const TQ3TriangleData *triangleData,
                                TQ3ViewObject view);

TQ3Status Q3Triangle_GetData  (TQ3GeometryObject triangle,
                                TQ3TriangleData *triangleData);

TQ3Status Q3Triangle_SetData  (TQ3GeometryObject triangle,
                                const TQ3TriangleData *triangleData);

TQ3Status Q3Triangle_EmptyData(TQ3TriangleData *triangleData);

TQ3Status Q3Triangle_GetVertexPosition (
                                TQ3GeometryObject triangle,
                                unsigned long index,
                                TQ3Point3D *point);
```

```
TQ3Status Q3Triangle_SetVertexPosition (
                              TQ3GeometryObject triangle,
                              unsigned long index,
                              const TQ3Point3D *point);

TQ3Status Q3Triangle_GetVertexAttributeSet (
                              TQ3GeometryObject triangle,
                              unsigned long index,
                              TQ3AttributeSet *attributeSet);

TQ3Status Q3Triangle_SetVertexAttributeSet (
                              TQ3GeometryObject triangle,
                              unsigned long index,
                              TQ3AttributeSet attributeSet);
```

Creating and Editing Simple Polygons

```
TQ3GeometryObject Q3Polygon_New (
                              const TQ3PolygonData *polygonData);

TQ3Status Q3Polygon_Submit   (const TQ3PolygonData *polygonData,
                              TQ3ViewObject view);

TQ3Status Q3Polygon_GetData  (TQ3GeometryObject polygon,
                              TQ3PolygonData *polygonData);

TQ3Status Q3Polygon_SetData  (TQ3GeometryObject polygon,
                              const TQ3PolygonData *polygonData);

TQ3Status Q3Polygon_EmptyData (TQ3PolygonData *polygonData);

TQ3Status Q3Polygon_GetVertexPosition (
                              TQ3GeometryObject polygon,
                              unsigned long index,
                              TQ3Point3D *point);

TQ3Status Q3Polygon_SetVertexPosition (
                              TQ3GeometryObject polygon,
                              unsigned long index,
                              const TQ3Point3D *point);
```

```
TQ3Status Q3Polygon_GetVertexAttributeSet (
                              TQ3GeometryObject polygon,
                              unsigned long index,
                              TQ3AttributeSet *attributeSet);

TQ3Status Q3Polygon_SetVertexAttributeSet (
                              TQ3GeometryObject polygon,
                              unsigned long index,
                              TQ3AttributeSet attributeSet);
```

Creating and Editing General Polygons

```
TQ3GeometryObject Q3GeneralPolygon_New (
                              const TQ3GeneralPolygonData
                              *generalPolygonData);

TQ3Status Q3GeneralPolygon_Submit (
                              const TQ3GeneralPolygonData
                              *generalPolygonData,
                              TQ3ViewObject view);

TQ3Status Q3GeneralPolygon_GetData (
                              TQ3GeometryObject generalPolygon,
                              TQ3GeneralPolygonData *generalPolygonData);

TQ3Status Q3GeneralPolygon_SetData (
                              TQ3GeometryObject generalPolygon,
                              const TQ3GeneralPolygonData
                              *generalPolygonData);

TQ3Status Q3GeneralPolygon_EmptyData (
                              TQ3GeneralPolygonData *generalPolygonData);

TQ3Status Q3GeneralPolygon_GetVertexPosition (
                              TQ3GeometryObject generalPolygon,
                              unsigned long contourIndex,
                              unsigned long pointIndex,
                              TQ3Point3D *position);
```

```
TQ3Status Q3GeneralPolygon_SetVertexPosition (
                                TQ3GeometryObject generalPolygon,
                                unsigned long contourIndex,
                                unsigned long pointIndex,
                                const TQ3Point3D *position);

TQ3Status Q3GeneralPolygon_GetVertexAttributeSet (
                                TQ3GeometryObject generalPolygon,
                                unsigned long contourIndex,
                                unsigned long pointIndex,
                                TQ3AttributeSet *attributeSet);

TQ3Status Q3GeneralPolygon_SetVertexAttributeSet (
                                TQ3GeometryObject generalPolygon,
                                unsigned long contourIndex,
                                unsigned long pointIndex,
                                TQ3AttributeSet attributeSet);

TQ3Status Q3GeneralPolygon_GetShapeHint (
                                TQ3GeometryObject generalPolygon,
                                TQ3GeneralPolygonShapeHint *shapeHint);

TQ3Status Q3GeneralPolygon_SetShapeHint (
                                TQ3GeometryObject generalPolygon,
                                TQ3GeneralPolygonShapeHint shapeHint);
```

Creating and Editing Boxes

```
TQ3GeometryObject Q3Box_New   (const TQ3BoxData *boxData);

TQ3Status Q3Box_Submit        (const TQ3BoxData *boxData,
                               TQ3ViewObject view);

TQ3Status Q3Box_GetData       (TQ3GeometryObject box,
                               TQ3BoxData *boxData);

TQ3Status Q3Box_SetData       (TQ3GeometryObject box,
                               const TQ3BoxData *boxData);

TQ3Status Q3Box_EmptyData     (TQ3BoxData *boxData);
```

```
TQ3Status Q3Box_GetOrigin     (TQ3GeometryObject box, TQ3Point3D *origin);

TQ3Status Q3Box_SetOrigin     (TQ3GeometryObject box,
                               const TQ3Point3D *origin);

TQ3Status Q3Box_GetOrientation(TQ3GeometryObject box,
                               TQ3Vector3D *orientation);

TQ3Status Q3Box_SetOrientation(TQ3GeometryObject box,
                               const TQ3Vector3D *orientation);

TQ3Status Q3Box_GetMajorAxis  (TQ3GeometryObject box,
                               TQ3Vector3D *majorAxis);

TQ3Status Q3Box_SetMajorAxis  (TQ3GeometryObject box,
                               const TQ3Vector3D *majorAxis);

TQ3Status Q3Box_GetMinorAxis  (TQ3GeometryObject box,
                               TQ3Vector3D *minorAxis);

TQ3Status Q3Box_SetMinorAxis  (TQ3GeometryObject box,
                               const TQ3Vector3D *minorAxis);

TQ3Status Q3Box_GetFaceAttributeSet (
                               TQ3GeometryObject box,
                               unsigned long faceIndex,
                               TQ3AttributeSet *faceAttributeSet);

TQ3Status Q3Box_SetFaceAttributeSet (
                               TQ3GeometryObject box,
                               unsigned long faceIndex,
                               TQ3AttributeSet faceAttributeSet);
```

Creating and Editing Trigrids

```
TQ3GeometryObject Q3TriGrid_New (
                               const TQ3TriGridData *triGridData);

TQ3Status Q3TriGrid_Submit    (const TQ3TriGridData *triGridData,
                               TQ3ViewObject view);
```

```
TQ3Status Q3TriGrid_GetData   (TQ3GeometryObject triGrid,
                               TQ3TriGridData *triGridData);

TQ3Status Q3TriGrid_SetData   (TQ3GeometryObject triGrid,
                               const TQ3TriGridData *triGridData);

TQ3Status Q3TriGrid_EmptyData (TQ3TriGridData *triGridData);

TQ3Status Q3TriGrid_GetVertexPosition (
                               TQ3GeometryObject triGrid,
                               unsigned long rowIndex,
                               unsigned long columnIndex,
                               TQ3Point3D *position);

TQ3Status Q3TriGrid_SetVertexPosition (
                               TQ3GeometryObject triGrid,
                               unsigned long rowIndex,
                               unsigned long columnIndex,
                               const TQ3Point3D *position);

TQ3Status Q3TriGrid_GetVertexAttributeSet (
                               TQ3GeometryObject triGrid,
                               unsigned long rowIndex,
                               unsigned long columnIndex,
                               TQ3AttributeSet *attributeSet);

TQ3Status Q3TriGrid_SetVertexAttributeSet (
                               TQ3GeometryObject triGrid,
                               unsigned long rowIndex,
                               unsigned long columnIndex,
                               TQ3AttributeSet attributeSet);

TQ3Status Q3TriGrid_GetFacetAttributeSet (
                               TQ3GeometryObject triGrid,
                               unsigned long faceIndex,
                               TQ3AttributeSet *facetAttributeSet);
```

```
TQ3Status Q3TriGrid_SetFacetAttributeSet (
                              TQ3GeometryObject triGrid,
                              unsigned long faceIndex,
                              TQ3AttributeSet facetAttributeSet);
```

Creating and Editing Meshes

```
TQ3GeometryObject Q3Mesh_New  (void);

TQ3MeshVertex Q3Mesh_VertexNew(TQ3GeometryObject mesh,
                              const TQ3Vertex3D *vertex);

TQ3Status Q3Mesh_VertexDelete (TQ3GeometryObject mesh, TQ3MeshVertex vertex);

TQ3MeshFace Q3Mesh_FaceNew    (TQ3GeometryObject mesh,
                              unsigned long numVertices,
                              const TQ3MeshVertex *vertices,
                              TQ3AttributeSet attributeSet);

TQ3Status Q3Mesh_FaceDelete   (TQ3GeometryObject mesh, TQ3MeshFace face);

TQ3Status Q3Mesh_DelayUpdates (TQ3GeometryObject mesh);

TQ3Status Q3Mesh_ResumeUpdates(TQ3GeometryObject mesh);

TQ3MeshContour Q3Mesh_FaceToContour (
                              TQ3GeometryObject mesh,
                              TQ3MeshFace containerFace,
                              TQ3MeshFace face);

TQ3MeshFace Q3Mesh_ContourToFace (
                              TQ3GeometryObject mesh,
                              TQ3MeshContour contour);

TQ3Status Q3Mesh_GetNumComponents (
                              TQ3GeometryObject mesh,
                              unsigned long *numComponents);

TQ3Status Q3Mesh_GetNumEdges  (TQ3GeometryObject mesh,
                              unsigned long *numEdges);
```

```
TQ3Status Q3Mesh_GetNumVertices (
                              TQ3GeometryObject mesh,
                              unsigned long *numVertices);

TQ3Status Q3Mesh_GetNumFaces  (TQ3GeometryObject mesh,
                              unsigned long *numFaces);

TQ3Status Q3Mesh_GetNumCorners(TQ3GeometryObject mesh,
                              unsigned long *numCorners);

TQ3Status Q3Mesh_GetOrientable(TQ3GeometryObject mesh,
                              TQ3Boolean *orientable);

TQ3Status Q3Mesh_GetComponentNumVertices (
                              TQ3GeometryObject mesh,
                              TQ3MeshComponent component,
                              unsigned long *numVertices);

TQ3Status Q3Mesh_GetComponentNumEdges (
                              TQ3GeometryObject mesh,
                              TQ3MeshComponent component,
                              unsigned long *numEdges);

TQ3Status Q3Mesh_GetComponentBoundingBox (
                              TQ3GeometryObject mesh,
                              TQ3MeshComponent component,
                              TQ3BoundingBox *boundingBox);

TQ3Status Q3Mesh_GetComponentOrientable (
                              TQ3GeometryObject mesh,
                              TQ3MeshComponent component,
                              TQ3Boolean *orientable);

TQ3Status Q3Mesh_GetVertexCoordinates (
                              TQ3GeometryObject mesh,
                              TQ3MeshVertex vertex,
                              TQ3Point3D *coordinates);
```

```
TQ3Status Q3Mesh_SetVertexCoordinates (
                              TQ3GeometryObject mesh,
                              TQ3MeshVertex vertex,
                              const TQ3Point3D *coordinates);

TQ3Status Q3Mesh_GetVertexIndex (
                              TQ3GeometryObject mesh,
                              TQ3MeshVertex vertex,
                              unsigned long *index);

TQ3Status Q3Mesh_GetVertexOnBoundary (
                              TQ3GeometryObject mesh,
                              TQ3MeshVertex vertex,
                              TQ3Boolean *onBoundary);

TQ3Status Q3Mesh_GetVertexComponent (
                              TQ3GeometryObject mesh,
                              TQ3MeshVertex vertex,
                              TQ3MeshComponent *component);

TQ3Status Q3Mesh_GetVertexAttributeSet (
                              TQ3GeometryObject mesh,
                              TQ3MeshVertex vertex,
                              TQ3AttributeSet *attributeSet);

TQ3Status Q3Mesh_SetVertexAttributeSet (
                              TQ3GeometryObject mesh,
                              TQ3MeshVertex vertex,
                              TQ3AttributeSet attributeSet);

TQ3Status Q3Mesh_GetFaceNumVertices (
                              TQ3GeometryObject mesh,
                              TQ3MeshFace face,
                              unsigned long *numVertices);

TQ3Status Q3Mesh_GetFacePlaneEquation (
                              TQ3GeometryObject mesh,
                              TQ3MeshFace face,
                              TQ3PlaneEquation *planeEquation);
```

```
TQ3Status Q3Mesh_GetFaceNumContours (
                              TQ3GeometryObject mesh,
                              TQ3MeshFace face,
                              unsigned long *numContours);

TQ3Status Q3Mesh_GetFaceIndex (TQ3GeometryObject mesh,
                              TQ3MeshFace face,
                              unsigned long *index);

TQ3Status Q3Mesh_GetFaceComponent (
                              TQ3GeometryObject mesh,
                              TQ3MeshFace face,
                              TQ3MeshComponent *component);

TQ3Status Q3Mesh_GetFaceAttributeSet (
                              TQ3GeometryObject mesh,
                              TQ3MeshFace face,
                              TQ3AttributeSet *attributeSet);

TQ3Status Q3Mesh_SetFaceAttributeSet (
                              TQ3GeometryObject mesh,
                              TQ3MeshFace face,
                              TQ3AttributeSet attributeSet);

TQ3Status Q3Mesh_GetEdgeVertices (
                              TQ3GeometryObject mesh,
                              TQ3MeshEdge edge,
                              TQ3MeshVertex *vertex1,
                              TQ3MeshVertex *vertex2);

TQ3Status Q3Mesh_GetEdgeFaces (TQ3GeometryObject mesh,
                              TQ3MeshEdge edge,
                              TQ3MeshFace *face1,
                              TQ3MeshFace *face2);

TQ3Status Q3Mesh_GetEdgeOnBoundary (
                              TQ3GeometryObject mesh,
                              TQ3MeshEdge edge,
                              TQ3Boolean *onBoundary);
```

```
TQ3Status Q3Mesh_GetEdgeComponent (
                                TQ3GeometryObject mesh,
                                TQ3MeshEdge edge,
                                TQ3MeshComponent *component);

TQ3Status Q3Mesh_GetEdgeAttributeSet (
                                TQ3GeometryObject mesh,
                                TQ3MeshEdge edge,
                                TQ3AttributeSet *attributeSet);

TQ3Status Q3Mesh_SetEdgeAttributeSet (
                                TQ3GeometryObject mesh,
                                TQ3MeshEdge edge,
                                TQ3AttributeSet attributeSet);

TQ3Status Q3Mesh_GetContourFace (
                                TQ3GeometryObject mesh,
                                TQ3MeshContour contour,
                                TQ3MeshFace *face);

TQ3Status Q3Mesh_GetContourNumVertices (
                                TQ3GeometryObject mesh,
                                TQ3MeshContour contour,
                                unsigned long *numVertices);

TQ3Status Q3Mesh_GetCornerAttributeSet (
                                TQ3GeometryObject mesh,
                                TQ3MeshVertex vertex,
                                TQ3MeshFace face,
                                TQ3AttributeSet *attributeSet);

TQ3Status Q3Mesh_SetCornerAttributeSet (
                                TQ3GeometryObject mesh,
                                TQ3MeshVertex vertex,
                                TQ3MeshFace face,
                                TQ3AttributeSet attributeSet);
```

Traversing Mesh Components, Vertices, Faces, and Edges

```
TQ3MeshComponent Q3Mesh_FirstMeshComponent (
                              TQ3GeometryObject mesh,
                              TQ3MeshIterator *iterator);

TQ3MeshComponent Q3Mesh_NextMeshComponent (
                              TQ3MeshIterator *iterator);

TQ3MeshVertex Q3Mesh_FirstComponentVertex (
                              TQ3MeshComponent component,
                              TQ3MeshIterator *iterator);

TQ3MeshVertex Q3Mesh_NextComponentVertex (
                              TQ3MeshIterator *iterator);

TQ3MeshEdge Q3Mesh_FirstComponentEdge (
                              TQ3MeshComponent component,
                              TQ3MeshIterator *iterator);

TQ3MeshEdge Q3Mesh_NextComponentEdge (
                              TQ3MeshIterator *iterator);

TQ3MeshVertex Q3Mesh_FirstMeshVertex (
                              TQ3GeometryObject mesh,
                              TQ3MeshIterator *iterator);

TQ3MeshVertex Q3Mesh_NextMeshVertex (
                              TQ3MeshIterator *iterator);

TQ3MeshFace Q3Mesh_FirstMeshFace (
                              TQ3GeometryObject mesh,
                              TQ3MeshIterator *iterator);

TQ3MeshFace Q3Mesh_NextMeshFace (
                              TQ3MeshIterator *iterator);

TQ3MeshEdge Q3Mesh_FirstMeshEdge (
                              TQ3GeometryObject mesh,
                              TQ3MeshIterator *iterator);

TQ3MeshEdge Q3Mesh_NextMeshEdge(TQ3MeshIterator *iterator);
```

```
TQ3MeshEdge Q3Mesh_FirstVertexEdge (
                                TQ3MeshVertex vertex,
                                TQ3MeshIterator *iterator);

TQ3MeshEdge Q3Mesh_NextVertexEdge (
                                TQ3MeshIterator *iterator);

TQ3MeshVertex Q3Mesh_FirstVertexVertex (
                                TQ3MeshVertex vertex,
                                TQ3MeshIterator *iterator);

TQ3MeshVertex Q3Mesh_NextVertexVertex (
                                TQ3MeshIterator *iterator);

TQ3MeshFace Q3Mesh_FirstVertexFace (
                                TQ3MeshVertex vertex,
                                TQ3MeshIterator *iterator);

TQ3MeshFace Q3Mesh_NextVertexFace (
                                TQ3MeshIterator *iterator);

TQ3MeshEdge Q3Mesh_FirstFaceEdge (
                                TQ3MeshFace face,
                                TQ3MeshIterator *iterator);

TQ3MeshEdge Q3Mesh_NextFaceEdge (
                                TQ3MeshIterator *iterator);

TQ3MeshVertex Q3Mesh_FirstFaceVertex (
                                TQ3MeshFace face, TQ3MeshIterator *iterator);

TQ3MeshVertex Q3Mesh_NextFaceVertex (
                                TQ3MeshIterator *iterator);

TQ3MeshFace Q3Mesh_FirstFaceFace (
                                TQ3MeshFace face,
                                TQ3MeshIterator *iterator);

TQ3MeshFace Q3Mesh_NextFaceFace (
                                TQ3MeshIterator *iterator);
```

```
TQ3MeshContour Q3Mesh_FirstFaceContour (
                              TQ3MeshFace face, TQ3MeshIterator *iterator);

TQ3MeshContour Q3Mesh_NextFaceContour (
                              TQ3MeshIterator *iterator);

TQ3MeshEdge Q3Mesh_FirstContourEdge (
                              TQ3MeshContour contour,
                              TQ3MeshIterator *iterator);

TQ3MeshEdge Q3Mesh_NextContourEdge (
                              TQ3MeshIterator *iterator);

TQ3MeshVertex Q3Mesh_FirstContourVertex (
                              TQ3MeshContour contour,
                              TQ3MeshIterator *iterator);

TQ3MeshVertex Q3Mesh_NextContourVertex (
                              TQ3MeshIterator *iterator);

TQ3MeshFace Q3Mesh_FirstContourFace (
                              TQ3MeshContour contour,
                              TQ3MeshIterator *iterator);

TQ3MeshFace Q3Mesh_NextContourFace (
                              TQ3MeshIterator *iterator);

#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)) )
```

Creating and Editing NURB Curves

```
TQ3GeometryObject Q3NURBCurve_New (
                                const TQ3NURBCurveData *curveData);

TQ3Status Q3NURBCurve_Submit  (const TQ3NURBCurveData *curveData,
                                TQ3ViewObject view);

TQ3Status Q3NURBCurve_GetData (TQ3GeometryObject curve,
                                TQ3NURBCurveData *nurbCurveData);

TQ3Status Q3NURBCurve_SetData (TQ3GeometryObject curve,
                                const TQ3NURBCurveData *nurbCurveData);

TQ3Status Q3NURBCurve_EmptyData (
                                TQ3NURBCurveData *nurbCurveData);

TQ3Status Q3NURBCurve_GetControlPoint (
                                TQ3GeometryObject curve,
                                unsigned long pointIndex,
                                TQ3RationalPoint4D *point4D);

TQ3Status Q3NURBCurve_SetControlPoint (
                                TQ3GeometryObject curve,
                                unsigned long pointIndex,
                                const TQ3RationalPoint4D *point4D);

TQ3Status Q3NURBCurve_GetKnot (TQ3GeometryObject curve,
                                unsigned long knotIndex,
                                float *knotValue);

TQ3Status Q3NURBCurve_SetKnot (TQ3GeometryObject curve,
                                unsigned long knotIndex,
                                float knotValue);
```

Creating and Editing NURB Patches

```
TQ3GeometryObject Q3NURBPatch_New (
                                const TQ3NURBPatchData *nurbPatchData);
```

```
TQ3Status Q3NURBPatch_Submit  (const TQ3NURBPatchData *nurbPatchData,
                               TQ3ViewObject view);

TQ3Status Q3NURBPatch_GetData (TQ3GeometryObject nurbPatch,
                               TQ3NURBPatchData *nurbPatchData);

TQ3Status Q3NURBPatch_SetData (TQ3GeometryObject nurbPatch,
                               const TQ3NURBPatchData *nurbPatchData);

TQ3Status Q3NURBPatch_EmptyData (
                               TQ3NURBPatchData *nurbPatchData);

TQ3Status Q3NURBPatch_GetControlPoint (
                               TQ3GeometryObject nurbPatch,
                               unsigned long rowIndex,
                               unsigned long columnIndex,
                               TQ3RationalPoint4D *point4D);

TQ3Status Q3NURBPatch_SetControlPoint (
                               TQ3GeometryObject nurbPatch,
                               unsigned long rowIndex,
                               unsigned long columnIndex,
                               const TQ3RationalPoint4D *point4D);

TQ3Status Q3NURBPatch_GetUKnot(TQ3GeometryObject nurbPatch,
                               unsigned long knotIndex,
                               float *knotValue);

TQ3Status Q3NURBPatch_SetUKnot(TQ3GeometryObject nurbPatch,
                               unsigned long knotIndex,
                               float knotValue);

TQ3Status Q3NURBPatch_GetVKnot(TQ3GeometryObject nurbPatch,
                               unsigned long knotIndex,
                               float *knotValue);

TQ3Status Q3NURBPatch_SetVKnot(TQ3GeometryObject nurbPatch,
                               unsigned long knotIndex,
                               float knotValue);
```

Creating and Editing Markers

```
TQ3GeometryObject Q3Marker_New(const TQ3MarkerData *markerData);

TQ3Status Q3Marker_Submit     (const TQ3MarkerData *markerData,
                               TQ3ViewObject view);

TQ3Status Q3Marker_GetData    (TQ3GeometryObject marker,
                               TQ3MarkerData *markerData);

TQ3Status Q3Marker_SetData    (TQ3GeometryObject marker,
                               const TQ3MarkerData *markerData);

TQ3Status Q3Marker_EmptyData  (TQ3MarkerData *markerData);

TQ3Status Q3Marker_GetPosition(TQ3GeometryObject marker,
                               TQ3Point3D *location);

TQ3Status Q3Marker_SetPosition(TQ3GeometryObject marker,
                               const TQ3Point3D *location);

TQ3Status Q3Marker_GetXOffset (TQ3GeometryObject marker, long *xOffset);

TQ3Status Q3Marker_SetXOffset (TQ3GeometryObject marker, long xOffset);

TQ3Status Q3Marker_GetYOffset (TQ3GeometryObject marker, long *yOffset);

TQ3Status Q3Marker_SetYOffset (TQ3GeometryObject marker, long yOffset);

TQ3Status Q3Marker_GetBitmap  (TQ3GeometryObject marker, TQ3Bitmap *bitmap);

TQ3Status Q3Marker_SetBitmap  (TQ3GeometryObject marker,
                               const TQ3Bitmap *bitmap);
```

Managing Bitmaps

```
TQ3Status Q3Bitmap_Empty      (TQ3Bitmap *bitmap);

unsigned long Q3Bitmap_GetImageSize (
                               unsigned long width, unsigned long height);
```

Errors, Warnings, and Notices

```
kQ3ErrorDegenerateGeometry
kQ3ErrorGeometryInsufficientNumberOfPoints
kQ3ErrorVector3DNotUnitLength
kQ3WarningQuaternionEntriesAreZero
kQ3NoticeMeshVertexHasNoComponent
kQ3NoticeMeshInvalidVertexFacePair
kQ3NoticeMeshEdgeVertexDoNotCorrespond
kQ3NoticeMeshEdgeIsNotBoundary
```

Attribute Objects

Contents

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 5-13 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” on page 5-14 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
} 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" on page 5-6 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" on page 5-9 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 5-9 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:

```
kQ3MethodTypeObjectReadData
kQ3MethodTypeObjectTraverse
kQ3MethodTypeObjectWrite
kQ3MethodTypeElementCopyAdd
kQ3MethodTypeElementDelete
kQ3MethodTypeElementCopyDuplicate
kQ3MethodTypeElementCopyGet
kQ3MethodTypeElementCopyReplace
kQ3MethodTypeAttributeCopyInherit
kQ3MethodTypeAttributeInherit
```

Listing 5-3 defines a simple attribute metahandler. See "Defining an Object Metahandler," beginning on page 3-15 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);
    }
}
```

As you can see, 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 defined 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 custom attribute. A dispose method is passed a pointer to the data associated with an attribute. Your dispose method should deallocate any storage you allocated yourself. Listing 5-4 shows a simple dispose method.

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 defined 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
} 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.

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 `Q3ObjectClass_Unregister` function.

Note
For complete details on adding custom attribute types, see "Defining Custom Attribute Types," beginning on page 5-9. ◆

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)`.) You also need to have your handler's dispose method deallocate any dynamically sized blocks.

SEE ALSO

See page 5-24 for information on methods for a custom attribute type.

Application-Defined Routines

This section describes the methods you can implement to handle a custom attribute.

TQ3AttributeCopyInheritMethod

You can define an attribute inheritance method to copy attributes during inheritance.

```
typedef TQ3Status (*TQ3AttributeCopyInheritMethod) (
                    const void *fromInternalAttribute,
                    void *toInternalAttribute);
```

`fromInternalAttribute`
: A pointer to the attribute data associated with an attribute having your custom attribute type.

`toInternalAttribute`
: On entry, a pointer to an uninitialized block of memory large enough to contain the attribute data associated with an attribute having your custom attribute type.

DESCRIPTION

Your `TQ3AttributeCopyInheritMethod` function should copy the attribute data pointed to by the `fromInternalAttribute` parameter into the location pointed to by the `toInternalAttribute` parameter. This method is called whenever the `Q3AttributeSet_Inherit` function is used to copy an attribute of your custom type from one set to another set.

You should strive to make your `TQ3AttributeCopyInheritMethod` method as fast as possible. For example, if your custom element contains objects, you should call the `Q3Shared_GetReference` function instead of the `Q3Object_Duplicate` function.

RESULT CODES

Your `TQ3AttributeCopyInheritMethod` function should return `kQ3Success` if it is successful and `kQ3Failure` otherwise.

TQ3AttributeInheritMethod

You must define an attribute inheritance method to manage inheritance of your custom attribute type.

```
typedef TQ3Boolean TQ3AttributeInheritMethod;
```

DESCRIPTION

Your `TQ3AttributeInheritMethod` function should return a Boolean value that indicates whether attributes of your custom type should be inherited (`kQ3True`) or not (`kQ3False`).

Summary of Attribute Objects

C Summary

Constants

```
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
} TQ3AttributeTypes;

#define kQ3MethodTypeAttributeCopyInherit    Q3_METHOD_TYPE('a','c','p','i')
#define kQ3MethodTypeAttributeInherit        Q3_METHOD_TYPE('i','n','h','t')
```

Data Types

```
typedef TQ3ElementType                          TQ3AttributeType;
```

Attribute Objects Routines

Drawing Attributes

```
TQ3Status Q3Attribute_Submit  (TQ3AttributeType attributeType,
                               const void *data,
                               TQ3ViewObject view);
```

Creating and Managing Attribute Sets

```
TQ3AttributeSet Q3AttributeSet_New (
                               void);

TQ3Status Q3AttributeSet_Add  (TQ3AttributeSet attributeSet,
                               TQ3AttributeType type,
                               const void *data);

TQ3Boolean Q3AttributeSet_Contains (
                               TQ3AttributeSet attributeSet,
                               TQ3AttributeType attributeType);

TQ3Status Q3AttributeSet_Get  (TQ3AttributeSet attributeSet,
                               TQ3AttributeType type,
                               void *data);

TQ3Status Q3AttributeSet_GetNextAttributeType (
                               TQ3AttributeSet source,
                               TQ3AttributeType *type);

TQ3Status Q3AttributeSet_Empty(TQ3AttributeSet target);

TQ3Status Q3AttributeSet_Clear(TQ3AttributeSet attributeSet,
                               TQ3AttributeType type);

TQ3Status Q3AttributeSet_Submit (
                               TQ3AttributeSet attributeSet,
                               TQ3ViewObject view);
```

```
TQ3Status Q3AttributeSet_Inherit (
                                TQ3AttributeSet parent,
                                TQ3AttributeSet child,
                                TQ3AttributeSet result);
```

Registering Custom Attributes

```
TQ3ObjectClass Q3AttributeClass_Register (
                                TQ3AttributeType attributeType,
                                const char *creatorName,
                                unsigned long sizeOfElement,
                                TQ3MetaHandler metaHandler);
```

Application-Defined Routines

```
typedef TQ3Status (*TQ3AttributeCopyInheritMethod) (
                                const void *fromInternalAttribute,
                                void *toInternalAttribute);

typedef TQ3Boolean TQ3AttributeInheritMethod;
```

Errors

`kQ3ErrorAttributeNotContained`	Attribute not contained in attribute set
`kQ3ErrorAttributeInvalidType`	Invalid type of attribute

Style Objects

Contents

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 6-10 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 6-11 for complete details on the meaning of subdivision method specifiers for each of the three subdivision methods.

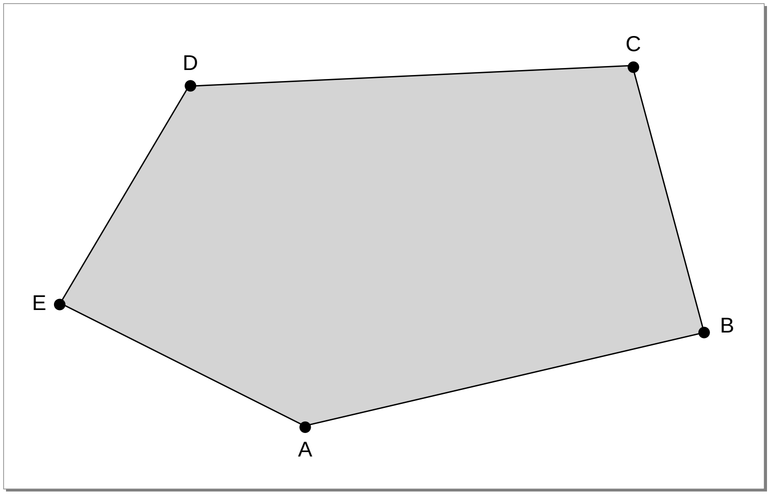
Orientation Styles

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



Note that 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.

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 6-33 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." ◆

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 on page 1-32 in the chapter "Introduction to QuickDraw 3D" for examples of submitting styles during retained mode rendering. See Listing 15-3 on page 15-15 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 its 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" on page 6-4 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" on page 6-5 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" on page 6-6 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" on page 6-6 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” on page 6-11 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" on page 6-7 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" on page 6-8 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" on page 6-9 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" on page 6-9 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 6-10 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" on page 6-9 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 6-10 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.

Summary of Style Objects

C Summary

Constants

```
typedef enum TQ3BackfacingStyle {
    kQ3BackfacingStyleBoth,
    kQ3BackfacingStyleRemove,
    kQ3BackfacingStyleFlip
} TQ3BackfacingStyle;

typedef enum TQ3InterpolationStyle {
    kQ3InterpolationStyleNone,
    kQ3InterpolationStyleVertex,
    kQ3InterpolationStylePixel
} TQ3InterpolationStyle;

typedef enum TQ3FillStyle {
    kQ3FillStyleFilled,
    kQ3FillStyleEdges,
    kQ3FillStylePoints
} TQ3FillStyle;

typedef enum TQ3SubdivisionMethod {
    kQ3SubdivisionMethodConstant,
    kQ3SubdivisionMethodWorldSpace,
    kQ3SubdivisionMethodScreenSpace
} TQ3SubdivisionMethod;
```

```
typedef enum TQ3OrientationStyle {
    kQ3OrientationStyleCounterClockwise,
    kQ3OrientationStyleClockwise
} TQ3OrientationStyle;

#define kQ3StyleTypeBackfacing           Q3_OBJECT_TYPE('b','c','k','f')
#define kQ3StyleTypeFill                 Q3_OBJECT_TYPE('f','i','s','t')
#define kQ3StyleTypeHighlight            Q3_OBJECT_TYPE('h','i','g','h')
#define kQ3StyleTypeInterpolation        Q3_OBJECT_TYPE('i','n','t','p')
#define kQ3StyleTypeOrientation          Q3_OBJECT_TYPE('o','f','d','r')
#define kQ3StyleTypePickID               Q3_OBJECT_TYPE('p','k','i','d')
#define kQ3StyleTypePickParts            Q3_OBJECT_TYPE('p','k','p','t')
#define kQ3StyleTypeReceiveShadows       Q3_OBJECT_TYPE('r','c','s','h')
#define kQ3StyleTypeSubdivision          Q3_OBJECT_TYPE('s','b','d','v')
```

Data Types

```
typedef struct TQ3SubdivisionStyleData {
    TQ3SubdivisionMethod        method;
    float                       c1;
    float                       c2;
} TQ3SubdivisionStyleData;
```

Style Objects Routines

Managing Styles

```
TQ3ObjectType Q3Style_GetType (TQ3StyleObject style);

TQ3Status Q3Style_Submit      (TQ3StyleObject style, TQ3ViewObject view);
```

Managing Backfacing Styles

```
TQ3StyleObject Q3BackfacingStyle_New (
                                TQ3BackfacingStyle backfacingStyle);
```

```
TQ3Status Q3BackfacingStyle_Submit (
                              TQ3BackfacingStyle backfacingStyle,
                              TQ3ViewObject view);

TQ3Status Q3BackfacingStyle_Get (
                              TQ3StyleObject backfacingObject,
                              TQ3BackfacingStyle *backfacingStyle);

TQ3Status Q3BackfacingStyle_Set (
                              TQ3StyleObject backfacingObject,
                              TQ3BackfacingStyle backfacingStyle);
```

Managing Interpolation Styles

```
TQ3StyleObject Q3InterpolationStyle_New (
                              TQ3InterpolationStyle interpolationStyle);

TQ3Status Q3InterpolationStyle_Submit (
                              TQ3InterpolationStyle interpolationStyle,
                              TQ3ViewObject view);

TQ3Status Q3InterpolationStyle_Get (
                              TQ3StyleObject interpolationObject,
                              TQ3InterpolationStyle *interpolationStyle);

TQ3Status Q3InterpolationStyle_Set (
                              TQ3StyleObject interpolationObject,
                              TQ3InterpolationStyle interpolationStyle);
```

Managing Fill Styles

```
TQ3StyleObject Q3FillStyle_New(TQ3FillStyle fillStyle);

TQ3Status Q3FillStyle_Submit  (TQ3FillStyle fillStyle, TQ3ViewObject view);

TQ3Status Q3FillStyle_Get     (TQ3StyleObject styleObject,
                               TQ3FillStyle *fillStyle);

TQ3Status Q3FillStyle_Set     (TQ3StyleObject styleObject,
                               TQ3FillStyle fillStyle);
```

Managing Highlight Styles

```
TQ3StyleObject Q3HighlightStyle_New (
                                TQ3AttributeSet highlightAttribute);
```

```
TQ3Status Q3HighlightStyle_Submit (
                                TQ3AttributeSet highlightAttribute,
                                TQ3ViewObject view);
```

```
TQ3Status Q3HighlightStyle_Get (TQ3StyleObject highlight,
                                TQ3AttributeSet *highlightAttribute);
```

```
TQ3Status Q3HighlightStyle_Set (TQ3StyleObject highlight,
                                TQ3AttributeSet highlightAttribute);
```

Managing Subdivision Styles

```
TQ3StyleObject Q3SubdivisionStyle_New (
                                const TQ3SubdivisionStyleData *data);
```

```
TQ3Status Q3SubdivisionStyle_Submit (
                                const TQ3SubdivisionStyleData *data,
                                TQ3ViewObject view);
```

```
TQ3Status Q3SubdivisionStyle_GetData (
                                TQ3StyleObject subdiv,
                                TQ3SubdivisionStyleData *data);
```

```
TQ3Status Q3SubdivisionStyle_SetData (
                                TQ3StyleObject subdiv,
                                const TQ3SubdivisionStyleData *data);
```

Managing Orientation Styles

```
TQ3StyleObject Q3OrientationStyle_New (
                                TQ3OrientationStyle frontFacingDirection);
```

```
TQ3Status Q3OrientationStyle_Submit (
                                TQ3OrientationStyle frontFacingDirection,
                                TQ3ViewObject view);
```

```
TQ3Status Q3OrientationStyle_Get (
                              TQ3StyleObject frontFacingDirectionObject,
                              TQ3OrientationStyle *frontFacingDirection);

TQ3Status Q3OrientationStyle_Set (
                              TQ3StyleObject frontFacingDirectionObject,
                              TQ3OrientationStyle frontFacingDirection);
```

Managing Shadow-Receiving Styles

```
TQ3StyleObject Q3ReceiveShadowsStyle_New (
                              TQ3Boolean receives);

TQ3Status Q3ReceiveShadowsStyle_Submit (
                              TQ3Boolean receives, TQ3ViewObject view);

TQ3Status Q3ReceiveShadowsStyle_Get (
                              TQ3StyleObject styleObject,
                              TQ3Boolean *receives);

TQ3Status Q3ReceiveShadowsStyle_Set (
                              TQ3StyleObject styleObject,
                              TQ3Boolean receives);
```

Managing Picking ID Styles

```
TQ3StyleObject Q3PickIDStyle_New (
                              unsigned long id);

TQ3Status Q3PickIDStyle_Submit(unsigned long id, TQ3ViewObject view);

TQ3Status Q3PickIDStyle_Get   (TQ3StyleObject pickIDObject,
                              unsigned long *id);

TQ3Status Q3PickIDStyle_Set   (TQ3StyleObject pickIDObject,
                              unsigned long id);
```

Managing Picking Parts Styles

```
TQ3StyleObject Q3PickPartsStyle_New (
                                TQ3PickParts parts);

TQ3Status Q3PickPartsStyle_Submit (
                                TQ3PickParts parts, TQ3ViewObject view);

TQ3Status Q3PickPartsStyle_Get(TQ3StyleObject pickPartsObject,
                                TQ3PickParts *parts);

TQ3Status Q3PickPartsStyle_Set(TQ3StyleObject pickPartsObject,
                                TQ3PickParts parts);
```

Transform Objects

Contents

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 7-16 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

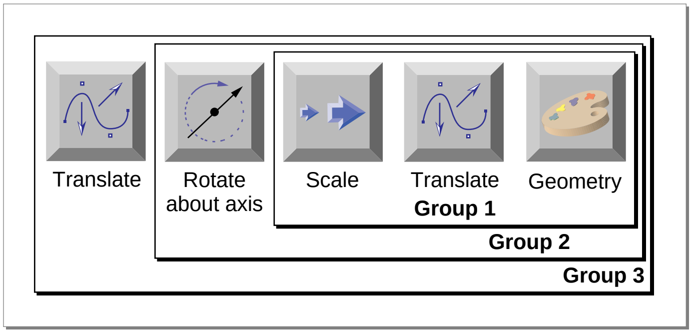
- rotate transforms
- rotate-about-point transforms
- rotate-about-axis transforms
- quaternion transforms

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



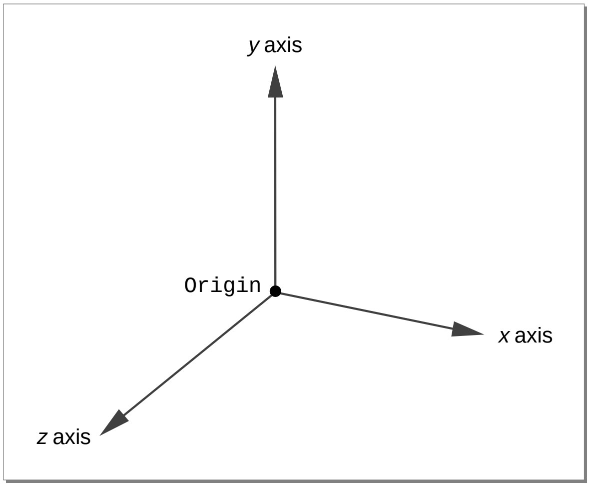
When this model is rendered, the transforms are applied to the geometric object in this order: scale, translate (group 1), rotate-about-axis (group 2), translate (group 3). Your application should add transforms to a group in the reverse order they are to be rendered. That is, in the example, you would 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 "QuickDraw 3D 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.

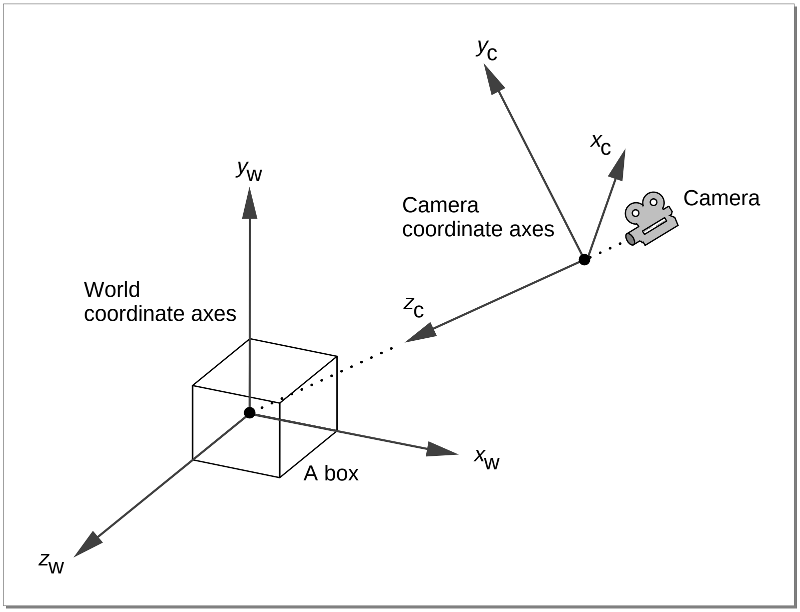
A world coordinate system defines the relative positions and sizes of geometric objects. When an object is rendered in a view, the view's camera specifies yet another coordinate system, the **camera coordinate system** (or **camera space**). A camera coordinate system is defined by the camera placement 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.

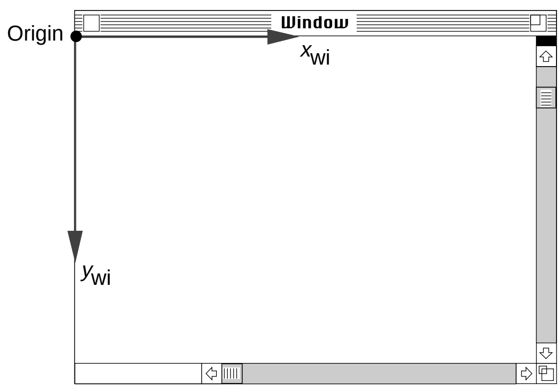
Figure 7-3 A camera coordinate system



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 on page 9-13, 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 on page 9-10). 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



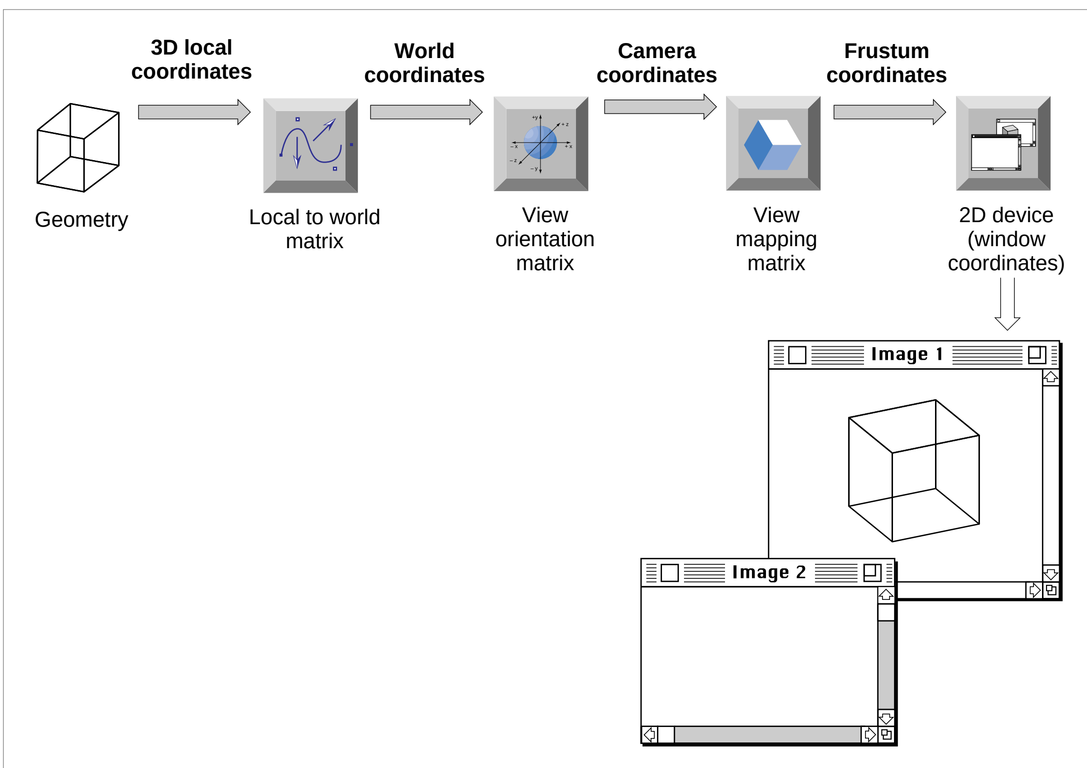
Note
A window coordinate system is sometimes called a **screen coordinate system** or a **draw context coordinate system,** and the space it defines is the **screen space** or **draw context space.** ◆

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

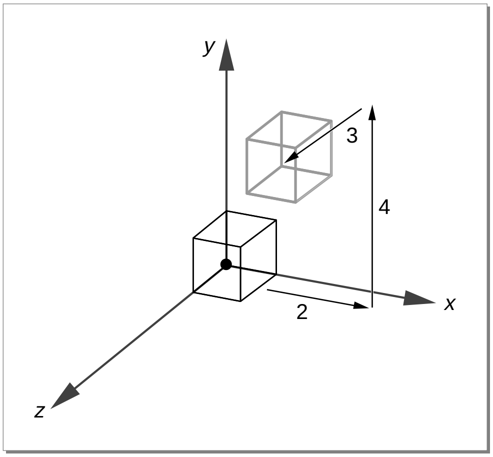
A **translate transform** translates an object along the *x*, *y*, and *z* axes by specified values. You specify the desired translation values using a vector. For example, to translate an object by 2 units along the positive *x* axis, by 4 units along the positive *y* axis, and by 3 units along the 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

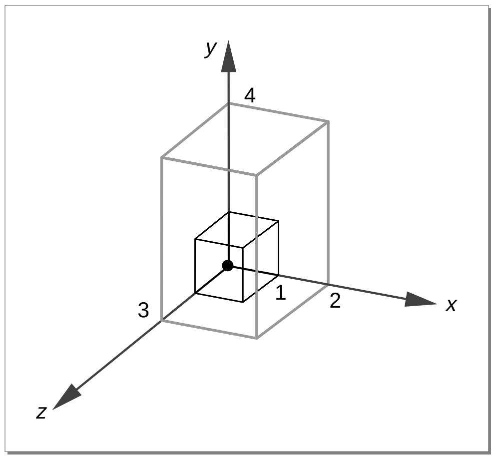
A **scale transform** scales an object along the *x*, *y*, and *z* axes by specified values. As with a translate transform, you specify the desired transform using a vector. For example, to scale an object by a factor of 2 along the positive *x* axis, by a factor of 4 along the positive *y* axis, and 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);
```

Figure 7-7 shows a unit cube before and after applying a scale transform.

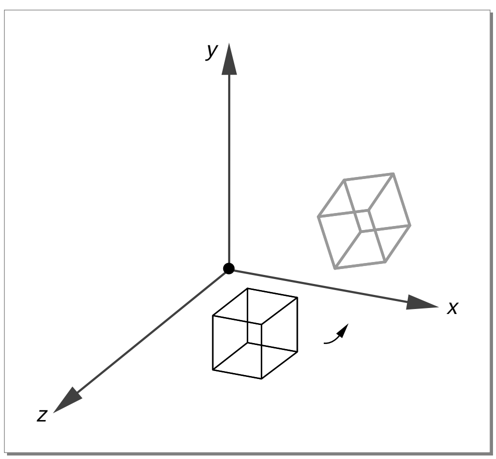
Figure 7-7 A scale transform



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 "QuickDraw 3D Mathematical Utilities" for details.) Figure 7-8 shows a unit cube before and after applying a rotate transform.

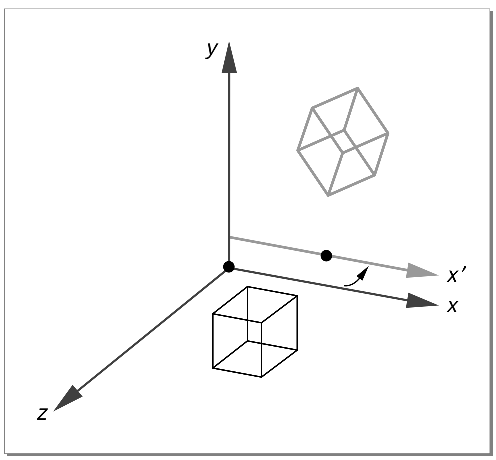
Figure 7-8 A rotate transform



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.

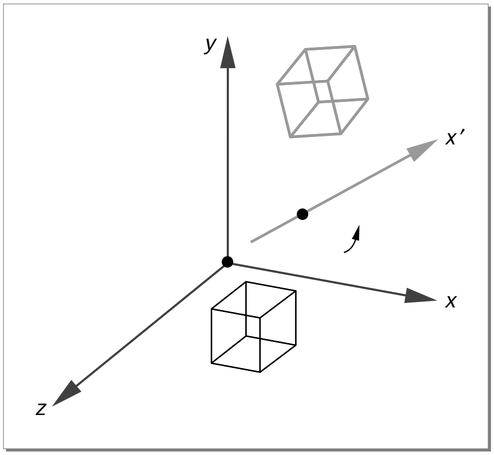
Figure 7-9 A rotate-about-point transform



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.

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 general 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 "QuickDraw 3D 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 "QuickDraw 3D 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 “QuickDraw 3D 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`.

Summary of Transform Objects

C Summary

Constants

```
#define kQ3TransformTypeMatrix               Q3_OBJECT_TYPE('m','t','r','x')
#define kQ3TransformTypeQuaternion           Q3_OBJECT_TYPE('q','t','r','n')
#define kQ3TransformTypeRotate               Q3_OBJECT_TYPE('r','o','t','t')
#define kQ3TransformTypeRotateAboutAxis      Q3_OBJECT_TYPE('r','t','a','a')
#define kQ3TransformTypeRotateAboutPoint     Q3_OBJECT_TYPE('r','t','a','p')
#define kQ3TransformTypeScale                Q3_OBJECT_TYPE('s','c','a','l')
#define kQ3TransformTypeTranslate            Q3_OBJECT_TYPE('t','r','n','s')
```

Data Types

```
typedef struct TQ3RotateTransformData {
    TQ3Axis                         axis;
    float                           radians;
} TQ3RotateTransformData;

typedef struct TQ3RotateAboutPointTransformData {
    TQ3Axis                         axis;
    float                           radians;
    TQ3Point3D                      about;
} TQ3RotateAboutPointTransformData;
```

```
typedef struct TQ3RotateAboutAxisTransformData {
    TQ3Point3D                          origin;
    TQ3Vector3D                         orientation;
    float                               radians;
} TQ3RotateAboutAxisTransformData;
```

Transform Objects Routines

Managing Transforms

```
TQ3ObjectType Q3Transform_GetType (
                               TQ3TransformObject transform);

TQ3Matrix4x4 *Q3Transform_GetMatrix (
                               TQ3TransformObject transform,
                               TQ3Matrix4x4 *matrix);

TQ3Status Q3Transform_Submit  (TQ3TransformObject transform,
                               TQ3ViewObject view);
```

Creating and Manipulating Matrix Transforms

```
TQ3TransformObject Q3MatrixTransform_New (
                               const TQ3Matrix4x4 *matrix);

TQ3Status Q3MatrixTransform_Submit (
                               const TQ3Matrix4x4 *matrix,
                               TQ3ViewObject view);

TQ3Status Q3MatrixTransform_Get (
                               TQ3TransformObject transform,
                               TQ3Matrix4x4 *matrix);

TQ3Status Q3MatrixTransform_Set (
                               TQ3TransformObject transform,
                               const TQ3Matrix4x4 *matrix);
```

Creating and Manipulating Rotate Transforms

```
TQ3TransformObject Q3RotateTransform_New (
                              const TQ3RotateTransformData *data);

TQ3Status Q3RotateTransform_Submit (
                              const TQ3RotateTransformData *data,
                              TQ3ViewObject view);

TQ3Status Q3RotateTransform_GetData (
                              TQ3TransformObject transform,
                              TQ3RotateTransformData *data);

TQ3Status Q3RotateTransform_SetData (
                              TQ3TransformObject transform,
                              const TQ3RotateTransformData *data);

TQ3Status Q3RotateTransform_GetAxis (
                              TQ3TransformObject transform,
                              TQ3Axis *axis);

TQ3Status Q3RotateTransform_SetAxis (
                              TQ3TransformObject transform,
                              TQ3Axis axis);

TQ3Status Q3RotateTransform_GetAngle (
                              TQ3TransformObject transform,
                              float *radians);

TQ3Status Q3RotateTransform_SetAngle (
                              TQ3TransformObject transform,
                              float radians);
```

Creating and Manipulating Rotate-About-Point Transforms

```
TQ3TransformObject Q3RotateAboutPointTransform_New (
                              const TQ3RotateAboutPointTransformData *data);

TQ3Status Q3RotateAboutPointTransform_Submit (
                              const TQ3RotateAboutPointTransformData *data,
                              TQ3ViewObject view);
```

```
TQ3Status Q3RotateAboutPointTransform_GetData (
                              TQ3TransformObject transform,
                              TQ3RotateAboutPointTransformData *data);

TQ3Status Q3RotateAboutPointTransform_SetData (
                              TQ3TransformObject transform,
                              const TQ3RotateAboutPointTransformData *data);

TQ3Status Q3RotateAboutPointTransform_GetAxis (
                              TQ3TransformObject transform,
                              TQ3Axis *axis);

TQ3Status Q3RotateAboutPointTransform_SetAxis (
                              TQ3TransformObject transform,
                              TQ3Axis axis);

TQ3Status Q3RotateAboutPointTransform_GetAngle (
                              TQ3TransformObject transform,
                              float *radians);

TQ3Status Q3RotateAboutPointTransform_SetAngle (
                              TQ3TransformObject transform,
                              float radians);

TQ3Status Q3RotateAboutPointTransform_GetAboutPoint (
                              TQ3TransformObject transform,
                              TQ3Point3D *about);

TQ3Status Q3RotateAboutPointTransform_SetAboutPoint (
                              TQ3TransformObject transform,
                              const TQ3Point3D *about);
```

Creating and Manipulating Rotate-About-Axis Transforms

```
TQ3TransformObject Q3RotateAboutAxisTransform_New (
                              const TQ3RotateAboutAxisTransformData *data);

TQ3Status Q3RotateAboutAxisTransform_Submit (
                              const TQ3RotateAboutAxisTransformData *data,
                              TQ3ViewObject view);
```

```
TQ3Status Q3RotateAboutAxisTransform_GetData (
                                TQ3TransformObject transform,
                                TQ3RotateAboutAxisTransformData *data);

TQ3Status Q3RotateAboutAxisTransform_SetData (
                                TQ3TransformObject transform,
                                const TQ3RotateAboutAxisTransformData *data);

TQ3Status Q3RotateAboutAxisTransform_GetOrigin (
                                TQ3TransformObject transform,
                                TQ3Point3D *origin);

TQ3Status Q3RotateAboutAxisTransform_SetOrigin (
                                TQ3TransformObject transform,
                                const TQ3Point3D *origin);

TQ3Status Q3RotateAboutAxisTransform_GetOrientation (
                                TQ3TransformObject transform,
                                TQ3Vector3D *axis);

TQ3Status Q3RotateAboutAxisTransform_SetOrientation (
                                TQ3TransformObject transform,
                                const TQ3Vector3D *axis);

TQ3Status Q3RotateAboutAxisTransform_GetAngle (
                                TQ3TransformObject transform,
                                float *radians);

TQ3Status Q3RotateAboutAxisTransform_SetAngle (
                                TQ3TransformObject transform,
                                float radians);
```

Creating and Manipulating Scale Transforms

```
TQ3TransformObject Q3ScaleTransform_New (
                                const TQ3Vector3D *scale);

TQ3Status Q3ScaleTransform_Submit (
                                TQ3Vector3D *scale, TQ3ViewObject view);
```

```
TQ3Status Q3ScaleTransform_Get (TQ3TransformObject transform,
                               TQ3Vector3D *scale);

TQ3Status Q3ScaleTransform_Set (TQ3TransformObject transform,
                               const TQ3Vector3D *scale);
```

Creating and Manipulating Translate Transforms

```
TQ3TransformObject Q3TranslateTransform_New (
                               const TQ3Vector3D *translate);

TQ3Status Q3TranslateTransform_Submit (
                               const TQ3Vector3D *translate,
                               TQ3ViewObject view);

TQ3Status Q3TranslateTransform_Get (
                               TQ3TransformObject transform,
                               TQ3Vector3D *translate);

TQ3Status Q3TranslateTransform_Set (
                               TQ3TransformObject transform,
                               const TQ3Vector3D *translate);
```

Creating and Manipulating Quaternion Transforms

```
TQ3TransformObject Q3QuaternionTransform_New (
                               const TQ3Quaternion *quaternion);

TQ3Status Q3QuaternionTransform_Submit (
                               const TQ3Quaternion *quaternion,
                               TQ3ViewObject view);

TQ3Status Q3QuaternionTransform_Get (
                               TQ3TransformObject transform,
                               TQ3Quaternion *quaternion);

TQ3Status Q3QuaternionTransform_Set (
                               TQ3TransformObject transform,
                               const TQ3Quaternion *quaternion);
```

Errors

`kQ3ErrorScaleOfZero`

Light Objects

Contents

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 8-9 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

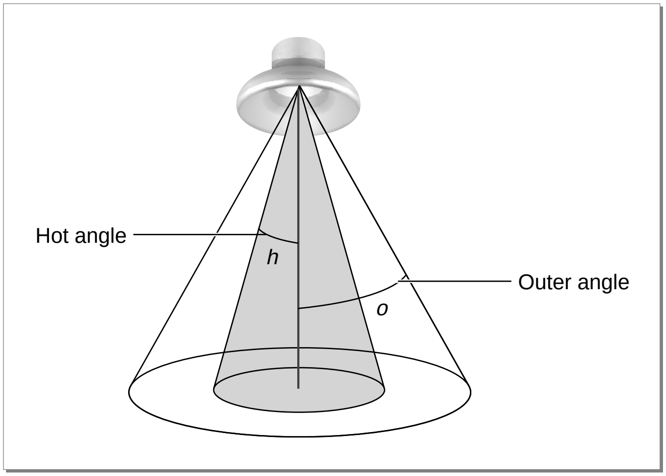
A **point light** is a light source that emits rays of light in all directions from a specific location. The illumination that a point light contributes to a surface depends on the basic properties of the light source (its intensity and color) together with the orientation of the surface and its distance 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” on page 8-10 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.

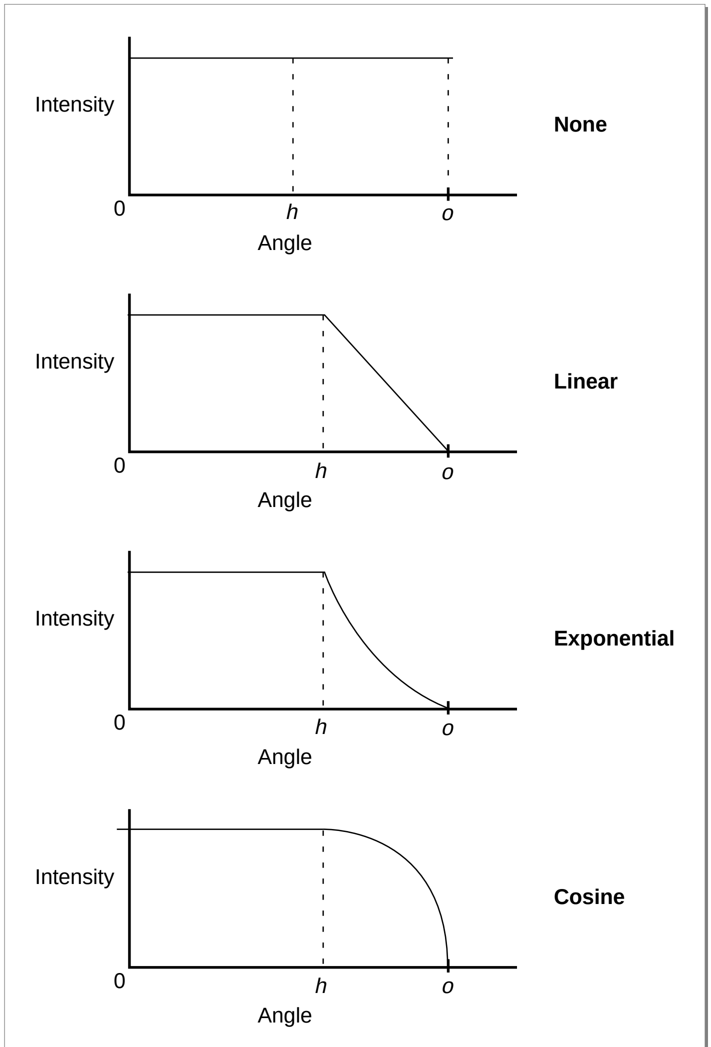
Figure 8-1 A spot light



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.

Figure 8-2 Fall-off algorithms



See "Light Fall-Off Values" on page 8-10 for a description of the constants you can use to specify a spot light's fall-off value.

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" on page 8-10 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" on page 8-10 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" on page 8-10 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.

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.

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" on page 8-11 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" on page 8-11 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" on page 8-12 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" on page 8-10 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" on page 8-10 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" on page 8-13 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" on page 8-10 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" on page 8-10 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" on page 8-10 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" on page 8-10 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" on page 8-13 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.

Summary of Light Objects

C Summary

Constants

Light Types

```
#define kQ3LightTypeAmbient                Q3_OBJECT_TYPE('a','m','b','n')
#define kQ3LightTypeDirectional            Q3_OBJECT_TYPE('d','r','c','t')
#define kQ3LightTypePoint                  Q3_OBJECT_TYPE('p','n','t','l')
#define kQ3LightTypeSpot                   Q3_OBJECT_TYPE('s','p','o','t')
```

Light Attenuation Values

```
typedef enum TQ3AttenuationType {
    kQ3AttenuationTypeNone,
    kQ3AttenuationTypeInverseDistance,
    kQ3AttenuationTypeInverseDistanceSquared
} TQ3AttenuationType;
```

Light Fall-Off Values

```
typedef enum TQ3FallOffType {
    kQ3FallOffTypeNone,
    kQ3FallOffTypeLinear,
    kQ3FallOffTypeExponential,
    kQ3FallOffTypeCosine
} TQ3FallOffType;
```

Data Types

Light Data Structure

```
typedef struct TQ3LightData {
    TQ3Boolean                      isOn;
    float                           brightness;
    TQ3ColorRGB                     color;
} TQ3LightData;
```

Directional Light Data Structure

```
typedef struct TQ3DirectionalLightData {
    TQ3LightData                    lightData;
    TQ3Boolean                      castsShadows;
    TQ3Vector3D                     direction;
} TQ3DirectionalLightData;
```

Point Light Data Structure

```
typedef struct TQ3PointLightData {
    TQ3LightData                    lightData;
    TQ3Boolean                      castsShadows;
    TQ3AttenuationType              attenuation;
    TQ3Point3D                      location;
} TQ3PointLightData;
```

Spot Light Data Structure

```
typedef struct TQ3SpotLightData {
    TQ3LightData                    lightData;
    TQ3Boolean                      castsShadows;
    TQ3AttenuationType              attenuation;
    TQ3Point3D                      location;
    TQ3Vector3D                     direction;
    float                           hotAngle;
```

```
    float                               outerAngle;
    TQ3FallOffType                      fallOff;
} TQ3SpotLightData;
```

Light Objects Routines

Managing Lights

```
TQ3ObjectType Q3Light_GetType (TQ3LightObject light);

TQ3Status Q3Light_GetState    (TQ3LightObject light, TQ3Boolean *isOn);

TQ3Status Q3Light_SetState    (TQ3LightObject light, TQ3Boolean isOn);

TQ3Status Q3Light_GetBrightness (
                               TQ3LightObject light, float *brightness);

TQ3Status Q3Light_SetBrightness (
                               TQ3LightObject light, float brightness);

TQ3Status Q3Light_GetColor    (TQ3LightObject light, TQ3ColorRGB *color);

TQ3Status Q3Light_SetColor    (TQ3LightObject light,
                               const TQ3ColorRGB *color);

TQ3Status Q3Light_GetData     (TQ3LightObject light,
                               TQ3LightData *lightData);

TQ3Status Q3Light_SetData     (TQ3LightObject light,
                               const TQ3LightData *lightData);
```

Managing Ambient Light

```
TQ3LightObject Q3AmbientLight_New (
                               const TQ3LightData *lightData);

TQ3Status Q3AmbientLight_GetData (
                               TQ3LightObject light,
                               TQ3LightData *lightData);
```

```
TQ3Status Q3AmbientLight_SetData (
                              TQ3LightObject light,
                              const TQ3LightData *lightData);
```

Managing Directional Lights

```
TQ3LightObject Q3DirectionalLight_New (
                              const TQ3DirectionalLightData
                              *directionalLightData);

TQ3Status Q3DirectionalLight_GetCastShadowsState (
                              TQ3LightObject light,
                              TQ3Boolean *castsShadows);

TQ3Status Q3DirectionalLight_SetCastShadowsState (
                              TQ3LightObject light,
                              TQ3Boolean castsShadows);

TQ3Status Q3DirectionalLight_GetDirection (
                              TQ3LightObject light,
                              TQ3Vector3D *direction);

TQ3Status Q3DirectionalLight_SetDirection (
                              TQ3LightObject light,
                              const TQ3Vector3D *direction);

TQ3Status Q3DirectionalLight_GetData (
                              TQ3LightObject light,
                              TQ3DirectionalLightData
                              *directionalLightData);

TQ3Status Q3DirectionalLight_SetData (
                              TQ3LightObject light,
                              const TQ3DirectionalLightData
                              *directionalLightData);
```

Managing Point Lights

```
TQ3LightObject Q3PointLight_New(const TQ3PointLightData *pointLightData);

TQ3Status Q3PointLight_GetCastShadowsState (
                              TQ3LightObject light,
                              TQ3Boolean *castsShadows);

TQ3Status Q3PointLight_SetCastShadowsState (
                              TQ3LightObject light,
                              TQ3Boolean castsShadows);

TQ3Status Q3PointLight_GetAttenuation (
                              TQ3LightObject light,
                              TQ3AttenuationType *attenuation);

TQ3Status Q3PointLight_SetAttenuation (
                              TQ3LightObject light,
                              TQ3AttenuationType attenuation);

TQ3Status Q3PointLight_GetLocation (
                              TQ3LightObject light,
                              TQ3Point3D *location);

TQ3Status Q3PointLight_SetLocation (
                              TQ3LightObject light,
                              const TQ3Point3D *location);

TQ3Status Q3PointLight_GetData(TQ3LightObject light,
                              TQ3PointLightData *pointLightData);

TQ3Status Q3PointLight_SetData(TQ3LightObject light,
                              const TQ3PointLightData *pointLightData);
```

Managing Spot Lights

```
TQ3LightObject Q3SpotLight_New(const TQ3SpotLightData *spotLightData);

TQ3Status Q3SpotLight_GetCastShadowsState (
                              TQ3LightObject light,
                              TQ3Boolean *castsShadows);
```

```
TQ3Status Q3SpotLight_SetCastShadowsState (
                              TQ3LightObject light,
                              TQ3Boolean castsShadows);

TQ3Status Q3SpotLight_GetAttenuation (
                              TQ3LightObject light,
                              TQ3AttenuationType *attenuation);

TQ3Status Q3SpotLight_SetAttenuation (
                              TQ3LightObject light,
                              TQ3AttenuationType attenuation);

TQ3Status Q3SpotLight_GetLocation (
                              TQ3LightObject light,
                              TQ3Point3D *location);

TQ3Status Q3SpotLight_SetLocation (
                              TQ3LightObject light,
                              const TQ3Point3D *location);

TQ3Status Q3SpotLight_GetDirection (
                              TQ3LightObject light,
                              TQ3Vector3D *direction);

TQ3Status Q3SpotLight_SetDirection (
                              TQ3LightObject light,
                              const TQ3Vector3D *direction);

TQ3Status Q3SpotLight_GetHotAngle (
                              TQ3LightObject light, float *hotAngle);

TQ3Status Q3SpotLight_SetHotAngle (
                              TQ3LightObject light, float hotAngle);

TQ3Status Q3SpotLight_GetOuterAngle (
                              TQ3LightObject light, float *outerAngle);

TQ3Status Q3SpotLight_SetOuterAngle (
                              TQ3LightObject light, float outerAngle);
```

```
TQ3Status Q3SpotLight_GetFallOff (
                               TQ3LightObject light,
                               TQ3FallOffType *fallOff);

TQ3Status Q3SpotLight_SetFallOff (
                               TQ3LightObject light,
                               TQ3FallOffType fallOff);

TQ3Status Q3SpotLight_GetData (TQ3LightObject light,
                               TQ3SpotLightData *spotLightData);

TQ3Status Q3SpotLight_SetData (TQ3LightObject light,
                               const TQ3SpotLightData *spotLightData);
```

Notices

`kQ3NoticeInvalidAttenuationTypeUsingInternalDefaults`	Attenuation type is invalid

Camera Objects

Contents

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.

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 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 9-17 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;
```

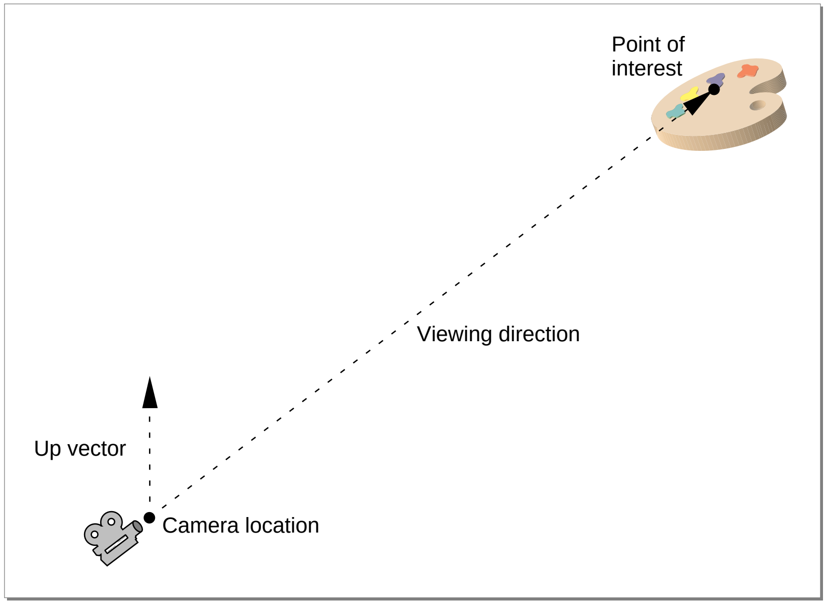
These fields specify the location and orientation of the camera, the visible range of interest, and the camera's view port and projection method. The following sections explain these concepts in greater detail.

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.

Note
Because a camera defines a point of view onto a model, the camera location is also called the *eye point*. ◆

Figure 9-1 A camera's placement



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.

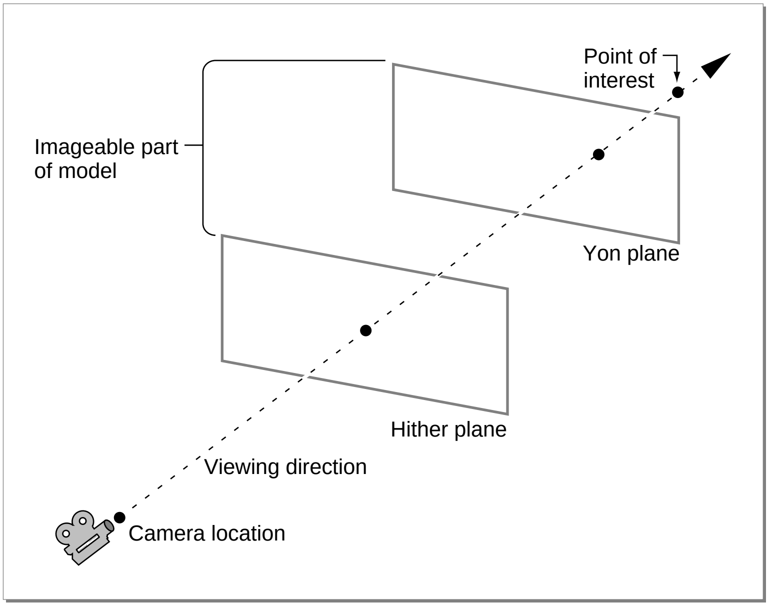
```
typedef struct TQ3CameraPlacement {
    TQ3Point3D                          cameraLocation;
    TQ3Point3D                          pointOfInterest;
    TQ3Vector3D                         upVector;
} TQ3CameraPlacement;
```

See "Camera Placement Structure" on page 9-18 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. 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" on page 9-11 for details on how to specify a parallel projection.

Figure 9-3 A parallel projection of an object

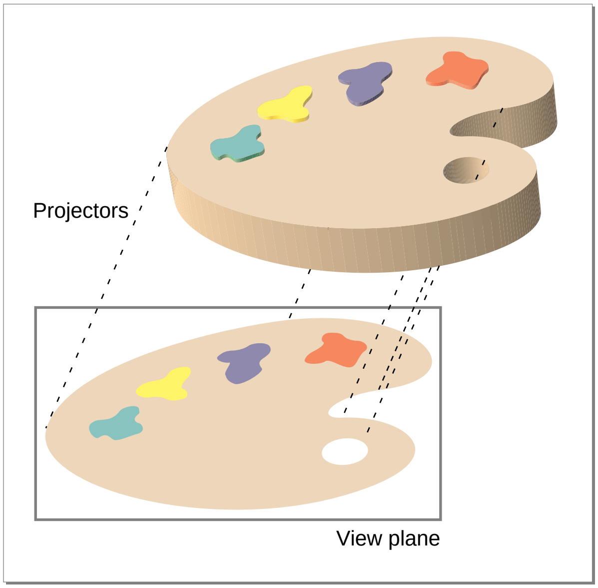
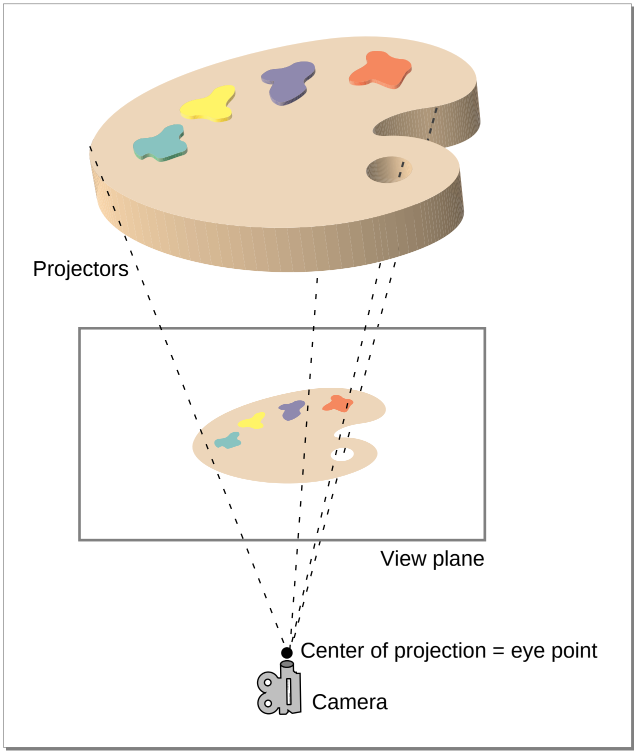


Figure 9-4 illustrates a perspective projection of an object. As you can see, the location of the view plane is very important in a perspective projection. When the view plane is close to the camera, the projectors are close together and the image they create is small. Conversely, when the view plane 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.**

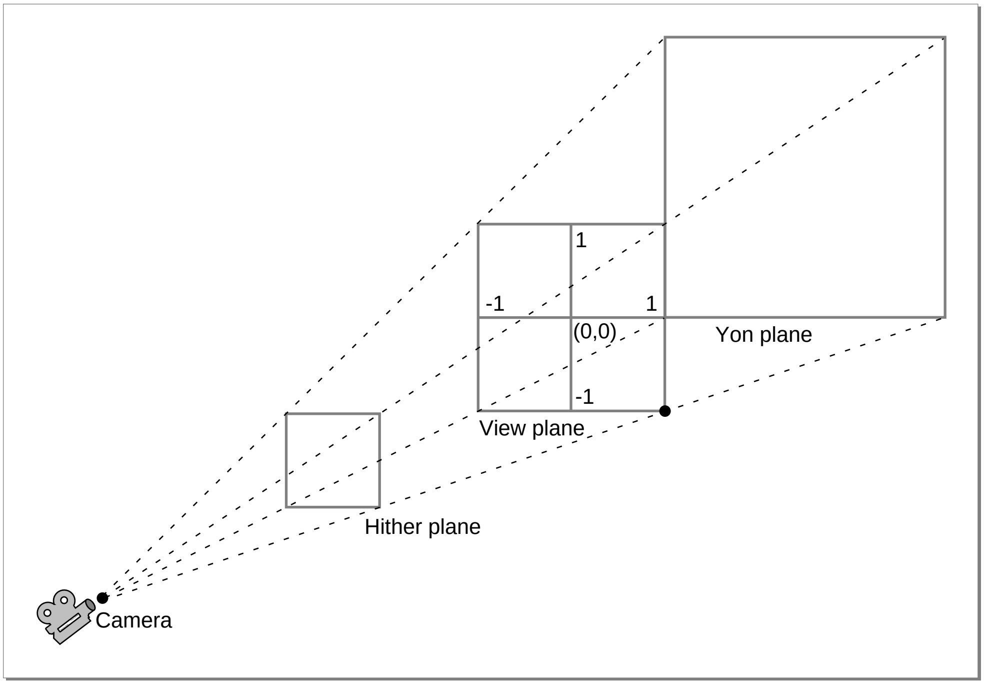
Figure 9-4 A perspective projection of an object



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" on page 9-13 and "Aspect Ratio Cameras" on page 9-15 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 that is bounded by the view volume (either a view box, for parallel projections, or a view frustum, for perspective projections). Figure 9-5 shows the default camera view port for a perspective camera.

Figure 9-5 The default camera view port



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.

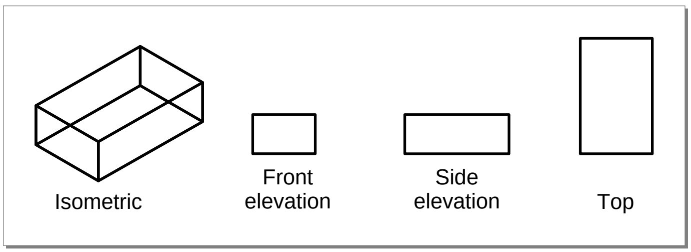
Note
The image displayed in a draw context is not necessarily the image drawn on the view port. The view port image is scaled to fit into the draw context pane and then clipped with the draw context mask. See the chapter "Draw Context Objects" for information about draw context panes and masks, and for further details on the relationship between a view port and a draw context. ◆

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.

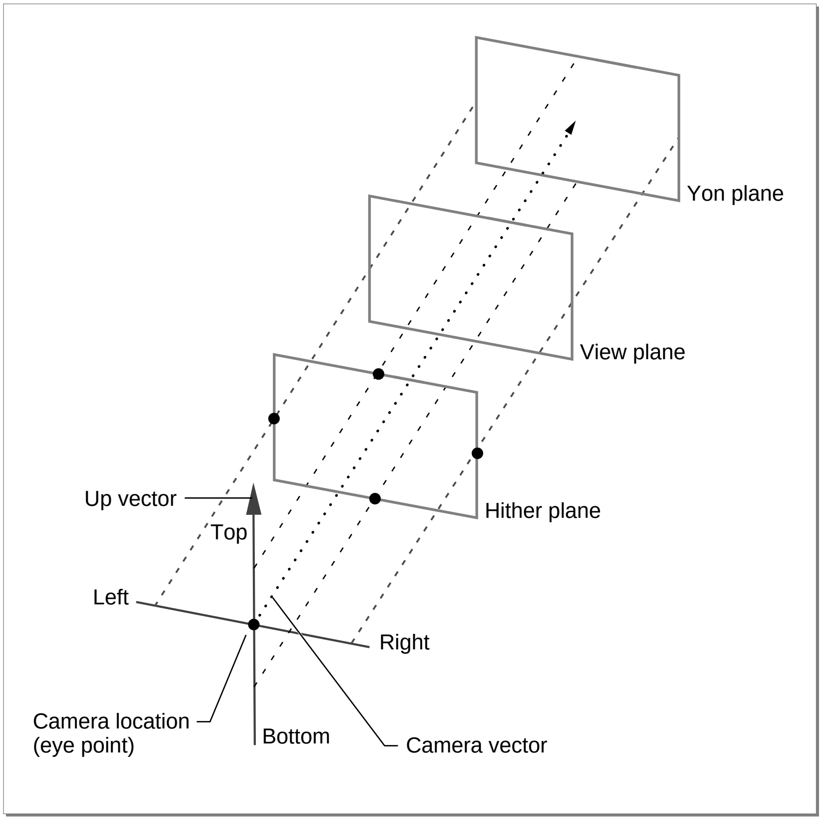
The two most common types of orthographic projection are isometric projection and elevation projection. An **isometric projection** is an orthographic projection in which the viewing direction makes equal angles with each of the three principal axes of an object. An **elevation projection** is an orthographic 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.

Figure 9-7 An orthographic camera



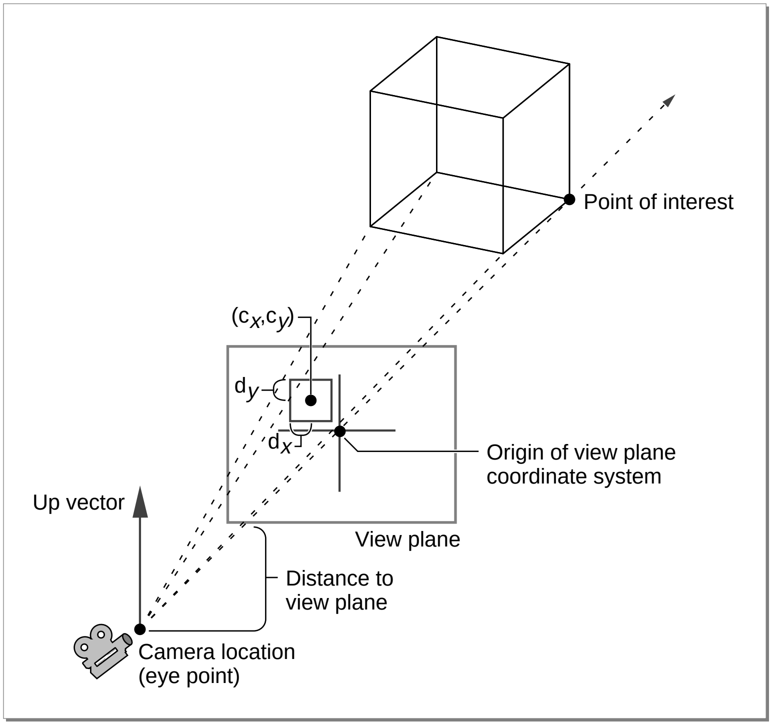
See "Orthographic Camera Data Structure" on page 9-20 for details on the data you need to provide to define an orthographic camera. See "Managing Orthographic Cameras," beginning on page 9-29 for a description of the routines you can use to create and manipulate orthographic cameras.

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

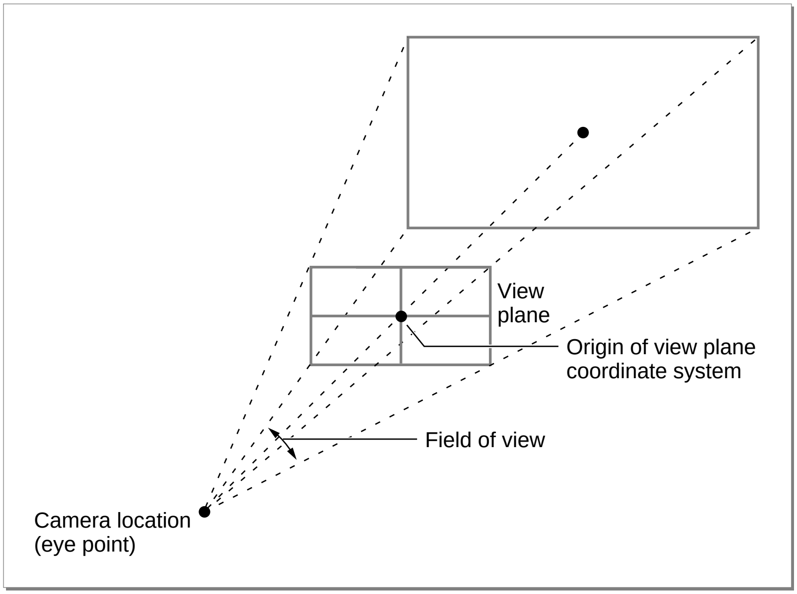


See "View Plane Camera Data Structure" on page 9-20 for complete details on the data you need to provide to define a view plane camera. See "Managing View Plane Cameras," beginning on page 9-35 for a description of the routines you can use to create and manipulate view plane cameras.

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

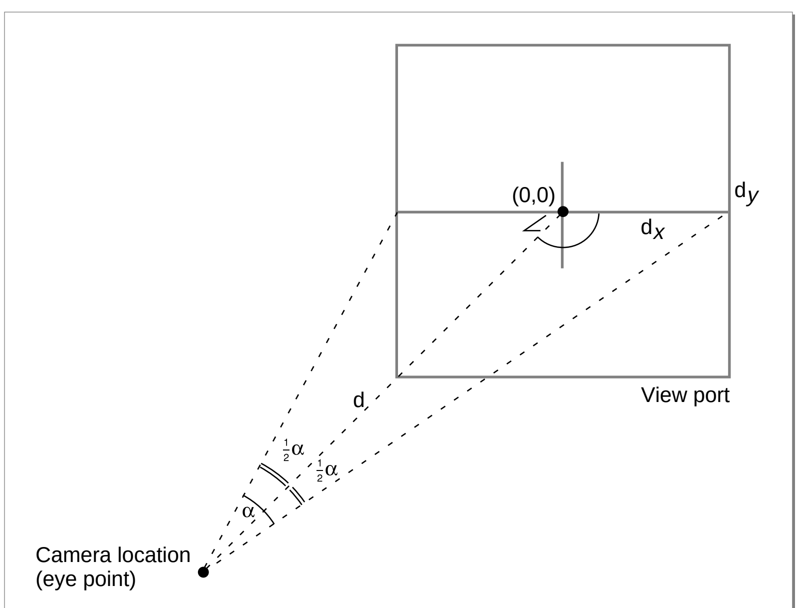


The orientation of the field of view is determined by the specified aspect ratio. If the aspect ratio is greater than 1.0, the field of view is vertical. If the aspect ratio is less than 1.0, the field of view is horizontal. In general, to avoid distortion, the aspect ratio should be the same as the 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” on page 9-21 for more details on the data you need to provide to define an aspect ratio camera. See “Managing Aspect Ratio Cameras,” beginning on page 9-42 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 on page 1-28 and Listing 1-9 on page 1-30 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. A camera's up-vector specifies the orientation of the camera. This vector 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" on page 9-19 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" on page 9-20 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" on page 9-20 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” on page 9-21 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 specified 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.

Summary of Camera Objects

C Summary

Constants

Camera Types

```
#define kQ3CameraTypeOrthographic          Q3_OBJECT_TYPE('o','r','t','h')
#define kQ3CameraTypeViewAngleAspect       Q3_OBJECT_TYPE('v','a','n','a')
#define kQ3CameraTypeViewPlane             Q3_OBJECT_TYPE('v','w','p','l')
```

Data Types

Camera Placement Structure

```
typedef struct TQ3CameraPlacement {
   TQ3Point3D                      cameraLocation;
   TQ3Point3D                      pointOfInterest;
   TQ3Vector3D                     upVector;
} TQ3CameraPlacement;
```

Camera Range Structure

```
typedef struct TQ3CameraRange {
   float                           hither;
   float                           yon;
} TQ3CameraRange;
```

Camera View Port

```
typedef struct TQ3CameraViewPort {
    TQ3Point2D                  origin;
    float                       width;
    float                       height;
} TQ3CameraViewPort;
```

Camera Data Structure

```
typedef struct TQ3CameraData {
    TQ3CameraPlacement          placement;
    TQ3CameraRange              range;
    TQ3CameraViewPort           viewPort;
} TQ3CameraData;
```

Orthographic Camera Data Structure

```
typedef struct TQ3OrthographicCameraData {
    TQ3CameraData               cameraData;
    float                       left;
    float                       top;
    float                       right;
    float                       bottom;
} TQ3OrthographicCameraData;
```

View Plane Camera Data Structure

```
typedef struct TQ3ViewPlaneCameraData {
    TQ3CameraData               cameraData;
    float                       viewPlane;
    float                       halfWidthAtViewPlane;
    float                       halfHeightAtViewPlane;
    float                       centerXOnViewPlane;
    float                       centerYOnViewPlane;
} TQ3ViewPlaneCameraData;
```

Aspect Ratio Camera Data Structure

```
typedef struct TQ3ViewAngleAspectCameraData {
    TQ3CameraData                   cameraData;
    float                           fov;
    float                           aspectRatioXToY;
} TQ3ViewAngleAspectCameraData;
```

Camera Objects Routines

Managing Cameras

```
TQ3ObjectType Q3Camera_GetType(TQ3CameraObject camera);

TQ3Status Q3Camera_GetData    (TQ3CameraObject camera,
                               TQ3CameraData *cameraData);

TQ3Status Q3Camera_SetData    (TQ3CameraObject camera,
                               const TQ3CameraData *cameraData);

TQ3Status Q3Camera_GetPlacement (
                               TQ3CameraObject camera,
                               TQ3CameraPlacement *placement);

TQ3Status Q3Camera_SetPlacement (
                               TQ3CameraObject camera,
                               const TQ3CameraPlacement *placement);

TQ3Status Q3Camera_GetRange   (TQ3CameraObject camera,
                               TQ3CameraRange *range);

TQ3Status Q3Camera_SetRange   (TQ3CameraObject camera,
                               const TQ3CameraRange *range);

TQ3Status Q3Camera_GetViewPort(TQ3CameraObject camera,
                               TQ3CameraViewPort *viewPort);

TQ3Status Q3Camera_SetViewPort(TQ3CameraObject camera,
                               const TQ3CameraViewPort *viewPort);
```

```
TQ3Status Q3Camera_GetWorldToView (
                              TQ3CameraObject camera,
                              TQ3Matrix4x4 *worldToView);

TQ3Status Q3Camera_GetViewToFrustum (
                              TQ3CameraObject camera,
                              TQ3Matrix4x4 *viewToFrustum);

TQ3Status Q3Camera_GetWorldToFrustum (
                              TQ3CameraObject camera,
                              TQ3Matrix4x4 *worldToFrustum);
```

Managing Orthographic Cameras

```
TQ3CameraObject Q3OrthographicCamera_New (
                              const TQ3OrthographicCameraData
                              *orthographicData);

TQ3Status Q3OrthographicCamera_GetData (
                              TQ3CameraObject camera,
                              TQ3OrthographicCameraData *cameraData);

TQ3Status Q3OrthographicCamera_SetData (
                              TQ3CameraObject camera,
                              const TQ3OrthographicCameraData *cameraData);

TQ3Status Q3OrthographicCamera_GetLeft (
                              TQ3CameraObject camera, float *left);

TQ3Status Q3OrthographicCamera_SetLeft (
                              TQ3CameraObject camera, float left);

TQ3Status Q3OrthographicCamera_GetTop (
                              TQ3CameraObject camera, float *top);

TQ3Status Q3OrthographicCamera_SetTop (
                              TQ3CameraObject camera, float top);

TQ3Status Q3OrthographicCamera_GetRight (
                              TQ3CameraObject camera, float *right);
```

```
TQ3Status Q3OrthographicCamera_SetRight (
                              TQ3CameraObject camera, float right);

TQ3Status Q3OrthographicCamera_GetBottom (
                              TQ3CameraObject camera, float *bottom);

TQ3Status Q3OrthographicCamera_SetBottom (
                              TQ3CameraObject camera, float bottom);
```

Managing View Plane Cameras

```
TQ3CameraObject Q3ViewPlaneCamera_New (
                              const TQ3ViewPlaneCameraData *cameraData);

TQ3Status Q3ViewPlaneCamera_GetData (
                              TQ3CameraObject camera,
                              TQ3ViewPlaneCameraData *cameraData);

TQ3Status Q3ViewPlaneCamera_SetData (
                              TQ3CameraObject camera,
                              const TQ3ViewPlaneCameraData *cameraData);

TQ3Status Q3ViewPlaneCamera_GetViewPlane (
                              TQ3CameraObject camera, float *viewPlane);

TQ3Status Q3ViewPlaneCamera_SetViewPlane (
                              TQ3CameraObject camera, float viewPlane);

TQ3Status Q3ViewPlaneCamera_GetHalfWidth (
                              TQ3CameraObject camera,
                              float *halfWidthAtViewPlane);

TQ3Status Q3ViewPlaneCamera_SetHalfWidth (
                              TQ3CameraObject camera,
                              float halfWidthAtViewPlane);

TQ3Status Q3ViewPlaneCamera_GetHalfHeight (
                              TQ3CameraObject camera,
                              float *halfHeightAtViewPlane);
```

```
TQ3Status Q3ViewPlaneCamera_SetHalfHeight (
                                TQ3CameraObject camera,
                                float halfHeightAtViewPlane);

TQ3Status Q3ViewPlaneCamera_GetCenterX (
                                TQ3CameraObject camera,
                                float *centerXOnViewPlane);

TQ3Status Q3ViewPlaneCamera_SetCenterX (
                                TQ3CameraObject camera,
                                float centerXOnViewPlane);

TQ3Status Q3ViewPlaneCamera_GetCenterY (
                                TQ3CameraObject camera,
                                float *centerYOnViewPlane);

TQ3Status Q3ViewPlaneCamera_SetCenterY (
                                TQ3CameraObject camera,
                                float centerYOnViewPlane);
```

Managing Aspect Ratio Cameras

```
TQ3CameraObject Q3ViewAngleAspectCamera_New (
                                const TQ3ViewAngleAspectCameraData
                                *cameraData);

TQ3Status Q3ViewAngleAspectCamera_GetData (
                                TQ3CameraObject camera,
                                TQ3ViewAngleAspectCameraData *cameraData);

TQ3Status Q3ViewAngleAspectCamera_SetData (
                                TQ3CameraObject camera,
                                const TQ3ViewAngleAspectCameraData
                                *cameraData);

TQ3Status Q3ViewAngleAspectCamera_GetFOV (
                                TQ3CameraObject camera, float *fov);

TQ3Status Q3ViewAngleAspectCamera_SetFOV (
                                TQ3CameraObject camera, float fov);
```

```
TQ3Status Q3ViewAngleAspectCamera_GetAspectRatio (
                              TQ3CameraObject camera,
                              float *aspectRatioXToY);

TQ3Status Q3ViewAngleAspectCamera_SetAspectRatio (
                              TQ3CameraObject camera,
                              float aspectRatioXToY);
```

Errors

`kQ3ErrorInvalidCameraValues`	Some camera values are invalid

Group Objects

Contents

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 10-11 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 10-8 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;
```

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.

Note
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 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;
```

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" on page 10-11 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" on page 10-11 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
```

Summary of Group Objects

C Summary

Constants

```
#define kQ3GroupTypeDisplay                   Q3_OBJECT_TYPE('d','s','p','g')
#define kQ3GroupTypeInfo                      Q3_OBJECT_TYPE('i','n','f','o')
#define kQ3GroupTypeLight                     Q3_OBJECT_TYPE('l','g','h','g')

#define kQ3DisplayGroupTypeIOProxy            Q3_OBJECT_TYPE('i','o','p','x')
#define kQ3DisplayGroupTypeOrdered            Q3_OBJECT_TYPE('o','r','d','g')

typedef enum TQ3DisplayGroupStateMasks {
    kQ3DisplayGroupStateNone                         = 0,
    kQ3DisplayGroupStateMaskIsDrawn                  = 1 << 0,
    kQ3DisplayGroupStateMaskIsInline                 = 1 << 1,
    kQ3DisplayGroupStateMaskUseBoundingBox           = 1 << 2,
    kQ3DisplayGroupStateMaskUseBoundingSphere        = 1 << 3,
    kQ3DisplayGroupStateMaskIsPicked                 = 1 << 4,
    kQ3DisplayGroupStateMaskIsWritten                = 1 << 5
} TQ3DisplayGroupStateMasks;
```

Data Types

```
typedef struct TQ3GroupPositionPrivate             *TQ3GroupPosition;

typedef unsigned long                              TQ3DisplayGroupState;
```

Group Objects Routines

Creating Groups

```
TQ3GroupObject Q3Group_New    (void);

TQ3GroupObject Q3LightGroup_New (
                               void);

TQ3GroupObject Q3DisplayGroup_New (
                               void);

TQ3GroupObject Q3InfoGroup_New(void);

TQ3GroupObject Q3OrderedDisplayGroup_New (
                               void);

TQ3GroupObject Q3IOProxyDisplayGroup_New (
                               void);
```

Managing Groups

```
TQ3ObjectType Q3Group_GetType (TQ3GroupObject group);

TQ3Status Q3Group_CountObjects(TQ3GroupObject group,
                               unsigned long *nObjects);

TQ3Status Q3Group_CountObjectsOfType (
                               TQ3GroupObject group,
                               TQ3ObjectType isType,
                               unsigned long *nObjects);

TQ3GroupPosition Q3Group_AddObject (
                               TQ3GroupObject group, TQ3Object object);

TQ3GroupPosition Q3Group_AddObjectBefore (
                               TQ3GroupObject group,
                               TQ3GroupPosition position,
                               TQ3Object object);
```

```
TQ3GroupPosition Q3Group_AddObjectAfter (
                              TQ3GroupObject group,
                              TQ3GroupPosition position,
                              TQ3Object object);

TQ3Status Q3Group_GetPositionObject (
                              TQ3GroupObject group,
                              TQ3GroupPosition position,
                              TQ3Object *object);

TQ3Status Q3Group_SetPositionObject (
                              TQ3GroupObject group,
                              TQ3GroupPosition position,
                              TQ3Object object);

TQ3Object Q3Group_RemovePosition (
                              TQ3GroupObject group,
                              TQ3GroupPosition position);

TQ3Status Q3Group_EmptyObjects(TQ3GroupObject group);

TQ3Status Q3Group_EmptyObjectsOfType (
                              TQ3GroupObject group, TQ3ObjectType isType);
```

Managing Display Groups

```
TQ3ObjectType Q3DisplayGroup_GetType (
                              TQ3GroupObject group);

TQ3Status Q3DisplayGroup_GetState (
                              TQ3GroupObject group,
                              TQ3DisplayGroupState *state);

TQ3Status Q3DisplayGroup_SetState (
                              TQ3GroupObject group,
                              TQ3DisplayGroupState state);

TQ3Status Q3DisplayGroup_Submit (
                              TQ3GroupObject group, TQ3ViewObject view);
```

Getting Group Positions

```
TQ3Status Q3Group_GetFirstPosition (
                              TQ3GroupObject group,
                              TQ3GroupPosition *position);

TQ3Status Q3Group_GetFirstPositionOfType (
                              TQ3GroupObject group,
                              TQ3ObjectType isType,
                              TQ3GroupPosition *position);

TQ3Status Q3Group_GetLastPosition (
                              TQ3GroupObject group,
                              TQ3GroupPosition *position);

TQ3Status Q3Group_GetLastPositionOfType (
                              TQ3GroupObject group,
                              TQ3ObjectType isType,
                              TQ3GroupPosition *position);

TQ3Status Q3Group_GetNextPosition (
                              TQ3GroupObject group,
                              TQ3GroupPosition *position);

TQ3Status Q3Group_GetNextPositionOfType (
                              TQ3GroupObject group,
                              TQ3ObjectType isType,
                              TQ3GroupPosition *position);

TQ3Status Q3Group_GetPreviousPosition (
                              TQ3GroupObject group,
                              TQ3GroupPosition *position);

TQ3Status Q3Group_GetPreviousPositionOfType (
                              TQ3GroupObject group,
                              TQ3ObjectType isType,
                              TQ3GroupPosition *position);
```

Getting Object Positions

```
TQ3Status Q3Group_GetFirstObjectPosition (
                              TQ3GroupObject group,
                              TQ3Object object,
                              TQ3GroupPosition *position);

TQ3Status Q3Group_GetLastObjectPosition (
                              TQ3GroupObject group,
                              TQ3Object object,
                              TQ3GroupPosition *position);

TQ3Status Q3Group_GetNextObjectPosition (
                              TQ3GroupObject group,
                              TQ3Object object,
                              TQ3GroupPosition *position);

TQ3Status Q3Group_GetPreviousObjectPosition (
                              TQ3GroupObject group,
                              TQ3Object object,
                              TQ3GroupPosition *position);
```

Errors

`kQ3ErrorInvalidPositionForGroup`	No such position in the group
`kQ3ErrorInvalidObjectForGroup`	No such object in the group
`kQ3ErrorInvalidObjectForPosition`	No such object in the position

Renderer Objects

Contents

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.

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 11-10 provides a complete description of the routines you can use to create and manipulate renderer objects.

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. 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`.

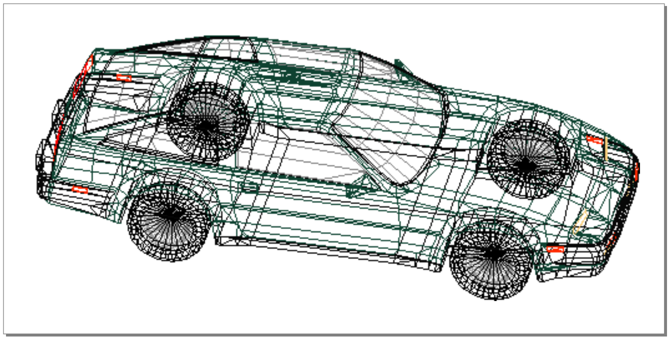
Types of Renderers

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).

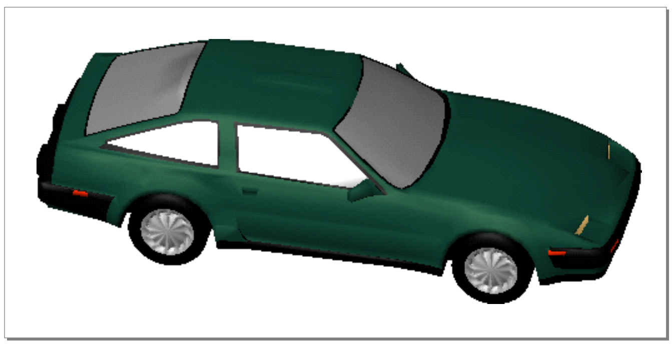
Because a wireframe image is simply a line drawing, there is no way to illuminate or shade surfaces. The wireframe renderer ignores the group of lights associated with a view and invokes none of the standard shaders supplied by QuickDraw 3D.

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.)

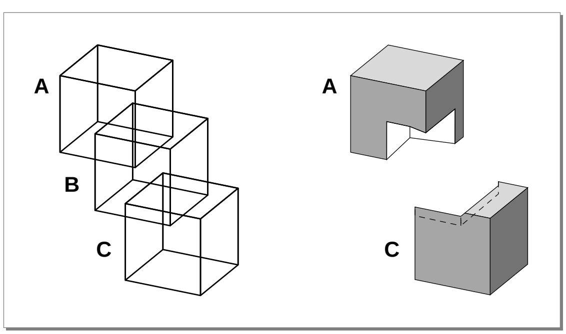
The interactive renderer supports all three available illumination shaders (Phong, Lambert, and null). Some rendering capabilities, however, are available only when the interactive renderer is using the hardware accelerator supplied by Apple Computer, Inc., including transparency, shadows, and constructive solid geometry (CSG). In addition, the interactive renderer always ignores the `clearImageMethod` field of a draw context data structure, whether using software-only rasterization or a hardware accelerator. The screen is always cleared with the clear image color specified in the `clearImageColor` field.

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 $(A \cup C) \cap \neg B$.

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, as an attribute of type `kQ3AttributeType_ConstructiveSolidGeometryID`. 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.

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
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

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 on page 11-6, drawn using the operation $(A \cup C) \cap \neg B$, is 00110010001100100011001000110010, or 0x32323232.

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 1-29) 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" on page 11-11 and "Engine IDs" on page 11-11.

Renderer Objects Reference

This section describes the constants 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
```

Constant descriptions

`kQAVendor_BestChoice`
: The best available choice. QuickDraw 3D selects the available drawing engine that produces the best output on the target device.

`kQAVendor_Apple`
: Apple Computer, Inc.

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 `kQ3AttributeType_ConstructiveSolidGeometryID` 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" on page 11-6 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.

Renderer Objects Routines

This section describes QuickDraw 3D routines that you can use to manage renderer objects.

Creating and Managing Renderers

QuickDraw 3D provides a routine 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 recreate a 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`.

Q3Renderer_Sync

You can use the `Q3Renderer_Sync` function to ensure that a drawing operation has completed.

```
TQ3Status Q3Renderer_Sync (
                    TQ3RendererObject renderer,
                    TQ3ViewObject view);
```

`renderer` A renderer.

`view` A view.

DESCRIPTION

The `Q3Renderer_Sync` function waits until the completion of the drawing that is currently being performed by the renderer specified by the `renderer` parameter in the view specified by the `view` parameter. If the specified renderer is implemented entirely in software, calling the `Q3Renderer_Sync` function has no effect. If, however, the specified renderer relies on a hardware accelerator for some or all of its operation, the `Q3Renderer_Sync` function waits until the renderer is done drawing in the specified view and then returns. In either case, therefore, you can safely perform any operations that depend on the completion of a renderer's drawing after `Q3Renderer_Sync` returns `kQ3Success`.

SPECIAL CONSIDERATIONS

Calling the `Q3Renderer_Sync` function can adversely affect the performance of your application. You should call this function only when you need to know that a drawing operation has completed (for example, if you want to allow the user to select objects in the model by clicking on the model's image on the screen, or if you want to grab a copy of the image on the screen).

Q3Renderer_Flush

You can use the `Q3Renderer_Flush` function to flush any image buffers maintained internally by a renderer.

```
TQ3Status Q3Renderer_Flush (
                    TQ3RendererObject renderer,
                    TQ3ViewObject view);
```

`renderer`	A renderer.
`view`	A view.

DESCRIPTION

The `Q3Renderer_Flush` function flushes any image buffers maintained internally by the renderer specified by the `renderer` parameter when drawing in the view specified by the `view` parameter. This function is useful only when the draw context associated with the specified view is in single-buffering mode. In that case, the renderer might need to allocate a temporary buffer to hold data before it can render an image. In general, the user will not see any of the image until your application calls `Q3View_EndRendering`. You can, however, call the `Q3Renderer_Flush` function inside the rendering loop to force the renderer to draw objects as they are submitted for drawing.

SPECIAL CONSIDERATIONS

Calling the `Q3Renderer_Flush` function can adversely affect the performance of your application. You should call this function only when you need to force the renderer to draw objects as they are submitted for drawing.

Managing Interactive Renderers

QuickDraw 3D provides routines that you can use to manage interactive renderers.

Q3InteractiveRenderer_GetPreferences

You can use the `Q3InteractiveRenderer_GetPreferences` function to get the current preference settings of the 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" on page 11-11 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" on page 11-11 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" on page 11-11 for the values you can pass in this parameter.

engineID An engine ID. See "Engine IDs" on page 11-11 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" on page 11-12 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" on page 11-12 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.

Summary of Renderer Objects

C Summary

Constants

Renderer Types

```
#define kQ3RendererTypeWireFrame           Q3_OBJECT_TYPE('w','r','f','r')
#define kQ3RendererTypeGeneric             Q3_OBJECT_TYPE('g','n','r','r')
#define kQ3RendererTypeInteractive         Q3_OBJECT_TYPE('c','t','w','n')
```

Vendor IDs

```
#define kQAVendor_BestChoice               (-1)
#define kQAVendor_Apple                    0
```

Engine IDs

```
#define kQAEngine_AppleHW                  (-1)
#define kQAEngine_AppleSW                  0
```

CSG Attribute Type

```
#define kQ3AttributeType_ConstructiveSolidGeometryID\
                                           Q3_OBJECT_TYPE('c','s','g','i')
```

CSG Object IDs

```
#define kQ3SolidGeometryObjA                    0
#define kQ3SolidGeometryObjB                    1
#define kQ3SolidGeometryObjC                    2
#define kQ3SolidGeometryObjD                    3
#define kQ3SolidGeometryObjE                    4
```

CSG Equations

```
typedef enum TQ3CSGEquation {
    kQ3CSGEquationAandB                      = (int) 0x88888888,
    kQ3CSGEquationAandnotB                   = 0x22222222,
    kQ3CSGEquationAanBonCad                  = 0x2F222F22,
    kQ3CSGEquationnotAandB                   = 0x44444444,
    kQ3CSGEquationnAaBorCanD                 = 0x74747474
} TQ3CSGEquation;
```

Renderer Objects Routines

Creating and Managing Renderers

```
TQ3RendererObject Q3Renderer_NewFromType (
                              TQ3ObjectType rendererObjectType);

TQ3ObjectType Q3Renderer_GetType (
                              TQ3RendererObject renderer);

TQ3Status Q3Renderer_Sync    (TQ3RendererObject renderer,
                              TQ3ViewObject view);

TQ3Status Q3Renderer_Flush   (TQ3RendererObject renderer,
                              TQ3ViewObject view);
```

Managing Interactive Renderers

```
TQ3Status Q3InteractiveRenderer_GetPreferences (
                                TQ3RendererObject renderer,
                                long *vendorID,
                                long *engineID);

TQ3Status Q3InteractiveRenderer_SetPreferences (
                                TQ3RendererObject renderer,
                                long vendorID,
                                long engineID);

TQ3Status Q3InteractiveRenderer_GetCSGEquation (
                                TQ3RendererObject renderer,
                                TQ3CSGEquation *equation);

TQ3Status Q3InteractiveRenderer_SetCSGEquation (
                                TQ3RendererObject renderer,
                                TQ3CSGEquation equation);

TQ3Status Q3InteractiveRenderer_GetDoubleBufferBypass (
                                TQ3RendererObject renderer,
                                TQ3Boolean *bypass);

TQ3Status Q3InteractiveRenderer_SetDoubleBufferBypass (
                                TQ3RendererObject renderer,
                                TQ3Boolean bypass);
```

Errors and Warnings

```
kQ3ErrorUnknownStudioType
kQ3ErrorAlreadyRendering
kQ3ErrorStartGroupRange
kQ3ErrorUnsupportedGeometryType
kQ3ErrorInvalidGeometryType
kQ3ErrorUnsupportedFunctionality
kQ3WarningFunctionalityNotSupported
```

Draw Context Objects

Contents

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 Macintosh 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 12-8 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 Macintosh 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 Macintosh 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 Macintosh 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 Macintosh 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

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;
```

These properties 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. ◆

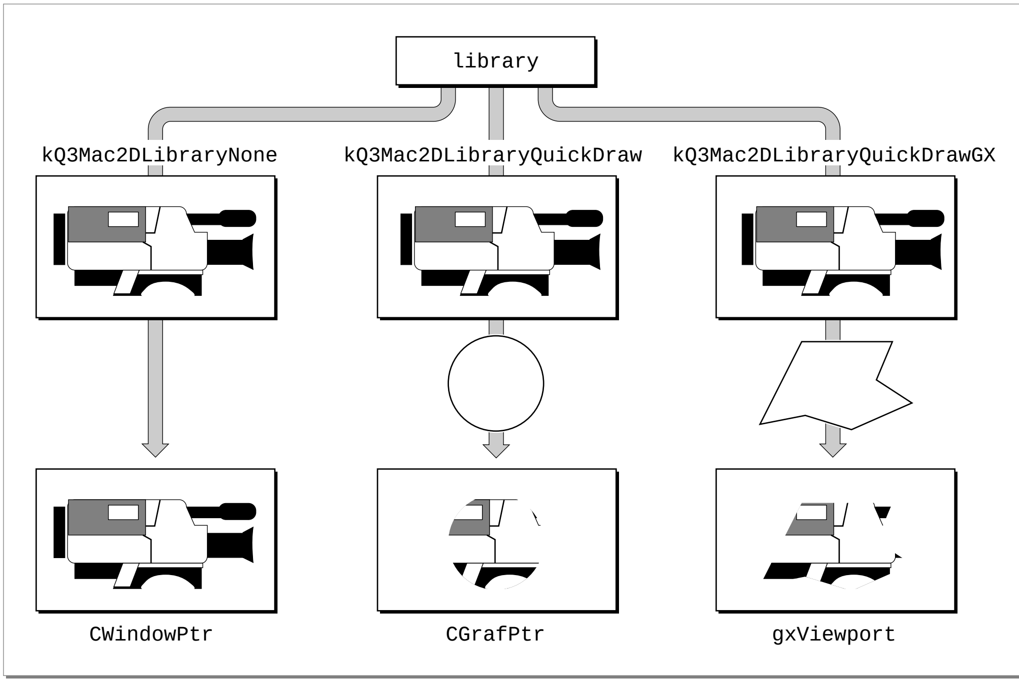
In addition to these basic properties that are common to all draw contexts, each specific type of draw context defines context-specific properties. For example, the Macintosh draw context maintains information about the window into which QuickDraw 3D is to draw, the optional use of a two-dimensional 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 Macintosh window. You specify a Macintosh 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.

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 12-4). The remaining fields contain specific information about the Macintosh window and 2D graphics library associated with the draw context.

See Listing 1-7 on page 1-27 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 `Q3Renderer_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 `Q3Renderer_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.

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
```

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 12-9 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 12-9 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” on page 12-10 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 12-11 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 12-11 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.

Summary of the Draw Context Objects

C Summary

Constants

```
#define kQ3DrawContextTypePixmap            Q3_OBJECT_TYPE('d','p','x','p')
#define kQ3DrawContextTypeMacintosh         Q3_OBJECT_TYPE('d','m','a','c')

typedef enum TQ3DrawContextClearImageMethod {
    kQ3ClearMethodNone,
    kQ3ClearMethodWithColor
} TQ3DrawContextClearImageMethod;

typedef enum TQ3MacDrawContext2DLibrary {
    kQ3Mac2DLibraryNone,
    kQ3Mac2DLibraryQuickDraw,
    kQ3Mac2DLibraryQuickDrawGX
} TQ3MacDrawContext2DLibrary;
```

Data Types

```
typedef TQ3SharedObject                          TQ3DrawContextObject;
```

Draw Context Data Structure

```
typedef struct TQ3DrawContextData {
    TQ3DrawContextClearImageMethod          clearImageMethod;
    TQ3ColorARGB                            clearImageColor;
    TQ3Area                                 pane;
    TQ3Boolean                              paneState;
```

```
    TQ3Bitmap                                   mask;
    TQ3Boolean                                  maskState;
    TQ3Boolean                                  doubleBufferState;
} TQ3DrawContextData;
```

Macintosh Draw Context Data Structure

```
typedef struct TQ3MacDrawContextData {
    TQ3DrawContextData                          drawContextData;
    CWindowPtr                                  window;
    TQ3MacDrawContext2DLibrary                  library;
    gxViewPort                                  viewPort;
    CGrafPtr                                    grafPort;
} TQ3MacDrawContextData;
```

Pixmap Draw Context Data Structure

```
typedef struct TQ3PixmapDrawContextData {
    TQ3DrawContextData                          drawContextData;
    TQ3Pixmap                                   pixmap;
} TQ3PixmapDrawContextData;
```

Draw Context Objects Routines

Managing Draw Contexts

```
TQ3ObjectType Q3DrawContext_GetType (
                               TQ3DrawContextObject drawContext);

TQ3Status Q3DrawContext_GetData (
                               TQ3DrawContextObject context,
                               TQ3DrawContextData *contextData);

TQ3Status Q3DrawContext_SetData (
                               TQ3DrawContextObject context,
                               const TQ3DrawContextData *contextData);
```

```
TQ3Status Q3DrawContext_GetClearImageColor (
                              TQ3DrawContextObject context,
                              TQ3ColorARGB *color);

TQ3Status Q3DrawContext_SetClearImageColor (
                              TQ3DrawContextObject context,
                              const TQ3ColorARGB *color);

TQ3Status Q3DrawContext_GetPane (
                              TQ3DrawContextObject context,
                              TQ3Area *pane);

TQ3Status Q3DrawContext_SetPane (
                              TQ3DrawContextObject context,
                              const TQ3Area *pane);

TQ3Status Q3DrawContext_GetPaneState (
                              TQ3DrawContextObject context,
                              TQ3Boolean *state);

TQ3Status Q3DrawContext_SetPaneState (
                              TQ3DrawContextObject context,
                              TQ3Boolean state);

TQ3Status Q3DrawContext_GetClearImageMethod (
                              TQ3DrawContextObject context,
                              TQ3DrawContextClearImageMethod *method);

TQ3Status Q3DrawContext_SetClearImageMethod (
                              TQ3DrawContextObject context,
                              TQ3DrawContextClearImageMethod method);

TQ3Status Q3DrawContext_GetMask (
                              TQ3DrawContextObject context,
                              TQ3Bitmap *mask);

TQ3Status Q3DrawContext_SetMask (
                              TQ3DrawContextObject context,
                              const TQ3Bitmap *mask);
```

```
TQ3Status Q3DrawContext_GetMaskState (
                              TQ3DrawContextObject context,
                              TQ3Boolean *state);

TQ3Status Q3DrawContext_SetMaskState (
                              TQ3DrawContextObject context,
                              TQ3Boolean state);

TQ3Status Q3DrawContext_GetDoubleBufferState (
                              TQ3DrawContextObject context,
                              TQ3Boolean *state);

TQ3Status Q3DrawContext_SetDoubleBufferState (
                              TQ3DrawContextObject context,
                              TQ3Boolean state);
```

Managing Macintosh Draw Contexts

```
TQ3DrawContextObject Q3MacDrawContext_New (
                              const TQ3MacDrawContextData *drawContextData);

TQ3Status Q3MacDrawContext_GetWindow (
                              TQ3DrawContextObject drawContext,
                              CWindowPtr *window);

TQ3Status Q3MacDrawContext_SetWindow (
                              TQ3DrawContextObject drawContext,
                              const CWindowPtr window);

TQ3Status Q3MacDrawContext_Get2DLibrary (
                              TQ3DrawContextObject drawContext,
                              TQ3MacDrawContext2DLibrary *library);

TQ3Status Q3MacDrawContext_Set2DLibrary (
                              TQ3DrawContextObject drawContext,
                              TQ3MacDrawContext2DLibrary library);

TQ3Status Q3MacDrawContext_GetGXViewPort (
                              TQ3DrawContextObject drawContext,
                              gxViewPort *viewPort);
```

```
TQ3Status Q3MacDrawContext_SetGXViewPort (
                              TQ3DrawContextObject drawContext,
                              const gxViewPort viewPort);

TQ3Status Q3MacDrawContext_GetGrafPort (
                              TQ3DrawContextObject drawContext,
                              CGrafPtr *grafPort);

TQ3Status Q3MacDrawContext_SetGrafPort (
                              TQ3DrawContextObject drawContext,
                              const CGrafPtr grafPort);
```

Managing Pixmap Draw Contexts

```
TQ3DrawContextObject Q3PixmapDrawContext_New (
                              const TQ3PixmapDrawContextData *contextData);

TQ3Status Q3PixmapDrawContext_GetPixmap (
                              TQ3DrawContextObject drawContext,
                              TQ3Pixmap *pixmap);

TQ3Status Q3PixmapDrawContext_SetPixmap (
                              TQ3DrawContextObject drawContext,
                              const TQ3Pixmap *pixmap);
```

Errors, Warnings, and Notices

kQ3ErrorBadDrawContextType	Unrecognized draw context type
kQ3ErrorBadDrawContextFlag	Unrecognized draw context flag
kQ3ErrorBadDrawContext	Invalid draw context
kQ3ErrorUnsupportedPixelDepth	Specified pixel depth not supported by draw context
kQ3WarningInvalidPaneDimensions	Invalid panel dimensions
kQ3NoticeDrawContextNotSetUsingInternalDefaults	Draw context not set

View Objects

Contents

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" in this book 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.

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 13-7 provides a complete description of the routines you can use to create and manipulate view objects.

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.

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.

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 on page 1-30 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" in this book.

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” in this book.

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" in this book.

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" in this book.

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” on page 13-4 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 `Q3Renderer_Sync` function (described in the chapter “Renderer Objects”). ▲

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" on page 13-4 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`.

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 you've seen (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" on page 13-41 for information on writing an idle method.

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 *idleData);
```

view	A view.
idleMethod	A pointer to an idle method.
idleData	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 idleData parameter is passed to your callback routine whenever it is executed.

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.

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" in this book. ◆

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.

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 kQ3ViewDefaultAmbientCoefficient      1.0
#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 kQ3ViewDefaultSubdivisionMethod       \
                                    kQ3SubdivisionMethodConstant
#define kQ3ViewDefaultSubdivisionC1           10.0
#define kQ3ViewDefaultSubdivisionC2           10.0
```

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 an idle method that QuickDraw 3D calls occasionally during lengthy operations.

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.

Summary of View Objects

C Summary

Constants

View Rendering Status Values

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

Compute Bounds Values

```
typedef enum TQ3ComputeBounds {
    kQ3ComputeBoundsExact,
    kQ3ComputeBoundsApproximate
} TQ3ComputeBounds;
```

Properties of the Default Material

```
#define kQ3ViewDefaultAmbientCoefficient          1.0
#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 kQ3ViewDefaultSubdivisionMethod           kQ3SubdivisionMethodConstant
#define kQ3ViewDefaultSubdivisionC1               10.0
#define kQ3ViewDefaultSubdivisionC2               10.0
```

View Objects Routines

Creating and Configuring Views

```
TQ3ViewObject Q3View_New      (void);

TQ3Status Q3View_GetRenderer  (TQ3ViewObject view,
                               TQ3RendererObject *renderer);

TQ3Status Q3View_SetRenderer  (TQ3ViewObject view,
                               TQ3RendererObject renderer);

TQ3Status Q3View_SetRendererByType (
                               TQ3ViewObject view, TQ3ObjectType type);

TQ3Status Q3View_GetCamera    (TQ3ViewObject view,
                               TQ3CameraObject *camera);

TQ3Status Q3View_SetCamera    (TQ3ViewObject view,
                               TQ3CameraObject camera);

TQ3Status Q3View_GetLightGroup(TQ3ViewObject view,
                               TQ3GroupObject *lightGroup);

TQ3Status Q3View_SetLightGroup(TQ3ViewObject view,
                               TQ3GroupObject lightGroup);

TQ3Status Q3View_GetDrawContext (
                               TQ3ViewObject view,
                               TQ3DrawContextObject *drawContext);

TQ3Status Q3View_SetDrawContext (
                               TQ3ViewObject view,
                               TQ3DrawContextObject drawContext);
```

Rendering in a View

```
TQ3Status Q3View_StartRendering(TQ3ViewObject view);

TQ3ViewStatus Q3View_EndRendering (
                               TQ3ViewObject view);

TQ3Status Q3View_Cancel       (TQ3ViewObject view);
```

Picking in a View

```
TQ3Status Q3View_StartPicking (TQ3ViewObject view, TQ3PickObject pick);
```

```
TQ3ViewStatus Q3View_EndPicking (
                                TQ3ViewObject view);
```

Writing in a View

```
TQ3Status Q3View_StartWriting (TQ3ViewObject view, TQ3FileObject file);
```

```
TQ3ViewStatus Q3View_EndWriting (
                                TQ3ViewObject view);
```

Bounding in a View

```
TQ3Status Q3View_StartBoundingBox (
                                TQ3ViewObject view,
                                TQ3ComputeBounds computeBounds);
```

```
TQ3ViewStatus Q3View_EndBoundingBox (
                                TQ3ViewObject view, TQ3BoundingBox *result);
```

```
TQ3Status Q3View_StartBoundingSphere (
                                TQ3ViewObject view,
                                TQ3ComputeBounds computeBounds);
```

```
TQ3ViewStatus Q3View_EndBoundingSphere (
                                TQ3ViewObject view,
                                TQ3BoundingSphere *result);
```

```
TQ3Boolean Q3View_IsBoundingBoxVisible (
                                TQ3ViewObject view,
                                const TQ3BoundingBox *bbox);
```

Setting Idle Methods

```
TQ3Status Q3View_SetIdleMethod(TQ3ViewObject view,
                                TQ3ViewIdleMethod idleMethod,
                                const void *idleData);
```

Writing Custom Data

```
TQ3Status Q3View_SubmitWriteData (
                                TQ3ViewObject view,
                                TQ3Size size,
                                void *data,
                                void (*deleteData));
```

Pushing and Popping the Graphics State

```
TQ3Status Q3Push_Submit        (TQ3ViewObject view);

TQ3Status Q3Pop_Submit         (TQ3ViewObject view);
```

Getting a View's Transforms

```
TQ3Status Q3View_GetLocalToWorldMatrixState (
                                TQ3ViewObject view, TQ3Matrix4x4 *matrix);

TQ3Status Q3View_GetWorldToFrustumMatrixState (
                                TQ3ViewObject view, TQ3Matrix4x4 *matrix);

TQ3Status Q3View_GetFrustumToWindowMatrixState (
                                TQ3ViewObject view, TQ3Matrix4x4 *matrix);
```

Managing a View's Style States

```
TQ3Status Q3View_GetBackfacingStyleState (
                                TQ3ViewObject view,
                                TQ3BackfacingStyle *backfacingStyle);

TQ3Status Q3View_GetInterpolationStyleState (
                                TQ3ViewObject view,
                                TQ3InterpolationStyle *interpolationType);

TQ3Status Q3View_GetFillStyleState (
                                TQ3ViewObject view,
                                TQ3FillStyle *fillStyle);
```

```
TQ3Status Q3View_GetHighlightStyleState (
                              TQ3ViewObject view,
                              TQ3AttributeSet *highlightStyle);

TQ3Status Q3View_GetSubdivisionStyleState (
                              TQ3ViewObject view,
                              TQ3SubdivisionStyleData *subdivisionStyle);

TQ3Status Q3View_GetOrientationStyleState (
                              TQ3ViewObject view,
                              TQ3OrientationStyle
                              *fontFacingDirectionStyle);

TQ3Status Q3View_GetReceiveShadowsStyleState (
                              TQ3ViewObject view,
                              TQ3Boolean *receives);

TQ3Status Q3View_GetPickIDStyleState (
                              TQ3ViewObject view,
                              unsigned long *pickIDStyle);

TQ3Status Q3View_GetPickPartsStyleState (
                              TQ3ViewObject view,
                              TQ3PickParts *pickPartsStyle);
```

Managing a View's Attribute Set

```
TQ3Status Q3View_GetDefaultAttributeSet (
                              TQ3ViewObject view,
                              TQ3AttributeSet *attributeSet);

TQ3Status Q3View_SetDefaultAttributeSet (
                              TQ3ViewObject view,
                              TQ3AttributeSet attributeSet);

TQ3Status Q3View_GetAttributeSetState (
                              TQ3ViewObject view,
                              TQ3AttributeSet *attributeSet);
```

```
TQ3Status Q3View_GetAttributeState (
                               TQ3ViewObject view,
                               TQ3AttributeType attributeType,
                               void *data);
```

Application-Defined Routines

```
typedef TQ3Status (*TQ3ViewIdleMethod) (
                               TQ3ViewObject view,
                               const void *idleData);
```

Errors and Warnings

```
kQ3ErrorViewNotStarted
kQ3ErrorViewIsStarted
kQ3ErrorRendererNotSet
kQ3ErrorRenderingIsActive
kQ3ErrorImmediateModeUnderflow
kQ3ErrorDisplayNotSet
kQ3ErrorCameraNotSet
kQ3ErrorDrawContextNotSet
kQ3ErrorNonInvertibleMatrix
kQ3ErrorRenderingNotStarted
kQ3ErrorPickingNotStarted
kQ3ErrorBoundsNotStarted
kQ3ErrorDataNotAvailable
kQ3ErrorNothingToPop
kQ3WarningViewTraversalInProgress
kQ3WarningNonInvertibleMatrix
```

Shader Objects

Contents

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 14-16 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" on page 14-10 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" on page 14-4 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:

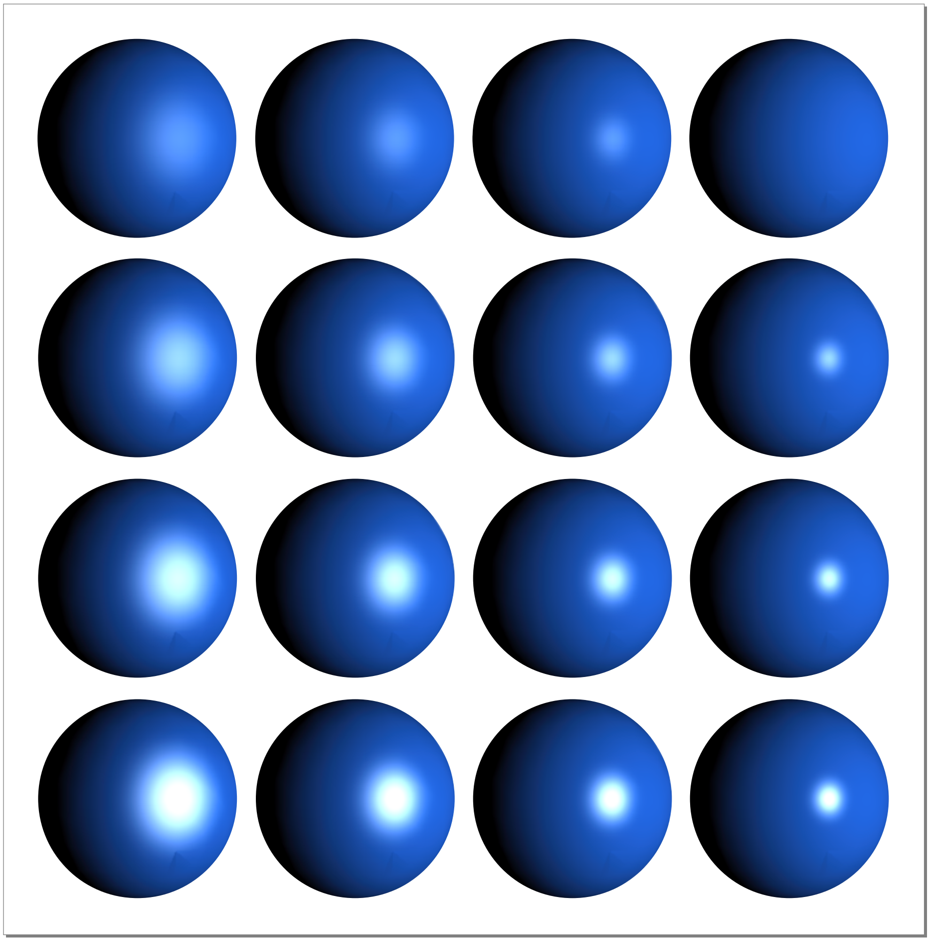
$$I_{Phong} = I_a k_a O_d + \sum_i [((N \bullet L_i) I_i k_d O_d) + ((R \bullet V)^n k_s)]$$

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.

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.

Figure 14-3 Phong illumination with various specular exponents and coefficients

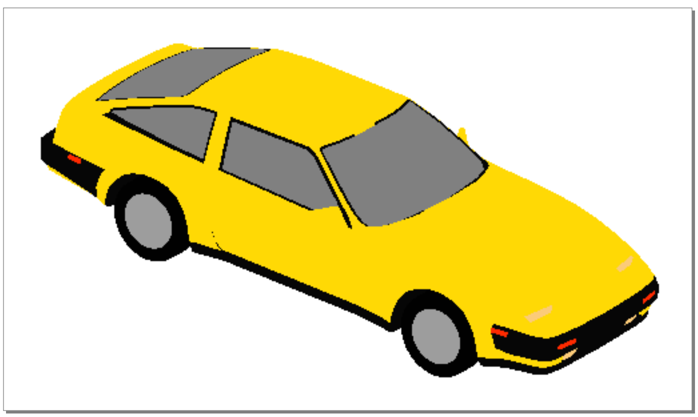


Note
A surface's specular reflection coefficient is also called its **specular control.** You specify a specular reflection coefficient by inserting an attribute of type `kQ3AttributeTypeSpecularControl` into the surface's attribute set. ◆

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" on page 14-11 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. Indeed, you can apply a surface shader in yet a third way, by attaching it to an object as an attribute. 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 on page 14-13) 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, neighborhoods, textures, and attachments.

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 14-16. ◆

```
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" on page 14-17 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" on page 14-17 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" on page 14-17 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" on page 14-17 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 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" on page 14-4 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" on page 14-4 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" on page 14-30 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 this constant:

```
kQ3TextureTypePixmap
```

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.

Summary of Shader Objects

C Summary

Constants

```
typedef enum TQ3ShaderUVBoundary {
    kQ3ShaderUVBoundaryWrap,
    kQ3ShaderUVBoundaryClamp
} TQ3ShaderUVBoundary;

#define kQ3ShaderTypeSurface            Q3_OBJECT_TYPE('s','u','s','h')
#define kQ3ShaderTypeIllumination       Q3_OBJECT_TYPE('i','l','s','h')
#define kQ3SurfaceShaderTypeTexture     Q3_OBJECT_TYPE('t','x','s','u')

#define kQ3IlluminationTypeLambert      Q3_OBJECT_TYPE('l','m','i','l')
#define kQ3IlluminationTypePhong        Q3_OBJECT_TYPE('p','h','i','l')
#define kQ3IlluminationTypeNULL         Q3_OBJECT_TYPE('n','u','l','l')

#define kQ3TextureTypePixmap            Q3_OBJECT_TYPE('t','x','p','m')
```

Shader Objects Routines

Managing Shaders

```
TQ3ObjectType Q3Shader_GetType(TQ3ShaderObject shader);

TQ3Status Q3Shader_Submit     (TQ3ShaderObject shader, TQ3ViewObject view);
```

Managing Shader Characteristics

```
TQ3Status Q3Shader_GetUVTransform (
                                TQ3ShaderObject shader,
                                TQ3Matrix3x3 *uvTransform);

TQ3Status Q3Shader_SetUVTransform (
                                TQ3ShaderObject shader,
                                const TQ3Matrix3x3 *uvTransform);

TQ3Status Q3Shader_GetUBoundary (
                                TQ3ShaderObject shader,
                                TQ3ShaderUVBoundary *uBoundary);

TQ3Status Q3Shader_SetUBoundary (
                                TQ3ShaderObject shader,
                                TQ3ShaderUVBoundary uBoundary);

TQ3Status Q3Shader_GetVBoundary (
                                TQ3ShaderObject shader,
                                TQ3ShaderUVBoundary *vBoundary);

TQ3Status Q3Shader_SetVBoundary (
                                TQ3ShaderObject shader,
                                TQ3ShaderUVBoundary vBoundary);
```

Managing Texture Shaders

```
TQ3ShaderObject Q3TextureShader_New (
                                TQ3TextureObject texture);

TQ3Status Q3TextureShader_GetTexture (
                                TQ3ShaderObject shader,
                                TQ3TextureObject *texture);

TQ3Status Q3TextureShader_SetTexture (
                                TQ3ShaderObject shader,
                                TQ3TextureObject texture);
```

Managing Illumination Shaders

```
TQ3ShaderObject Q3LambertIllumination_New (
                               void);

TQ3ShaderObject Q3PhongIllumination_New (
                               void);

TQ3ShaderObject Q3NULLIllumination_New (
                               void);

TQ3ObjectType Q3IlluminationShader_GetType (
                               TQ3ShaderObject shader);
```

Managing Textures

```
TQ3ObjectType Q3Texture_GetType (
                               TQ3TextureObject texture);

TQ3Status Q3Texture_GetWidth  (TQ3TextureObject texture,
                               unsigned long *width);

TQ3Status Q3Texture_GetHeight (TQ3TextureObject texture,
                               unsigned long *height);
```

Managing Pixmap Textures

```
TQ3TextureObject Q3PixmapTexture_New (
                               const TQ3StoragePixmap *pixmap);

TQ3Status Q3PixmapTexture_GetPixmap (
                               TQ3TextureObject texture,
                               TQ3StoragePixmap *pixmap);

TQ3Status Q3PixmapTexture_SetPixmap (
                               TQ3TextureObject texture,
                               const TQ3StoragePixmap *pixmap);
```

Pick Objects

Contents

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" earlier in this book. For information about views, 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 pick objects and their features. Then it shows how to create and use pick objects. The section "Pick Objects Reference," beginning on page 15-17 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 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 specify 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 on page 15-12 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 object. 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 some pick geometries, 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 are used only with one- and two-dimensional pick geometries.

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

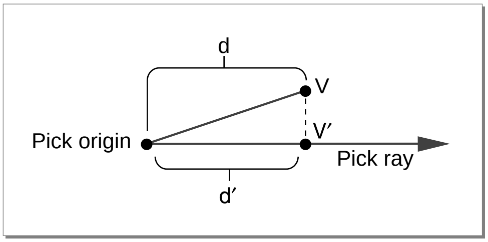
In some cases, you can have QuickDraw 3D sort a hit list before returning it to your application. The sorting is based on either increasing or decreasing distance from some point, the **pick origin.** As a result, hit-list sorting is possible only when the pick geometry has a clearly defined pick origin. 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.

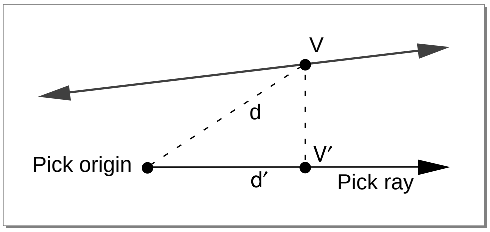
Note that the distances used to establish a sort order might not be the same distances reported to your application when you retrieve hit information. Consider, for example, the situation illustrated in Figure 15-1. Here, the vertex *V* is within the current vertex tolerance of the world ray pick object 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



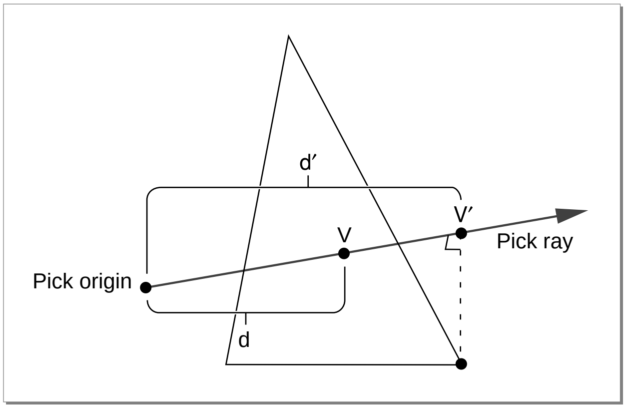
QuickDraw 3D calculates distances to edges and faces in an analogous manner. If the pick ray passes within the current edge tolerance of an edge, the sorting distance is set to the distance *d′* from the pick ray origin to the projection onto the pick ray of the point on the edge that is closest to the 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_GetHitData` function. You pass this function a pick object and a pointer to a **hit data structure.** A hit data structure is defined by the `TQ3HitData` data type.

```
typedef struct TQ3HitData {
   TQ3PickParts                  part;
   TQ3PickDetail                 validMask;
   unsigned long                 pickID;
   TQ3HitPath                    path;
   TQ3Object                     object;
   TQ3Matrix4x4                  localToWorldMatrix;
   TQ3Point3D                    xyzPoint;
   float                         distance;
   TQ3Vector3D                   normal;
   TQ3ShapePartObject            shapePart;
} TQ3HitData;
```

Note
See "Hit Data Structure" on page 15-23 for complete information about the fields of a hit data structure. ◆

QuickDraw 3D fills in fields of the structure you pass it and sets the `validMask` field to indicate which of the fields are valid. Before reading any information from the fields of a returned hit data structure, you should check `validMask` to see what information QuickDraw 3D has returned. The values in the `mask` field of a pick data structure and the `validMask` field of a hit data structure 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 in the `pickID`, `path`, `object`, and `localToWorldMatrix` fields. 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;
    TQ3HitData                  myHitData;
    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, myPickObject, myView);
    } while (Q3View_EndPicking(myView) == kQ3ViewStatusRetraverse);

    /*See whether any hits occurred.*/
    if (!Q3Pick_GetNumHits(myPickObject, &myNumHits) || !(myNumHits == 0)) {
       Q3Object_Dispose(myPickObject);
       return;
    }

    /*Process each hit.*/
    for (myIndex = 0; myIndex < myNumHits; myIndex++) {
       Q3Pick_GetHitData(myPickObject, myIndex, &myHitData);
       /*operate on myHitData, then...*/
          …
       Q3Hit_EmptyData(&myHitData);           /*dispose of hit data*/
    }

    /*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 a hit data structure returned by `Q3Pick_GetHitData` applies to a shape part by inspecting the `shapePart` field of that structure. If the value in the field is non-`NULL`, the structure 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_GetHitData(myPickObject, myIndex, &myHitData);

if (myHitData.shapePart != NULL) {
    switch(Q3Object_GetLeafType(myHitData.shapePart)) {
        case kQ3MeshPartTypeMeshFacePart:
            Q3MeshFacePart_GetFace(myHitData.shapePart, &myFace);
```

```
            MyDoPickFace(myHitData.object, myFace);
            break;
        case kQ3MeshPartTypeMeshEdgePart:
            Q3MeshEdgePart_GetEdge(myHitData.shapePart, &myEdge);
            MyDoPickEdge(myHitData.object, myEdge);
            break;
        case kQ3MeshPartTypeMeshVertexPart:
            Q3MeshVertexPart_GetVertex(myHitData.shapePart, &myVertex);
            MyDoPickVertex(myHitData.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;
    TQ3HitData                      myHitData;
    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 hit data and check its pick ID.*/
        Q3Pick_GetHitData(myPick, 0, &myHitData);
        if (myHitData.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 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

You specify a hit information mask in the `mask` field of a pick data structure to indicate the type of information you want returned for the items in a hit list. When QuickDraw 3D returns a hit list to you, it sets the bits in the `validMask` field of a hit data structure to indicate the types of information it is returning. The hit information masks correspond to the fields in the hit data structure. See "Hit Data Structure" on page 15-23 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
} 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 picked object 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 normalized.

`kQ3PickDetailMaskShapePart`
: The shape part object 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;
```

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. A pick data structure 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 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 window coordinates.

`vertexTolerance` The vertex tolerance.

`edgeTolerance` The edge tolerance.

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 window coordinates.

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 Data Structure

You use a **hit data structure** to get information about an item in the hit list. The `validMask` field indicates which of the fields in the structure contain valid information. A hit data structure is defined by the `TQ3HitData` data type.

```
typedef struct TQ3HitData {
    TQ3PickParts                  part;
    TQ3PickDetail                 validMask;
    unsigned long                 pickID;
    TQ3HitPath                    path;
    TQ3Object                     object;
    TQ3Matrix4x4                  localToWorldMatrix;
    TQ3Point3D                    xyzPoint;
    float                         distance;
    TQ3Vector3D                   normal;
    TQ3ShapePartObject            shapePart;
} TQ3HitData;
```

Field descriptions

`part` The part picked. See "Pick Parts Masks" on page 15-20 for the constants that can be returned in this field.

`validMask` A long integer whose bits specify which of the following fields contain information about a picked object. See "Hit Information Masks" on page 15-18 for a list of the masks you can use to check the bits in this field.

`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 on page 15-15 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" on page 15-22 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.

`xyzPoint` — 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.

`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.

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" on page 15-21 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.

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.

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.

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.

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_GetHitData

You can use the `Q3Pick_GetHitData` function to get an item in the hit list of a pick object.

```
TQ3Status Q3Pick_GetHitData (
                    TQ3PickObject pick,
                    unsigned long index,
                    TQ3HitData *hitData);
```

`pick` A pick object.

`index` An index into a hit list. This number should be between 0 and one less than the number of items in the hit list of the specified pick object, inclusive.

`hitData` On entry, a pointer to a hit data structure. On exit, a pointer to a hit data structure for the specified item in the hit list of the specified pick object.

DESCRIPTION

The `Q3Pick_GetHitData` function returns, in the `hitData` parameter, a pointer to a hit data structure for the item that has the index specified by the `index` parameter in the hit list associated with the pick object specified by the `pick` parameter. The hit data structure whose address is passed in the `hitData` parameter must be created by your application. QuickDraw 3D allocates memory to hold any additional information returned in the hit data structure; you should call `Q3Hit_EmptyData` to dispose of that memory when you are finished using the hit data.

Q3Hit_EmptyData

You can use the `Q3Hit_EmptyData` function to empty a hit data structure.

```
TQ3Status Q3Hit_EmptyData (TQ3HitData *hitData);
```

`hitData` A pointer to a hit data structure.

DESCRIPTION

The Q3Hit_EmptyData function disposes of all QuickDraw 3D-allocated memory occupied by the data in the hit data structure specified by the hitData parameter. You should call Q3Hit_EmptyData for any hit data structures you had filled out by Q3Pick_GetHitData.

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.

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.

Note
You don't need to call Q3ShapePart_GetShape if you've already retrieved a hit data structure by calling Q3Pick_GetHitData because the containing object is specified by the object field of that structure. ◆

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 one of these constants:

```
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.

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” on page 15-21 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.

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 specified 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" on page 15-22 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.

Summary of Pick Objects

C Summary

Constants

```
#define kQ3ReturnAllHits                    0
```

Pick Object Types

```
#define kQ3PickTypeWindowPoint              Q3_OBJECT_TYPE('p','k','w','p')
#define kQ3PickTypeWindowRect               Q3_OBJECT_TYPE('p','k','w','r')
```

Shape Part and Mesh Part Types

```
#define kQ3ShapePartTypeMeshPart            Q3_OBJECT_TYPE('s','p','m','h')
#define kQ3MeshPartTypeMeshFacePart         Q3_OBJECT_TYPE('m','f','a','c')
#define kQ3MeshPartTypeMeshEdgePart         Q3_OBJECT_TYPE('m','e','d','g')
#define kQ3MeshPartTypeMeshVertexPart       Q3_OBJECT_TYPE('m','v','t','x')
```

Hit List Sorting Values

```
typedef enum TQ3PickSort {
    kQ3PickSortNone,
    kQ3PickSortNearToFar,
    kQ3PickSortFarToNear
} TQ3PickSort;
```

Hit Information Masks

```
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}
TQ3PickDetailMasks;
```

Pick Parts Values

```
typedef enum TQ3PickPartsMasks {
    kQ3PickPartsObject                        = 0,
    kQ3PickPartsMaskFace                      = 1 << 0,
    kQ3PickPartsMaskEdge                      = 1 << 1,
    kQ3PickPartsMaskVertex                    = 1 << 2
} TQ3PickPartsMasks;
```

Data Types

```
typedef unsigned long                         TQ3PickDetail;

typedef unsigned long                         TQ3PickParts;

typedef TQ3ShapePartObject                    TQ3MeshPartObject;

typedef TQ3MeshPartObject                     TQ3MeshFacePartObject;
typedef TQ3MeshPartObject                     TQ3MeshEdgePartObject;
typedef TQ3MeshPartObject                     TQ3MeshVertexPartObject;
```

Pick Data Structure

```
typedef struct TQ3PickData {
    TQ3PickSort                     sort;
    TQ3PickDetail                   mask;
    unsigned long                   numHitsToReturn;
} TQ3PickData;
```

Window-Point Pick Data Structure

```
typedef struct TQ3WindowPointPickData {
    TQ3PickData                     data;
    TQ3Point2D                      point;
    float                           vertexTolerance;
    float                           edgeTolerance;
} TQ3WindowPointPickData;
```

Window-Rectangle Pick Data Structure

```
typedef struct TQ3WindowRectPickData {
    TQ3PickData                     data;
    TQ3Area                         rect;
} TQ3WindowRectPickData;
```

Hit Path Structure

```
typedef struct TQ3HitPath {
    unsigned long                   depth;
    TQ3GroupPosition                *positions;
} TQ3HitPath;
```

Hit Data Structure

```
typedef struct TQ3HitData {
    TQ3PickParts                    part;
    TQ3PickDetail                   validMask;
    unsigned long                   pickID;
```

```
    TQ3HitPath                          path;
    TQ3Object                           object;
    TQ3Matrix4x4                        localToWorldMatrix;
    TQ3Point3D                          xyzPoint;
    float                               distance;
    TQ3Vector3D                         normal;
    TQ3ShapePartObject                  shapePart;
} TQ3HitData;
```

Pick Objects Routines

Managing Pick Objects

```
TQ3ObjectType Q3Pick_GetType  (TQ3PickObject pick);

TQ3Status Q3Pick_GetData      (TQ3PickObject pick, TQ3PickData *data);

TQ3Status Q3Pick_SetData      (TQ3PickObject pick, const TQ3PickData *data);

TQ3Status Q3Pick_GetVertexTolerance (
                               TQ3PickObject pick, float *vertexTolerance);

TQ3Status Q3Pick_SetVertexTolerance (
                               TQ3PickObject pick, float vertexTolerance);

TQ3Status Q3Pick_GetEdgeTolerance (
                               TQ3PickObject pick, float *edgeTolerance);

TQ3Status Q3Pick_SetEdgeTolerance (
                               TQ3PickObject pick, float edgeTolerance);

TQ3Status Q3Pick_GetNumHits   (TQ3PickObject pick, unsigned long *numHits);

TQ3Status Q3Pick_GetHitData   (TQ3PickObject pick,
                               unsigned long index,
                               TQ3HitData *hitData);

TQ3Status Q3Hit_EmptyData     (TQ3HitData *hitData);

TQ3Status Q3Pick_EmptyHitList (TQ3PickObject pick);
```

Managing Shape Parts and Mesh Parts

```
TQ3Status Q3ShapePart_GetShape(TQ3ShapePartObject shapePartObject,
                               TQ3ShapeObject *shapeObject);

TQ3ObjectType Q3ShapePart_GetType (
                               TQ3ShapePartObject shapePartObject);

TQ3ObjectType Q3MeshPart_GetType (
                               TQ3MeshPartObject meshPartObject);

TQ3Status Q3MeshPart_GetComponent (
                               TQ3MeshPartObject meshPartObject,
                               TQ3MeshComponent *component);

TQ3Status Q3MeshFacePart_GetFace (
                               TQ3MeshFacePartObject meshFacePartObject,
                               TQ3MeshFace *face);

TQ3Status Q3MeshEdgePart_GetEdge (
                               TQ3MeshEdgePartObject meshEdgePartObject,
                               TQ3MeshEdge *edge);

TQ3Status Q3MeshVertexPart_GetVertex (
                               TQ3MeshVertexPartObject meshVertexPartObject,
                               TQ3MeshVertex *vertex);
```

Picking With Window Points

```
TQ3PickObject Q3WindowPointPick_New (
                               const TQ3WindowPointPickData *data);

TQ3Status Q3WindowPointPick_GetPoint (
                               TQ3PickObject pick, TQ3Point2D *point);

TQ3Status Q3WindowPointPick_SetPoint (
                               TQ3PickObject pick, const TQ3Point2D *point);

TQ3Status Q3WindowPointPick_GetData (
                               TQ3PickObject pick,
                               TQ3WindowPointPickData *data);
```

```
TQ3Status Q3WindowPointPick_SetData (
                              TQ3PickObject pick,
                              const TQ3WindowPointPickData *data);
```

Picking With Window Rectangles

```
TQ3PickObject Q3WindowRectPick_New (
                              const TQ3WindowRectPickData *data);

TQ3Status Q3WindowRectPick_GetRect (
                              TQ3PickObject pick, TQ3Area *rect);

TQ3Status Q3WindowRectPick_SetRect (
                              TQ3PickObject pick, const TQ3Area *rect);

TQ3Status Q3WindowRectPick_GetData (
                              TQ3PickObject pick,
                              TQ3WindowRectPickData *data);

TQ3Status Q3WindowRectPick_SetData (
                              TQ3PickObject pick,
                              const TQ3WindowRectPickData *data);
```

Warnings

```
kQ3WarningPickParamOutside
```

Storage Objects

Contents

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” earlier in this book. 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 16-9 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 yourself. Whenever QuickDraw 3D opens a file or allocates memory to create a storage object, it closes the file or disposes of the memory itself. ▲

It's possible, however, to have QuickDraw 3D create and manage a piece of storage for you. For example, you can create a memory storage object by calling `Q3MemoryStorage_NewBuffer` as follows:

```
myStorageObj = Q3MemoryStorage_NewBuffer(NULL, 0, 0);
```

Notice that the first parameter in this call is `NULL`; this value indicates that you want QuickDraw 3D to allocate a buffer internally and automatically expand the buffer whenever necessary. (The initial size of the buffer and the grow size of the buffer are determined by internal QuickDraw 3D settings.) In addition, when you close or dispose of the storage object, QuickDraw 3D disposes of any memory it has allocated on your behalf.

You can also have QuickDraw 3D open and close files on your behalf. On the Macintosh Operating System, you can call the `Q3FSSpecStorage_New` function, passing a file system specification structure describing a closed file. The following line of code illustrates how to do this:

```
myStorageObj = Q3FSSpecStorage_New(&myFSSpec);
```

QuickDraw 3D opens the file and creates a storage object associated with that file. When you later close or dispose of that storage object, QuickDraw 3D also closes the associated Macintosh file. Similarly, you can call `Q3UnixPathStorage_New` to have QuickDraw 3D open a file described by a path name and create a new storage object associated with it. When you later close or dispose of that storage object, QuickDraw 3D also closes the associated file.

▲ **WARNING**
No matter whether you opened a piece of storage (that is, a file or a block of memory) yourself or allowed QuickDraw 3D to open it for you, 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

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
```

Summary of Storage Objects

C Summary

Constants

```
#define kQ3StorageTypeMemory                Q3_OBJECT_TYPE('m','e','m','s')
#define kQ3StorageTypeMacintosh             Q3_OBJECT_TYPE('m','a','c','n')
#define kQ3StorageTypeUnix                  Q3_OBJECT_TYPE('u','x','s','t')

#define kQ3MemoryStorageTypeHandle          Q3_OBJECT_TYPE('h','n','d','l')
#define kQ3MacintoshStorageTypeFSSpec       Q3_OBJECT_TYPE('m','a','c','p')
#define kQ3UnixStorageTypePath              Q3_OBJECT_TYPE('u','x','i','x')
```

Storage Objects Routines

Managing Storage Objects

```
TQ3ObjectType Q3Storage_GetType (
                              TQ3StorageObject storage);

TQ3Status Q3Storage_GetSize  (TQ3StorageObject storage,
                              unsigned long *size);

TQ3Status Q3Storage_GetData  (TQ3StorageObject storage,
                              unsigned long offset,
                              unsigned long dataSize,
                              unsigned char *data,
                              unsigned long *sizeRead);
```

```
TQ3Status Q3Storage_SetData   (TQ3StorageObject storage,
                               unsigned long offset,
                               unsigned long dataSize,
                               const unsigned char *data,
                               unsigned long *sizeWritten);
```

Creating and Accessing Memory Storage Objects

```
TQ3StorageObject Q3MemoryStorage_New (
                               const unsigned char *buffer,
                               unsigned long validSize);

TQ3StorageObject Q3MemoryStorage_NewBuffer (
                               unsigned char *buffer,
                               unsigned long validSize,
                               unsigned long bufferSize);

TQ3Status Q3MemoryStorage_Set (TQ3StorageObject storage,
                               const unsigned char *buffer,
                               unsigned long validSize);

TQ3Status Q3MemoryStorage_GetBuffer (
                               TQ3StorageObject storage,
                               unsigned char **buffer,
                               unsigned long *validSize,
                               unsigned long *bufferSize);

TQ3Status Q3MemoryStorage_SetBuffer (
                               TQ3StorageObject storage,
                               unsigned char *buffer,
                               unsigned long validSize,
                               unsigned long bufferSize);

TQ3ObjectType Q3MemoryStorage_GetType (
                               TQ3StorageObject storage);
```

Creating and Accessing Handle Storage Objects

```
TQ3StorageObject Q3HandleStorage_New (
                               Handle handle, unsigned long validSize);

TQ3Status Q3HandleStorage_Get (TQ3StorageObject storage,
                               Handle *handle,
                               unsigned long *validSize);

TQ3Status Q3HandleStorage_Set (TQ3StorageObject storage,
                               Handle handle,
                               unsigned long validSize);
```

Creating and Accessing Macintosh Storage Objects

```
TQ3StorageObject Q3MacintoshStorage_New (
                               short fsRefNum);

TQ3Status Q3MacintoshStorage_Get (
                               TQ3StorageObject storage, short *fsRefNum);

TQ3Status Q3MacintoshStorage_Set (
                               TQ3StorageObject storage, short fsRefNum);

TQ3ObjectType Q3MacintoshStorage_GetType (
                               TQ3StorageObject storage);
```

Creating and Accessing FSSpec Storage Objects

```
TQ3StorageObject Q3FSSpecStorage_New (
                               const FSSpec *fs);

TQ3Status Q3FSSpecStorage_Get (TQ3StorageObject storage, FSSpec *fs);

TQ3Status Q3FSSpecStorage_Set (TQ3StorageObject storage, const FSSpec *fs);
```

Creating and Accessing UNIX Storage Objects

```
TQ3StorageObject Q3UnixStorage_New (
                               FILE *stdFile);

TQ3Status Q3UnixStorage_Get   (TQ3StorageObject storage, FILE **stdFile);
```

```
TQ3Status Q3UnixStorage_Set   (TQ3StorageObject storage, FILE *stdFile);

TQ3ObjectType Q3UnixStorage_GetType (
                              TQ3StorageObject storage);
```

Creating and Accessing UNIX Path Name Storage Objects

```
TQ3StorageObject Q3UnixPathStorage_New (
                              const char *pathName);

TQ3Status Q3UnixPathStorage_Get (
                              TQ3StorageObject storage,
                              char *pathName);

TQ3Status Q3UnixPathStorage_Set (
                              TQ3StorageObject storage,
                              const char *pathName);
```

Errors

```
kQ3ErrorAccessRestricted
kQ3ErrorBadFormatString
kQ3ErrorInvalidName
kQ3ErrorStorageInUse
kQ3ErrorStorageAlreadyOpen
kQ3ErrorStorageNotOpen
kQ3ErrorStorageIsOpen
```

File Objects

Contents

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” earlier in this book. 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 17-12 provides a complete description of file objects and the routines you can use to create and manipulate them.

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 *3D Metafile Reference* for complete information about the structure of the QuickDraw 3D Object Metafile. ◆

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 17-11 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.

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

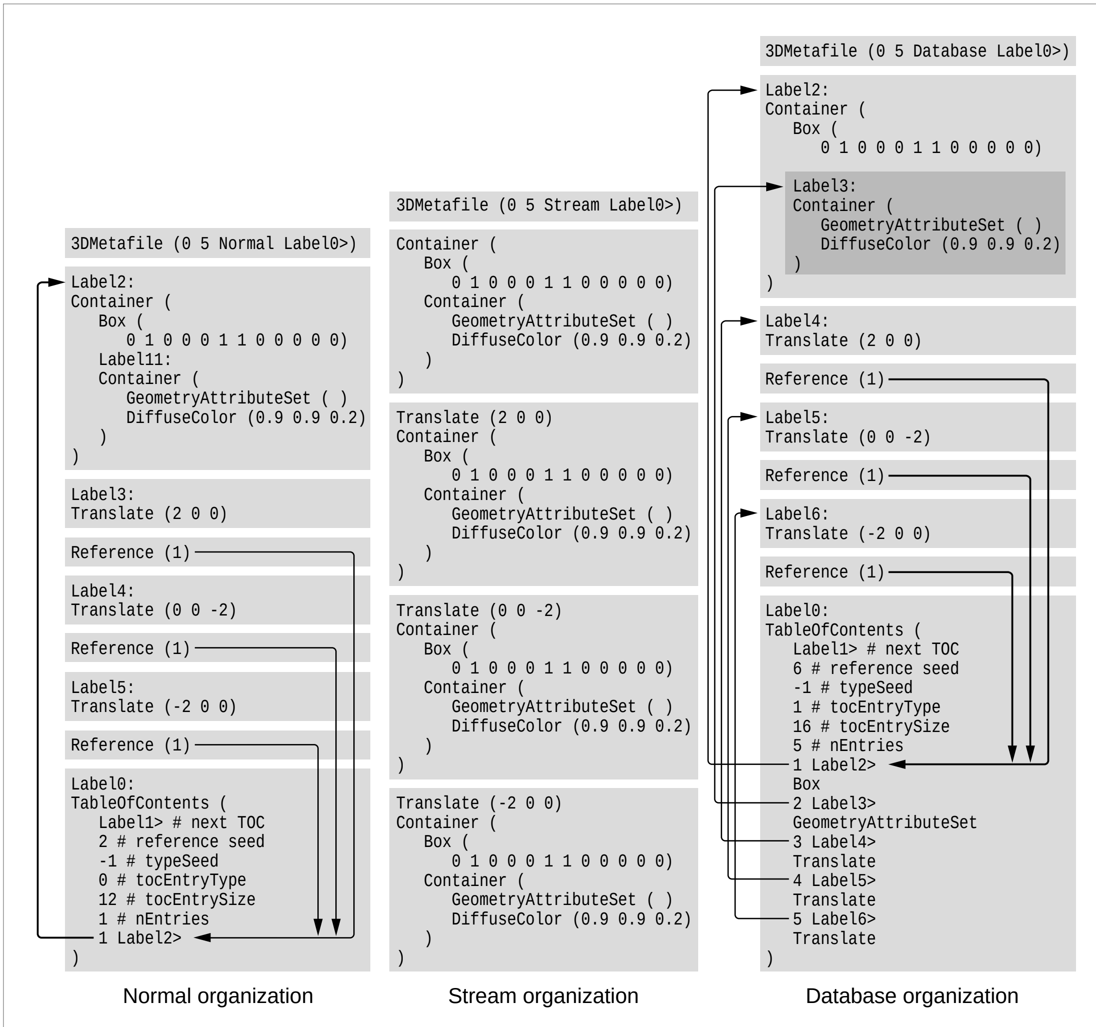
Disk-based metafile data, whether a text file or a binary file, should be contained in a file of type '3DMF'. ▲

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*.

Figure 17-1 Types of file objects



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.

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. This section describes how to perform these tasks.

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;
   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);
```

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.

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" on page 17-12 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" on page 17-12 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" on page 17-12 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" on page 17-65 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 17-69 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.

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.

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.

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.

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)`.

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.

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.

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" on page 12-9 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" on page 12-9 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.

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.

TQ3ObjectReadDataMethod

You can define a method to read an object of your custom type and any attached subobjects from a file object.

```
typedef TQ3Status (*TQ3ObjectReadDataMethod) (
                        TQ3Object parentObject,
                        TQ3FileObject file);
```

`parentObject`
An object to attach your custom data to.

`file` A file object.

DESCRIPTION

Your `TQ3ObjectReadDataMethod` function should read an object and any attached subobjects from the current location in the file object specified by the `file` parameter and attach that object to the object specified by the `parentObject` parameter. If the object read is a custom element (or a custom attribute), you should allocate space on the stack and call `Q3Set_Add` (or `Q3AttributeSet_Add`) on the object specified by the `parentObject` parameter, which is a set (or an attribute set). If the object read is not an element, you should attach your custom data to the object specified by the `parentObject` parameter.

On entrance to your custom read method, you should read the custom object data using the primitive data type _Read functions described in "Reading and Writing File Data," beginning on page 17-27 (for example, Q3Uns64_Read and Q3Point3D_Read). In general, you know the structure of the custom object data, so you can stop reading when you've read an entire object. Alternatively, you can read data until the function Q3File_IsEndOfData returns kQ3True.

Once you've read the custom object data, you should read any subobjects attached to the object. Because a metafile object has subobjects only if it is in a container, you can use the Q3File_IsEndOfContainer function to determine whether there are any subobjects (if Q3File_IsEndOfContainer returns kQ3False, there are subobjects to read). If you have created an object, pass it to Q3File_IsEndOfContainer so that the subobjects with automatic attachment methods can be attached to your object. Otherwise, pass the value NULL to Q3File_IsEndOfContainer to have all subobjects returned to you. Note that when you call Q3File_IsEndOfContainer, all unread parent object data is skipped and a warning is issued.

At this point, you can use the functions that retrieve subobjects (for example, Q3File_GetNextObjectType and Q3File_ReadObject) to iterate through the subobjects until Q3File_IsEndOfContainer returns kQ3True.

RESULT CODES

Your TQ3ObjectReadDataMethod function should return kQ3Success if it is successful and kQ3Failure otherwise.

TQ3ObjectTraverseMethod

You can define a method to traverse a custom object and write its data and the data of any attached subobjects to a file.

```
typedef TQ3Status (*TQ3ObjectTraverseMethod) (
                    TQ3Object object,
                    TQ3FileObject file);
```

object	A QuickDraw 3D object.
file	A file object.

DESCRIPTION

Your TQ3ObjectTraverseMethod method should perform a number of operations necessary to write the object specified by the object parameter, as well as any subobjects attached to it, to the file specified by the file parameter.

First, your traverse method should determine whether the specified object should be written to the file. It's possible that you won't want to write certain types of custom objects or certain types of data associated with a custom object. If you decide not to write the specified object and its subobjects to the file, your traverse method should return kQ3Success without calling any _Submit functions.

Next, you should calculate the size on disk of your custom object. This size must be aligned on a 4-byte boundary. Then you should retrieve whatever view state information you need to preserve. The state of the view is not valid in your custom object write method, but it is valid in your traverse method; if you need view state information in your write method, you can pass a temporary buffer to it.

Once you've preserved whatever view state information you need, you should submit your data by calling Q3View_SubmitWriteData. Then you should submit subobjects by calling the appropriate _Submit functions. You must call Q3View_SubmitWriteData before calling _Submit functions to submit any subobjects.

RESULT CODES

Your TQ3ObjectTraverseMethod function should return kQ3Success if it is successful and kQ3Failure otherwise.

TQ3ObjectWriteMethod

You can define a method to write an object of your custom type to a file object.

```
typedef TQ3Status (*TQ3ObjectWriteMethod) (
                    const void *object,
                    TQ3FileObject file);
```

object	A QuickDraw 3D object.
file	A file object.

DESCRIPTION

Your `TQ3ObjectWriteMethod` function should write the root object data of the object specified by the `object` parameter, starting at the current location in the file object specified by the `file` parameter. You should use the primitive data type `_Write` functions described in "Reading and Writing File Data," beginning on page 17-27 (for example, `Q3Uns64_Write` and `Q3Point3D_Write`).

RESULT CODES

Your `TQ3ObjectWriteMethod` function should return `kQ3Success` if it is successful and `kQ3Failure` otherwise.

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.

SPECIAL CONSIDERATIONS

You must not call any QuickDraw 3D routines inside your idle method.

Summary of File Objects

C Summary

Constants

```
typedef enum TQ3FileModeMasks {
    kQ3FileModeNormal                   = 0,
    kQ3FileModeStream                   = 1 << 0,
    kQ3FileModeDatabase                 = 1 << 1,
    kQ3FileModeText                     = 1 << 2
} TQ3FileModeMasks;

#define kQ3OldVersion                                   Q3FileVersion(0,2)
#define kQ3CurrentVersion                               Q3FileVersion(0,5)

#define kQ3StringMaximumLength                          1024
```

Unknown Object Types

```
#define kQ3UnknownTypeBinary                Q3_OBJECT_TYPE('u','k','b','n')
#define kQ3UnknownTypeText                  Q3_OBJECT_TYPE('u','k','t','x')
```

Data Types

Basic Types

```
typedef unsigned long               TQ3FileVersion;

typedef unsigned long               TQ3FileMode;
```

```
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 {
   unsigned long             hi;
   unsigned long             lo;
} TQ3Uns64;                                 /*8-byte unsigned integer*/

typedef struct TQ3Int64 {
   signed long               hi;
   unsigned long             lo;
} TQ3Uns64;                                 /*8-byte signed integer*/

typedef float                TQ3Float32;  /*4-byte floating-pt number*/
typedef double               TQ3Float64;  /*8-byte floating-pt number*/

typedef TQ3Uns32             TQ3Size;
```

Unknown Object Data Types

```
typedef struct TQ3UnknownTextData {
   char                         *objectName;       /*'\0' terminated*/
   char                         *contents;         /*'\0' terminated*/
} TQ3UnknownTextData;

typedef struct TQ3UnknownBinaryData {
   TQ3ObjectType                objectType;
   unsigned long                size;
   TQ3Endian                    byteOrder;
   char                         *contents;
} TQ3UnknownBinaryData;
```

File Objects Routines

Creating File Objects

```
TQ3FileObject Q3File_New      (void);
```

Attaching File Objects to Storage Objects

```
TQ3Status Q3File_GetStorage   (TQ3FileObject file,
                               TQ3StorageObject *storage);

TQ3Status Q3File_SetStorage   (TQ3FileObject file, TQ3StorageObject storage);
```

Accessing File Objects

```
TQ3Status Q3File_OpenRead     (TQ3FileObject file, TQ3FileMode *mode);

TQ3Status Q3File_OpenWrite    (TQ3FileObject file, TQ3FileMode mode);

TQ3Status Q3File_IsOpen       (TQ3FileObject file, TQ3Boolean *isOpen);

TQ3Status Q3File_Close        (TQ3FileObject file);

TQ3Status Q3File_Cancel       (TQ3FileObject file);

TQ3Status Q3File_GetMode      (TQ3FileObject file, TQ3FileMode *fileMode);

TQ3Status Q3File_GetVersion   (TQ3FileObject file, TQ3FileVersion *version);
```

Accessing Objects Directly

```
TQ3ObjectType Q3File_GetNextObjectType (
                               TQ3FileObject file);

TQ3Boolean Q3File_IsNextObjectOfType (
                               TQ3FileObject file,
                               TQ3ObjectType ofType);

TQ3Object Q3File_ReadObject   (TQ3FileObject file);

TQ3Status Q3File_SkipObject   (TQ3FileObject file);

TQ3Boolean Q3File_IsEndOfFile (TQ3FileObject file);
```

Setting Idle Methods

```
TQ3Status Q3File_SetIdleMethod(TQ3FileObject file,
                               TQ3FileIdleMethod idle,
                               const void *idleData);
```

Reading and Writing File Subobjects

```
TQ3Boolean Q3File_IsEndOfData (TQ3FileObject file);

TQ3Boolean Q3File_IsEndOfContainer (
                               TQ3FileObject file,
                               TQ3Object rootObject);
```

Reading and Writing File Data

```
TQ3Status Q3Uns8_Read         (TQ3Uns8 *data, TQ3FileObject file);

TQ3Status Q3Uns8_Write        (const TQ3Uns8 data, TQ3FileObject file);

TQ3Status Q3Uns16_Read        (TQ3Uns16 *data, TQ3FileObject file);

TQ3Status Q3Uns16_Write       (const TQ3Uns16 data, TQ3FileObject file);

TQ3Status Q3Uns32_Read        (TQ3Uns32 *data, TQ3FileObject file);

TQ3Status Q3Uns32_Write       (const TQ3Uns32 data, TQ3FileObject file);

TQ3Status Q3Int32_Read        (TQ3Int32 *data, TQ3FileObject file);

TQ3Status Q3Int32_Write       (const TQ3Int32 data, TQ3FileObject file);

TQ3Status Q3Uns64_Read        (TQ3Uns64 *data, TQ3FileObject file);

TQ3Status Q3Uns64_Write       (const TQ3Uns64 data, TQ3FileObject file);

TQ3Status Q3Float32_Read      (TQ3Float32 *data, TQ3FileObject file);

TQ3Status Q3Float32_Write     (const TQ3Float32 data, TQ3FileObject file);

TQ3Status Q3Float64_Read      (TQ3Float64 *data, TQ3FileObject file);

TQ3Status Q3Float64_Write     (const TQ3Float64 data, TQ3FileObject file);

TQ3Size Q3Size_Pad            (TQ3Size size);
```

```
TQ3Status Q3String_Read        (char *data,
                                unsigned long *length,
                                TQ3FileObject file);

TQ3Status Q3String_Write       (const char *data, TQ3FileObject file);

TQ3Status Q3RawData_Read       (unsigned char *data,
                                unsigned long size,
                                TQ3FileObject file);

TQ3Status Q3RawData_Write      (const unsigned char *data,
                                unsigned long size,
                                TQ3FileObject file);

TQ3Status Q3Point2D_Read       (TQ3Point2D *point2D, TQ3FileObject file);

TQ3Status Q3Point2D_Write      (const TQ3Point2D *point2D,
                                TQ3FileObject file);

TQ3Status Q3Point3D_Read       (TQ3Point3D *point3D, TQ3FileObject file);

TQ3Status Q3Point3D_Write      (const TQ3Point3D *point3D,
                                TQ3FileObject file);

TQ3Status Q3RationalPoint3D_Read (
                                TQ3RationalPoint3D *point3D,
                                TQ3FileObject file);

TQ3Status Q3RationalPoint3D_Write (
                                const TQ3RationalPoint3D *point3D,
                                TQ3FileObject file);

TQ3Status Q3RationalPoint4D_Read (
                                TQ3RationalPoint4D *point4D,
                                TQ3FileObject file);

TQ3Status Q3RationalPoint4D_Write (
                                const TQ3RationalPoint4D *point4D,
                                TQ3FileObject file);

TQ3Status Q3Vector2D_Read      (TQ3Vector2D *vector2D, TQ3FileObject file);
```

```
TQ3Status Q3Vector2D_Write    (const TQ3Vector2D *vector2D,
                               TQ3FileObject file);

TQ3Status Q3Vector3D_Read     (TQ3Vector3D *vector3D, TQ3FileObject file);

TQ3Status Q3Vector3D_Write    (const TQ3Vector3D *vector3D,
                               TQ3FileObject file);

TQ3Status Q3Matrix4x4_Read    (TQ3Matrix4x4 *matrix4x4,
                               TQ3FileObject file);

TQ3Status Q3Matrix4x4_Write   (const TQ3Matrix4x4 *matrix4x4,
                               TQ3FileObject file);

TQ3Status Q3Tangent2D_Read    (TQ3Tangent2D *tangent2D,
                               TQ3FileObject file);

TQ3Status Q3Tangent2D_Write   (const TQ3Tangent2D *tangent2D,
                               TQ3FileObject file);

TQ3Status Q3Tangent3D_Read    (TQ3Tangent3D *tangent3D,
                               TQ3FileObject file);

TQ3Status Q3Tangent3D_Write   (const TQ3Tangent3D *tangent3D,
                               TQ3FileObject file);

TQ3Status Q3Comment_Write     (char *comment, TQ3FileObject file);
```

Managing Unknown Objects

```
TQ3ObjectType Q3Unknown_GetType (
                               TQ3UnknownObject unknownObject);

TQ3Status Q3Unknown_GetDirtyState (
                               TQ3UnknownObject unknownObject,
                               TQ3Boolean *isDirty);

TQ3Status Q3Unknown_SetDirtyState (
                               TQ3UnknownObject unknownObject,
                               TQ3Boolean isDirty);
```

```
TQ3Status Q3UnknownText_GetData (
                              TQ3UnknownObject unknownObject,
                              TQ3UnknownTextData *unknownTextData);

TQ3Status Q3UnknownText_EmptyData (
                              TQ3UnknownTextData *unknownTextData);

TQ3Status Q3UnknownBinary_GetData (
                              TQ3UnknownObject unknownObject,
                              TQ3UnknownBinaryData *unknownBinaryData);

TQ3Status Q3UnknownBinary_EmptyData (
                              TQ3UnknownBinaryData *unknownBinaryData);
```

Managing View Hints Objects

```
TQ3ViewHintsObject Q3ViewHints_New (
                              TQ3ViewObject view);

TQ3Status Q3ViewHints_GetRenderer (
                              TQ3ViewHintsObject viewHints,
                              TQ3RendererObject *renderer);

TQ3Status Q3ViewHints_SetRenderer (
                              TQ3ViewHintsObject viewHints,
                              TQ3RendererObject renderer);

TQ3Status Q3ViewHints_GetCamera (
                              TQ3ViewHintsObject viewHints,
                              TQ3CameraObject *camera);

TQ3Status Q3ViewHints_SetCamera (
                              TQ3ViewHintsObject viewHints,
                              TQ3CameraObject camera);

TQ3Status Q3ViewHints_GetLightGroup (
                              TQ3ViewHintsObject viewHints,
                              TQ3GroupObject *lightGroup);
```

```
TQ3Status Q3ViewHints_SetLightGroup (
                              TQ3ViewHintsObject viewHints,
                              TQ3GroupObject lightGroup);

TQ3Status Q3ViewHints_GetAttributeSet (
                              TQ3ViewHintsObject viewHints,
                              TQ3AttributeSet *attributeSet);

TQ3Status Q3ViewHints_SetAttributeSet (
                              TQ3ViewHintsObject viewHints,
                              TQ3AttributeSet attributeSet);

TQ3Status Q3ViewHints_GetDimensionsState (
                              TQ3ViewHintsObject viewHints,
                              TQ3Boolean *isValid);

TQ3Status Q3ViewHints_SetDimensionsState (
                              TQ3ViewHintsObject viewHints,
                              TQ3Boolean isValid);

TQ3Status Q3ViewHints_GetDimensions (
                              TQ3ViewHintsObject viewHints,
                              unsigned long *width,
                              unsigned long *height);

TQ3Status Q3ViewHints_SetDimensions (
                              TQ3ViewHintsObject viewHints,
                              unsigned long width,
                              unsigned long height);

TQ3Status Q3ViewHints_GetMaskState (
                              TQ3ViewHintsObject viewHints,
                              TQ3Boolean *isValid);

TQ3Status Q3ViewHints_SetMaskState (
                              TQ3ViewHintsObject viewHints,
                              TQ3Boolean isValid);

TQ3Status Q3ViewHints_GetMask (TQ3ViewHintsObject viewHints,
                              TQ3Bitmap *mask);
```

```
TQ3Status Q3ViewHints_SetMask (TQ3ViewHintsObject viewHints,
                               const TQ3Bitmap *mask);

TQ3Status Q3ViewHints_GetClearImageMethod (
                               TQ3ViewHintsObject viewHints,
                               TQ3DrawContextClearImageMethod *clearMethod);

TQ3Status Q3ViewHints_SetClearImageMethod (
                               TQ3ViewHintsObject viewHints,
                               TQ3DrawContextClearImageMethod clearMethod);

TQ3Status Q3ViewHints_GetClearImageColor (
                               TQ3ViewHintsObject viewHints,
                               TQ3ColorARGB *color);

TQ3Status Q3ViewHints_SetClearImageColor (
                               TQ3ViewHintsObject viewHints,
                               const TQ3ColorARGB *color);
```

Version Macros

```
#define Q3FileVersion(majorVersion, minorVersion)                  \
          (TQ3FileVersion) ((((TQ3Uns32) majorVersion & 0xFFFF) << 16) | \
                            ((TQ3Uns32) minorVersion & 0xFFFF))
```

Application-Defined Routines

```
typedef TQ3Status (*TQ3ObjectReadDataMethod) (
                               TQ3Object parentObject,
                               TQ3FileObject file);

typedef TQ3Status (*TQ3ObjectTraverseMethod) (
                               TQ3Object object,
                               TQ3FileObject file);

typedef TQ3Status (*TQ3ObjectWriteMethod) (
                               const void *object,
                               TQ3FileObject file);
```

```
typedef TQ3Status (*TQ3FileIdleMethod) (
                              TQ3FileObject file,
                              const void *idlerData);
```

Errors, Warnings, and Notices

```
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
kQ3ErrorSizeMismatch
kQ3ErrorStringExceedsMaximumLength
kQ3ErrorNonUniqueLabel
kQ3ErrorUnmatchedEndGroup
kQ3WarningFilePointerResolutionFailed
kQ3WarningStringExceedsMaximumLength
kQ3NoticeFileAliasWasChanged
```

QuickDraw 3D Pointing Device Manager

Contents

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 QuickDraw 3D 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 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.

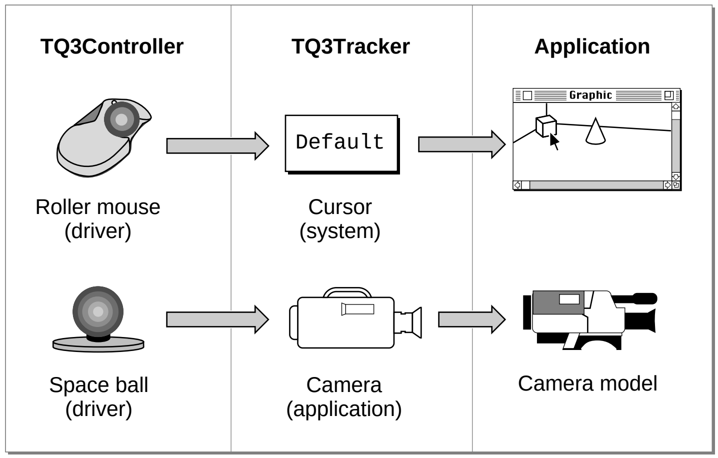
Controllers

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. ◆

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, cursors, and color schemes.

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 18-47 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 18-48 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 18-15) 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" on page 18-11 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 18-50 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 18-18 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 18-19 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` on page 18-34 for information on setting the notify function of a tracker.

Summary of the QuickDraw 3D Pointing Device Manager

C Summary

Constants

```
#define kQ3ControllerSetChannelMaxDataSize              256
```

Data Types

Controller Data Types

```
typedef struct TQ3ControllerData {
    char                            *signature;
    unsigned long                   valueCount;
    unsigned long                   channelCount;
    TQ3ChannelGetMethod             channelGetMethod;
    TQ3ChannelSetMethod             channelSetMethod;
} TQ3ControllerData;

typedef void                                        *TQ3ControllerRef;
```

QuickDraw 3D Pointing Device Manager Routines

Creating and Managing Controllers

```
TQ3ControllerRef Q3Controller_New (
                                    const TQ3ControllerData *controllerData);
```

```
TQ3Status Q3Controller_GetListChanged (
                              TQ3Boolean *listChanged,
                              unsigned long *serialNumber);

TQ3Status Q3Controller_Next   (TQ3ControllerRef controllerRef,
                              TQ3ControllerRef *nextControllerRef);

TQ3Status Q3Controller_Decommission (
                              TQ3ControllerRef controllerRef);

TQ3Status Q3Controller_GetActivation (
                              TQ3ControllerRef controllerRef,
                              TQ3Boolean *active);

TQ3Status Q3Controller_SetActivation (
                              TQ3ControllerRef controllerRef,
                              TQ3Boolean active);

TQ3Status Q3Controller_GetSignature (
                              TQ3ControllerRef controllerRef,
                              char *signature,
                              unsigned long numChars);

TQ3Status Q3Controller_GetChannel (
                              TQ3ControllerRef controllerRef,
                              unsigned long channel,
                              void *data,
                              unsigned long *dataSize);

TQ3Status Q3Controller_SetChannel (
                              TQ3ControllerRef controllerRef,
                              unsigned long channel,
                              const void *data,
                              unsigned long dataSize);

TQ3Status Q3Controller_GetValueCount (
                              TQ3ControllerRef controllerRef,
                              unsigned long *valueCount);
```

```
TQ3Status Q3Controller_SetTracker (
                              TQ3ControllerRef controllerRef,
                              TQ3TrackerObject tracker);

TQ3Status Q3Controller_HasTracker (
                              TQ3ControllerRef controllerRef,
                              TQ3Boolean *hasTracker);

TQ3Status Q3Controller_Track2DCursor (
                              TQ3ControllerRef controllerRef,
                              TQ3Boolean *track2DCursor);

TQ3Status Q3Controller_Track3DCursor (
                              TQ3ControllerRef controllerRef,
                              TQ3Boolean *track3DCursor);

TQ3Status Q3Controller_GetButtons (
                              TQ3ControllerRef controllerRef,
                              unsigned long *buttons);

TQ3Status Q3Controller_SetButtons (
                              TQ3ControllerRef controllerRef,
                              unsigned long buttons);

TQ3Status Q3Controller_GetTrackerPosition (
                              TQ3ControllerRef controllerRef,
                              TQ3Point3D *position);

TQ3Status Q3Controller_SetTrackerPosition (
                              TQ3ControllerRef controllerRef,
                              const TQ3Point3D *position);

TQ3Status Q3Controller_MoveTrackerPosition (
                              TQ3ControllerRef controllerRef,
                              const TQ3Vector3D *delta);

TQ3Status Q3Controller_GetTrackerOrientation (
                              TQ3ControllerRef controllerRef,
                              TQ3Quaternion *orientation);
```

```
TQ3Status Q3Controller_SetTrackerOrientation (
                              TQ3ControllerRef controllerRef,
                              const TQ3Quaternion *orientation);

TQ3Status Q3Controller_MoveTrackerOrientation (
                              TQ3ControllerRef controllerRef,
                              const TQ3Quaternion *delta);

TQ3Status Q3Controller_GetValues (
                              TQ3ControllerRef controllerRef,
                              unsigned long valueCount,
                              float *values,
                              TQ3Boolean *changed,
                              unsigned long *serialNumber);

TQ3Status Q3Controller_SetValues (
                              TQ3ControllerRef controllerRef,
                              const float *values,
                              unsigned long valueCount);
```

Managing Controller States

```
TQ3ControllerStateObject Q3ControllerState_New (
                              TQ3ControllerRef controllerRef);

TQ3Status Q3ControllerState_SaveAndReset (
                              TQ3ControllerStateObject
                              controllerStateObject);

TQ3Status Q3ControllerState_Restore (
                              TQ3ControllerStateObject
                              controllerStateObject);
```

Creating and Managing Trackers

```
TQ3TrackerObject Q3Tracker_New(TQ3TrackerNotifyFunc notifyFunc);
```

```
TQ3Status Q3Tracker_GetNotifyThresholds (
                               TQ3TrackerObject trackerObject,
                               float *positionThresh,
                               float *orientationThresh);

TQ3Status Q3Tracker_SetNotifyThresholds (
                               TQ3TrackerObject trackerObject,
                               float positionThresh,
                               float orientationThresh);

TQ3Status Q3Tracker_GetActivation (
                               TQ3TrackerObject trackerObject,
                               TQ3Boolean *active);

TQ3Status Q3Tracker_SetActivation (
                               TQ3TrackerObject trackerObject,
                               TQ3Boolean active);

TQ3Status Q3Tracker_GetEventCoordinates (
                               TQ3TrackerObject trackerObject,
                               unsigned long timeStamp,
                               unsigned long *buttons,
                               TQ3Point3D *position,
                               TQ3Quaternion *orientation);

TQ3Status Q3Tracker_SetEventCoordinates (
                               TQ3TrackerObject trackerObject,
                               unsigned long timeStamp,
                               unsigned long buttons,
                               const TQ3Point3D *position,
                               const TQ3Quaternion *orientation);

TQ3Status Q3Tracker_GetButtons(TQ3TrackerObject trackerObject,
                               unsigned long *buttons);
```

```
TQ3Status Q3Tracker_ChangeButtons (
                              TQ3TrackerObject trackerObject,
                              TQ3ControllerRef controllerRef,
                              unsigned long buttons,
                              unsigned long buttonMask);

TQ3Status Q3Tracker_GetPosition (
                              TQ3TrackerObject trackerObject,
                              TQ3Point3D *position,
                              TQ3Vector3D *delta,
                              TQ3Boolean *changed,
                              unsigned long *serialNumber);

TQ3Status Q3Tracker_SetPosition (
                              TQ3TrackerObject trackerObject,
                              TQ3ControllerRef controllerRef,
                              const TQ3Point3D *position);

TQ3Status Q3Tracker_MovePosition (
                              TQ3TrackerObject trackerObject,
                              TQ3ControllerRef controllerRef,
                              const TQ3Vector3D *delta);

TQ3Status Q3Tracker_GetOrientation (
                              TQ3TrackerObject trackerObject,
                              TQ3Quaternion *orientation,
                              TQ3Quaternion *delta,
                              TQ3Boolean *changed,
                              unsigned long *serialNumber);

TQ3Status Q3Tracker_SetOrientation (
                              TQ3TrackerObject trackerObject,
                              TQ3ControllerRef controllerRef,
                              const TQ3Quaternion *orientation);

TQ3Status Q3Tracker_MoveOrientation (
                              TQ3TrackerObject trackerObject,
                              TQ3ControllerRef controllerRef,
                              const TQ3Quaternion *delta);
```

Application-Defined Routines

```
typedef TQ3Status (*TQ3ChannelGetMethod) (
                              TQ3ControllerRef controllerRef,
                              unsigned long channel,
                              void *data,
                              unsigned long *dataSize);

typedef TQ3Status (*TQ3ChannelSetMethod) (
                              TQ3ControllerRef controllerRef,
                              unsigned long channel,
                              const void *data,
                              unsigned long dataSize);

typedef TQ3Status (*TQ3TrackerNotifyFunc) (
                              TQ3TrackerObject trackerObject,
                              TQ3ControllerRef controllerRef);
```

Error Manager

Contents

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.

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. Whether future calls actually do fail depends on whether those calls are suitably related to the call that generated the fatal error. For example, even if a fatal error occurs during rendering, you might still be able to perform file operations (perhaps to save the data that couldn't be rendered).
- 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 19-11 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 19-12 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 19-13 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. ▲

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.

You should call `Q3MacintoshError_Get` when `Q3Error_Get` returns `kQ3ErrorMacintoshError`, and you should call `Q3UnixError_Get` when `Q3Error_Get` returns `kQ3ErrorUnixError`.

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.

Q3UnixError_Get

You can use the `Q3UnixError_Get` function to get the most recent and the oldest unreported error generated by the UNIX operating system.

```
int Q3UnixError_Get (int *firstUnixError);
```

`firstUnixError`
On exit, the first unreported error from a UNIX routine.

DESCRIPTION

The `Q3UnixError_Get` function returns, as its function result, the most recent error generated by the UNIX kernel. `Q3UnixError_Get` also returns, in the `firstUnixError` parameter, the oldest unreported error that occurred during a UNIX operating system routine.

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.

Summary of the Error Manager

C Summary

Data Types

```
typedef long                                                    TQ3Error;

typedef long                                                    TQ3Warning;

typedef long                                                    TQ3Notice;
```

Error Manager Routines

Registering Error, Warning, and Notice Callback Routines

```
TQ3Status Q3Error_Register    (TQ3ErrorMethod errorPost, long reference);

TQ3Status Q3Warning_Register  (TQ3WarningMethod warningPost, long reference);

TQ3Status Q3Notice_Register   (TQ3NoticeMethod noticePost, long reference);
```

Determining Whether an Error is Fatal

```
TQ3Boolean Q3Error_IsFatalError (
                               TQ3Error error);
```

Getting Errors, Warnings, and Notices Directly

```
TQ3Error Q3Error_Get          (TQ3Error *firstError);

TQ3Warning Q3Warning_Get      (TQ3Warning *firstWarning);

TQ3Notice Q3Notice_Get        (TQ3Notice *firstNotice);
```

Getting Operating System Errors

```
OSErr Q3MacintoshError_Get    (OSErr *firstMacErr);

int Q3UnixError_Get           (int *firstUnixError);
```

Application-Defined Routines

```
typedef void (*TQ3ErrorMethod)(TQ3Error firstError,
                               TQ3Error lastError,
                               long reference);

typedef void (*TQ3WarningMethod) (
                               TQ3Warning firstWarning,
                               TQ3Warning lastWarning,
                               long reference);

typedef void (*TQ3NoticeMethod) (
                               TQ3Notice firstNotice,
                               TQ3Notice lastNotice,
                               long reference);
```

Errors

kQ3ErrorUnixError	A UNIX operating system error
kQ3ErrorMacintoshError	A Macintosh Operating System error

QuickDraw 3D Mathematical Utilities

Contents

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 QuickDraw 3D 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.

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.

```
#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_SetRotateX

You can use the `Q3Quaternion_SetRotateX` function to configure a quaternion that rotates objects around the *x* axis.

```
TQ3Quaternion *Q3Quaternion_SetRotateX (
                        TQ3Quaternion *quaternion,
                        float angle);
```

`quaternion` A quaternion.

`angle` The desired angle of rotation around the *x* coordinate axis, in radians.

DESCRIPTION

The `Q3Quaternion_SetRotateX` 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_SetRotateY

You can use the `Q3Quaternion_SetRotateY` function to configure a quaternion that rotates objects around the *y* axis.

```
TQ3Quaternion *Q3Quaternion_SetRotateY (
                        TQ3Quaternion *quaternion,
                        float angle);
```

`quaternion` A quaternion.

`angle` The desired angle of rotation around the *y* coordinate axis, in radians.

DESCRIPTION

The Q3Quaternion_SetRotateY 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_SetRotateZ

You can use the Q3Quaternion_SetRotateZ function to configure a quaternion that rotates objects around the *z* axis.

```
TQ3Quaternion *Q3Quaternion_SetRotateZ (
                    TQ3Quaternion *quaternion,
                    float angle);
```

quaternion	A quaternion.
angle	The desired angle of rotation around the *z* coordinate axis, in radians.

DESCRIPTION

The Q3Quaternion_SetRotateZ 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_SetRotateXYZ

You can use the Q3Quaternion_SetRotateXYZ function to configure a quaternion having a specified rotation around the *x*, *y*, and *z* axes.

```
TQ3Quaternion *Q3Quaternion_SetRotateXYZ (
                    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_SetRotateXYZ` 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.

Summary of QuickDraw 3D Mathematical Utilities

C Summary

Constants

Real Zero Definition

```
#ifdef FLT_EPSILON
#   define kQ3RealZero           (FLT_EPSILON)
#else
#   define kQ3RealZero           ((float)1.19209290e-07)
#endif
```

Maximum Floating-Point Value

```
#ifdef FLT_MAX
#   define kQ3MaxFloat           (FLT_MAX)
#else
#   define kQ3MaxFloat           ((float)3.40282347e+38)
#endif
```

Pi

```
#define kQ3Pi                    (3.1415926535898)
#define kQ32Pi                   (2.0 * kQ3Pi)
```

Data Types

Bounding Boxes and Spheres

```
typedef struct TQ3BoundingBox {
    TQ3Point3D                  min;
    TQ3Point3D                  max;
    TQ3Boolean                  isEmpty;
} TQ3BoundingBox;

typedef struct TQ3BoundingSphere {
    TQ3Point3D                  origin;
    float                       radius;
    TQ3Boolean                  isEmpty;
} TQ3BoundingSphere;
```

QuickDraw 3D Mathematical Utilities

Setting Points and Vectors

```
TQ3Point2D *Q3Point2D_Set     (TQ3Point2D *point2D, float x, float y);

TQ3Param2D *Q3Param2D_Set     (TQ3Param2D *param2D, float u, float v);

TQ3Point3D *Q3Point3D_Set     (TQ3Point3D *point3D,
                               float x,
                               float y,
                               float z);

TQ3RationalPoint3D *Q3RationalPoint3D_Set (
                               TQ3RationalPoint3D *point3D,
                               float x,
                               float y,
                               float w);
```

```
TQ3RationalPoint4D *Q3RationalPoint4D_Set (
                              TQ3RationalPoint4D *point4D,
                              float x,
                              float y,
                              float z,
                              float w);

TQ3PolarPoint *Q3PolarPoint_Set (
                              TQ3PolarPoint *polarPoint,
                              float r,
                              float theta);

TQ3SphericalPoint *Q3SphericalPoint_Set (
                              TQ3SphericalPoint *sphericalPoint,
                              float rho,
                              float theta,
                              float phi);

TQ3Vector2D *Q3Vector2D_Set  (TQ3Vector2D *vector2D,
                              float x,
                              float y);

TQ3Vector3D *Q3Vector3D_Set  (TQ3Vector3D *vector3D,
                              float x,
                              float y,
                              float z);
```

Converting Dimensions of Points and Vectors

```
TQ3Point3D *Q3Point2D_To3D   (const TQ3Point2D *point2D,
                              TQ3Point3D *result);

TQ3RationalPoint4D *Q3Point3D_To4D (
                              const TQ3Point3D *point3D,
                              TQ3RationalPoint4D *result);

TQ3Point2D *Q3RationalPoint3D_To2D (
                              const TQ3RationalPoint3D *point3D,
                              TQ3Point2D *result);
```

```
TQ3Point3D *Q3RationalPoint4D_To3D (
                              const TQ3RationalPoint4D *point4D,
                              TQ3Point3D *result);

TQ3Vector3D *Q3Vector2D_To3D  (const TQ3Vector2D *vector2D,
                              TQ3Vector3D *result);

TQ3Vector2D *Q3Vector3D_To2D  (const TQ3Vector3D *vector3D,
                              TQ3Vector2D *result);
```

Subtracting Points

```
TQ3Vector2D *Q3Point2D_Subtract (
                              const TQ3Point2D *p1,
                              const TQ3Point2D *p2,
                              TQ3Vector2D *result);

TQ3Vector2D *Q3Param2D_Subtract (
                              const TQ3Param2D *p1,
                              const TQ3Param2D *p2,
                              TQ3Vector2D *result);

TQ3Vector3D *Q3Point3D_Subtract (
                              const TQ3Point3D *p1,
                              const TQ3Point3D *p2,
                              TQ3Vector3D *result);
```

Calculating Distances Between Points

```
float Q3Point2D_Distance      (const TQ3Point2D *p1, const TQ3Point2D *p2);

float Q3Param2D_Distance      (const TQ3Param2D *p1, const TQ3Param2D *p2);

float Q3Point3D_Distance      (const TQ3Point3D *p1, const TQ3Point3D *p2);

float Q3RationalPoint3D_Distance (
                              const TQ3RationalPoint3D *p1,
                              const TQ3RationalPoint3D *p2);
```

```
float Q3RationalPoint4D_Distance (
                              const TQ3RationalPoint4D *p1,
                              const TQ3RationalPoint4D *p2);

float Q3Point2D_DistanceSquared(const TQ3Point2D *p1, const TQ3Point2D *p2);

float Q3Param2D_DistanceSquared(const TQ3Param2D *p1, const TQ3Param2D *p2);

float Q3Point3D_DistanceSquared(const TQ3Point3D *p1, const TQ3Point3D *p2);

float Q3RationalPoint3D_DistanceSquared (
                              const TQ3RationalPoint3D *p1,
                              const TQ3RationalPoint3D *p2);

float Q3RationalPoint4D_DistanceSquared (
                              const TQ3RationalPoint4D *p1,
                              const TQ3RationalPoint4D *p2);
```

Determining Point Relative Ratios

```
TQ3Point2D *Q3Point2D_RRatio  (const TQ3Point2D *p1,
                              const TQ3Point2D *p2,
                              float r1,
                              float r2,
                              TQ3Point2D *result);

TQ3Param2D *Q3Param2D_RRatio  (const TQ3Param2D *p1,
                              const TQ3Param2D *p2,
                              float r1,
                              float r2,
                              TQ3Param2D *result);

TQ3Point3D *Q3Point3D_RRatio  (const TQ3Point3D *p1,
                              const TQ3Point3D *p2,
                              float r1,
                              float r2,
                              TQ3Point3D *result);
```

```
TQ3RationalPoint4D *Q3RationalPoint4D_RRatio (
                              const TQ3RationalPoint4D *p1,
                              const TQ3RationalPoint4D *p2,
                              float r1,
                              float r2,
                              TQ3RationalPoint4D *result);
```

Adding and Subtracting Points and Vectors

```
TQ3Point2D *Q3Point2D_Vector2D_Add (
                              const TQ3Point2D *point2D,
                              const TQ3Vector2D *vector2D,
                              TQ3Point2D *result);

TQ3Param2D *Q3Param2D_Vector2D_Add (
                              const TQ3Param2D *param2D,
                              const TQ3Vector2D *vector2D,
                              TQ3Param2D *result);

TQ3Point3D *Q3Point3D_Vector3D_Add (
                              const TQ3Point3D *point3D,
                              const TQ3Vector3D *vector3D,
                              TQ3Point3D *result);

TQ3Point2D *Q3Point2D_Vector2D_Subtract (
                              const TQ3Point2D *point2D,
                              const TQ3Vector2D *vector2D,
                              TQ3Point2D *result);

TQ3Param2D *Q3Param2D_Vector2D_Subtract (
                              const TQ3Param2D *param2D,
                              const TQ3Vector2D *vector2D,
                              TQ3Param2D *result);

TQ3Point3D *Q3Point3D_Vector3D_Subtract (
                              const TQ3Point3D *point3D,
                              const TQ3Vector3D *vector3D,
                              TQ3Point3D *result);
```

Scaling Vectors

```
TQ3Vector2D *Q3Vector2D_Scale (const TQ3Vector2D *vector2D,
                               float scalar,
                               TQ3Vector2D *result);

TQ3Vector3D *Q3Vector3D_Scale (const TQ3Vector3D *vector3D,
                               float scalar,
                               TQ3Vector3D *result);
```

Determining the Lengths of Vectors

```
float Q3Vector2D_Length       (const TQ3Vector2D *vector2D);

float Q3Vector3D_Length       (const TQ3Vector3D *vector3D);
```

Normalizing Vectors

```
TQ3Vector2D *Q3Vector2D_Normalize (
                               const TQ3Vector2D *vector2D,
                               TQ3Vector2D *result);

TQ3Vector3D *Q3Vector3D_Normalize (
                               const TQ3Vector3D *vector3D,
                               TQ3Vector3D *result);
```

Adding and Subtracting Vectors

```
TQ3Vector2D *Q3Vector2D_Add   (const TQ3Vector2D *v1,
                               const TQ3Vector2D *v2,
                               TQ3Vector2D *result);

TQ3Vector3D *Q3Vector3D_Add   (const TQ3Vector3D *v1,
                               const TQ3Vector3D *v2,
                               TQ3Vector3D *result);

TQ3Vector2D *Q3Vector2D_Subtract (
                               const TQ3Vector2D *v1,
                               const TQ3Vector2D *v2,
                               TQ3Vector2D *result);
```

```
TQ3Vector3D *Q3Vector3D_Subtract (
                              const TQ3Vector3D *v1,
                              const TQ3Vector3D *v2,
                              TQ3Vector3D *result);
```

Determining Vector Cross Products

```
float Q3Vector2D_Cross        (const TQ3Vector2D *v1,
                               const TQ3Vector2D *v2);

TQ3Vector3D *Q3Vector3D_Cross (const TQ3Vector3D *v1,
                               const TQ3Vector3D *v2,
                               TQ3Vector3D *result);

TQ3Vector3D *Q3Point3D_CrossProductTri (
                               const TQ3Point3D *point1,
                               const TQ3Point3D *point2,
                               const TQ3Point3D *point3,
                               TQ3Vector3D *crossVector);
```

Determining Vector Dot Products

```
float Q3Vector2D_Dot          (const TQ3Vector2D *v1, const TQ3Vector2D *v2);

float Q3Vector3D_Dot          (const TQ3Vector3D *v1, const TQ3Vector3D *v2);
```

Transforming Points and Vectors

```
TQ3Vector2D *Q3Vector2D_Transform (
                               const TQ3Vector2D *vector2D,
                               const TQ3Matrix3x3 *matrix3x3,
                               TQ3Vector2D *result);

TQ3Vector3D *Q3Vector3D_Transform (
                               const TQ3Vector3D *vector3D,
                               const TQ3Matrix4x4 *matrix4x4,
                               TQ3Vector3D *result);
```

```
TQ3Point2D *Q3Point2D_Transform (
                              const TQ3Point2D *point2D,
                              const TQ3Matrix3x3 *matrix3x3,
                              TQ3Point2D *result);

TQ3Param2D *Q3Param2D_Transform (
                              const TQ3Param2D *param2D,
                              const TQ3Matrix3x3 *matrix3x3,
                              TQ3Param2D *result);

TQ3Point3D *Q3Point3D_Transform (
                              const TQ3Point3D *point3D,
                              const TQ3Matrix4x4 *matrix4x4,
                              TQ3Point3D *result);

TQ3RationalPoint4D *Q3RationalPoint4D_Transform (
                              const TQ3RationalPoint4D *point4D,
                              const TQ3Matrix4x4 *matrix4x4,
                              TQ3RationalPoint4D *result);

TQ3Status Q3Point3D_To3DTransformArray (
                              const TQ3Point3D *inVertex,
                              const TQ3Matrix4x4 *matrix,
                              TQ3Point3D *outVertex,
                              long numVertices,
                              unsigned long inStructSize,
                              unsigned long outStructSize);

TQ3Status Q3Point3D_To4DTransformArray (
                              const TQ3Point3D *inVertex,
                              const TQ3Matrix4x4 *matrix,
                              TQ3RationalPoint4D *outVertex,
                              long numVertices,
                              unsigned long inStructSize,
                              unsigned long outStructSize);
```

```
TQ3Status Q3RationalPoint4D_To4DTransformArray (
                              const TQ3RationalPoint4D *inVertex,
                              const TQ3Matrix4x4 *matrix,
                              TQ3RationalPoint4D *outVertex,
                              long numVertices,
                              unsigned long inStructSize,
                              unsigned long outStructSize);
```

Negating Vectors

```
TQ3Vector2D *Q3Vector2D_Negate(const TQ3Vector2D *vector2D,
                              TQ3Vector2D *result);

TQ3Vector3D *Q3Vector3D_Negate(const TQ3Vector3D *vector3D,
                              TQ3Vector3D *result);
```

Converting Points from Cartesian to Polar or Spherical Form

```
TQ3PolarPoint *Q3Point2D_ToPolar (
                              const TQ3Point2D *point2D,
                              TQ3PolarPoint *result);

TQ3Point2D *Q3PolarPoint_ToPoint2D (
                              const TQ3PolarPoint *polarPoint,
                              TQ3Point2D *result);

TQ3SphericalPoint *Q3Point3D_ToSpherical (
                              const TQ3Point3D *point3D,
                              TQ3SphericalPoint *result);

TQ3Point3D *Q3SphericalPoint_ToPoint3D (
                              const TQ3SphericalPoint *sphericalPoint,
                              TQ3Point3D *result);
```

Determining Point Affine Combinations

```
TQ3Point2D *Q3Point2D_AffineComb (
                              const TQ3Point2D *points2D,
                              const float *weights,
                              unsigned long nPoints,
                              TQ3Point2D *result);

TQ3Param2D *Q3Param2D_AffineComb (
                              const TQ3Param2D *params2D,
                              const float *weights,
                              unsigned long nPoints,
                              TQ3Param2D *result);

TQ3Point3D *Q3Point3D_AffineComb (
                              const TQ3Point3D *points3D,
                              const float *weights,
                              unsigned long nPoints,
                              TQ3Point3D *result);

TQ3RationalPoint3D *Q3RationalPoint3D_AffineComb (
                              const TQ3RationalPoint3D *points3D,
                              const float *weights,
                              unsigned long nPoints,
                              TQ3RationalPoint3D *result);

TQ3RationalPoint4D *Q3RationalPoint4D_AffineComb (
                              const TQ3RationalPoint4D *points4D,
                              const float *weights,
                              unsigned long nPoints,
                              TQ3RationalPoint4D *result);
```

Managing Matrices

```
TQ3Matrix3x3 *Q3Matrix3x3_Copy(const TQ3Matrix3x3 *matrix3x3,
                              TQ3Matrix3x3 *result);

TQ3Matrix4x4 *Q3Matrix4x4_Copy(const TQ3Matrix4x4 *matrix4x4,
                              TQ3Matrix4x4 *result);
```

```
TQ3Matrix3x3 *Q3Matrix3x3_SetIdentity (
                              TQ3Matrix3x3 *matrix3x3);

TQ3Matrix4x4 *Q3Matrix4x4_SetIdentity (
                              TQ3Matrix4x4 *matrix4x4);

TQ3Matrix3x3 *Q3Matrix3x3_Transpose (
                              const TQ3Matrix3x3 *matrix3x3,
                              TQ3Matrix3x3 *result);

TQ3Matrix4x4 *Q3Matrix4x4_Transpose (
                              const TQ3Matrix4x4 *matrix4x4,
                              TQ3Matrix4x4 *result);

TQ3Matrix3x3 *Q3Matrix3x3_Invert (
                              const TQ3Matrix3x3 *matrix3x3,
                              TQ3Matrix3x3 *result);

TQ3Matrix4x4 *Q3Matrix4x4_Invert (
                              const TQ3Matrix4x4 *matrix4x4,
                              TQ3Matrix4x4 *result);

TQ3Matrix3x3 *Q3Matrix3x3_Adjoint (
                              const TQ3Matrix3x3 *matrix3x3,
                              TQ3Matrix3x3 *result);

TQ3Matrix3x3 *Q3Matrix3x3_Multiply (
                              const TQ3Matrix3x3 *matrixA,
                              const TQ3Matrix3x3 *matrixB,
                              TQ3Matrix3x3 *result);

TQ3Matrix4x4 *Q3Matrix4x4_Multiply (
                              const TQ3Matrix4x4 *matrixA,
                              const TQ3Matrix4x4 *matrixB,
                              TQ3Matrix4x4 *result);

float Q3Matrix3x3_Determinant (const TQ3Matrix3x3 *matrix3x3);

float Q3Matrix4x4_Determinant (const TQ3Matrix4x4 *matrix4x4);
```

Setting Up Transformation Matrices

```
TQ3Matrix3x3 *Q3Matrix3x3_SetTranslate (
                              TQ3Matrix3x3 *matrix3x3,
                              float xTrans,
                              float yTrans);

TQ3Matrix3x3 *Q3Matrix3x3_SetScale (
                              TQ3Matrix3x3 *matrix3x3,
                              float xScale,
                              float yScale);

TQ3Matrix3x3 *Q3Matrix3x3_SetRotateAboutPoint (
                              TQ3Matrix3x3 *matrix3x3,
                              const TQ3Point2D *origin,
                              float angle);

TQ3Matrix4x4 *Q3Matrix4x4_SetTranslate (
                              TQ3Matrix4x4 *matrix4x4,
                              float xTrans,
                              float yTrans,
                              float zTrans);

TQ3Matrix4x4 *Q3Matrix4x4_SetScale (
                              TQ3Matrix4x4 *matrix4x4,
                              float xScale,
                              float yScale,
                              float zScale);

TQ3Matrix4x4 *Q3Matrix4x4_SetRotateAboutPoint (
                              TQ3Matrix4x4 *matrix4x4,
                              const TQ3Point3D *origin,
                              float xAngle,
                              float yAngle,
                              float zAngle);
```

```
TQ3Matrix4x4 *Q3Matrix4x4_SetRotateAboutAxis (
                              TQ3Matrix4x4 *matrix4x4,
                              const TQ3Point3D *origin,
                              const TQ3Vector3D *orientation,
                              float angle);

TQ3Matrix4x4 *Q3Matrix4x4_SetRotate_X (
                              TQ3Matrix4x4 *matrix4x4, float angle);

TQ3Matrix4x4 *Q3Matrix4x4_SetRotate_Y (
                              TQ3Matrix4x4 *matrix4x4, float angle);

TQ3Matrix4x4 *Q3Matrix4x4_SetRotate_Z (
                              TQ3Matrix4x4 *matrix4x4, float angle);

TQ3Matrix4x4 *Q3Matrix4x4_SetRotate_XYZ (
                              TQ3Matrix4x4 *matrix4x4,
                              float xAngle,
                              float yAngle,
                              float zAngle);

TQ3Matrix4x4 *Q3Matrix4x4_SetRotateVectorToVector (
                              TQ3Matrix4x4 *matrix4x4,
                              const TQ3Vector3D *v1,
                              const TQ3Vector3D *v2);

TQ3Matrix4x4 *Q3Matrix4x4_SetQuaternion (
                              TQ3Matrix4x4 *matrix,
                              const TQ3Quaternion *quaternion);
```

Utility Functions

```
#define Q3Math_DegreesToRadians(x)        ((x) * kQ3Pi / 180.0)

#define Q3Math_RadiansToDegrees(x)        ((x) * 180.0 / kQ3Pi)

#define Q3Math_Min(x,y)                   ((x) <= (y) ? (x) : (y))

#define Q3Math_Max(x,y)                   ((x) >= (y) ? (x) : (y))
```

Managing Quaternions

```
TQ3Quaternion *Q3Quaternion_Set (
                              TQ3Quaternion *quaternion,
                              float w,
                              float x,
                              float y,
                              float z);

TQ3Quaternion *Q3Quaternion_SetIdentity (
                              TQ3Quaternion *quaternion);

TQ3Quaternion *Q3Quaternion_Copy (
                              const TQ3Quaternion *quaternion,
                              TQ3Quaternion *result);

TQ3Boolean Q3Quaternion_IsIdentity (
                              const TQ3Quaternion *quaternion);

TQ3Quaternion *Q3Quaternion_Invert (
                              const TQ3Quaternion *quaternion,
                              TQ3Quaternion *result);

TQ3Quaternion *Q3Quaternion_Normalize (
                              const TQ3Quaternion *quaternion,
                              TQ3Quaternion *result);

float Q3Quaternion_Dot        (const TQ3Quaternion *q1,
                              const TQ3Quaternion *q2);

TQ3Quaternion *Q3Quaternion_Multiply (
                              const TQ3Quaternion *q1,
                              const TQ3Quaternion *q2,
                              TQ3Quaternion *result);

TQ3Quaternion *Q3Quaternion_SetRotateAboutAxis (
                              TQ3Quaternion *quaternion,
                              const TQ3Vector3D *axis,
                              float angle);
```

```
TQ3Quaternion *Q3Quaternion_SetRotateX (
                               TQ3Quaternion *quaternion,
                               float angle);

TQ3Quaternion *Q3Quaternion_SetRotateY (
                               TQ3Quaternion *quaternion,
                               float angle);

TQ3Quaternion *Q3Quaternion_SetRotateZ (
                               TQ3Quaternion *quaternion,
                               float angle);

TQ3Quaternion *Q3Quaternion_SetRotateXYZ (
                               TQ3Quaternion *quaternion,
                               float xAngle,
                               float yAngle,
                               float zAngle);

TQ3Quaternion *Q3Quaternion_SetMatrix (
                               TQ3Quaternion *quaternion,
                               const TQ3Matrix4x4 *matrix);

TQ3Quaternion *Q3Quaternion_SetRotateVectorToVector (
                               TQ3Quaternion *quaternion,
                               const TQ3Vector3D *v1,
                               const TQ3Vector3D *v2);

TQ3Quaternion *Q3Quaternion_MatchReflection (
                               const TQ3Quaternion *q1,
                               const TQ3Quaternion *q2,
                               TQ3Quaternion *result);

TQ3Quaternion *Q3Quaternion_InterpolateFast (
                               const TQ3Quaternion *q1,
                               const TQ3Quaternion *q2,
                               float t,
                               TQ3Quaternion *result);
```

```
TQ3Quaternion *Q3Quaternion_InterpolateLinear (
                              const TQ3Quaternion *q1,
                              const TQ3Quaternion *q2,
                              float t,
                              TQ3Quaternion *result) ;

TQ3Vector3D *Q3Vector3D_TransformQuaternion (
                              const TQ3Vector3D *vector,
                              const TQ3Quaternion *quaternion,
                              TQ3Vector3D *result);

TQ3Point3D *Q3Point3D_TransformQuaternion (
                              const TQ3Point3D *point,
                              const TQ3Quaternion *quaternion,
                              TQ3Point3D *result);
```

Managing Bounding Boxes

```
TQ3BoundingBox *Q3BoundingBox_Copy (
                              const TQ3BoundingBox *src,
                              TQ3BoundingBox *dest);

TQ3BoundingBox *Q3BoundingBox_Union (
                              const TQ3BoundingBox *v1,
                              const TQ3BoundingBox *v2,
                              TQ3BoundingBox *result);

TQ3BoundingBox *Q3BoundingBox_Set (
                              TQ3BoundingBox *bBox,
                              const TQ3Point3D *min,
                              const TQ3Point3D *max,
                              TQ3Boolean isEmpty);

TQ3BoundingBox *Q3BoundingBox_UnionPoint3D (
                              const TQ3BoundingBox *bBox,
                              const TQ3Point3D *pt3D,
                              TQ3BoundingBox *result);
```

```
TQ3BoundingBox *Q3BoundingBox_UnionRationalPoint4D (
                              const TQ3BoundingBox *bBox,
                              const TQ3RationalPoint4D *pt4D,
                              TQ3BoundingBox *result);

TQ3BoundingBox *Q3BoundingBox_SetFromPoints3D (
                              TQ3BoundingBox *bBox,
                              const TQ3Point3D *pts,
                              unsigned long nPts,
                              unsigned long structSize);

TQ3BoundingBox *Q3BoundingBox_SetFromRationalPoints4D (
                              TQ3BoundingBox *bBox,
                              const TQ3RationalPoint4D *pts,
                              unsigned long nPts,
                              unsigned long structSize);
```

Managing Bounding Spheres

```
TQ3BoundingSphere *Q3BoundingSphere_Copy (
                              const TQ3BoundingSphere *src,
                              TQ3BoundingSphere *dest);

TQ3BoundingSphere *Q3BoundingSphere_Union (
                              const TQ3BoundingSphere *s1,
                              const TQ3BoundingSphere *s2,
                              TQ3BoundingSphere *result);

TQ3BoundingSphere *Q3BoundingSphere_Set (
                              TQ3BoundingSphere *bSphere,
                              const TQ3Point3D *origin,
                              float radius,
                              TQ3Boolean isEmpty);

TQ3BoundingSphere *Q3BoundingSphere_UnionPoint3D (
                              const TQ3BoundingSphere *bSphere,
                              const TQ3Point3D *pt3D,
                              TQ3BoundingSphere *result);
```

```
TQ3BoundingSphere *Q3BoundingSphere_UnionRationalPoint4D (
                              const TQ3BoundingSphere *bSphere,
                              const TQ3RationalPoint4D *pt4D,
                              TQ3BoundingSphere *result);

TQ3BoundingSphere *Q3BoundingSphere_SetFromPoints3D (
                              TQ3BoundingSphere *bSphere,
                              const TQ3Point3D *pts,
                              unsigned long nPts,
                              unsigned long structSize);

TQ3BoundingSphere *Q3BoundingSphere_SetFromRationalPoints4D (
                              TQ3BoundingSphere *bSphere,
                              const TQ3RationalPoint4D *pts,
                              unsigned long nPts,
                              unsigned long structSize);
```

QuickDraw 3D Color Utilities

Contents

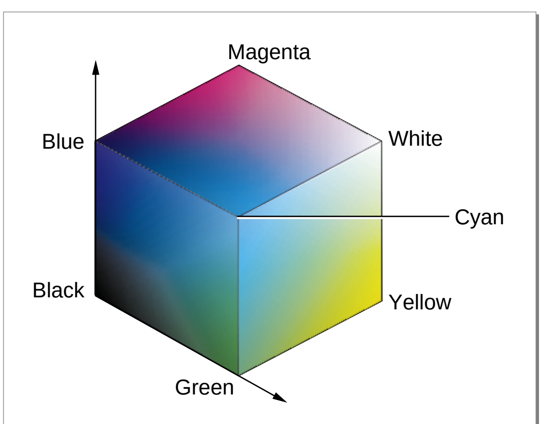
This chapter describes the QuickDraw 3D Color Utilities, a set of functions that you can use to manage colors. You can use these functions to develop distinctive color schemes for the user interface elements of your application.

About the QuickDraw 3D 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.

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. See also Color Plate 2 at the front of this book.

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:

$$\textit{luminance} = (0.30078125 \times \texttt{color.r}) + (0.58984375 \times \texttt{color.g}) + (0.109375 \times \texttt{color.b})$$

Summary of the QuickDraw 3D Color Utilities

C Summary

Data Types

Color Structures

```
typedef struct TQ3ColorRGB {
    float         r;                /*red component*/
    float         g;                /*green component*/
    float         b;                /*blue component*/
} TQ3ColorRGB;

typedef struct TQ3ColorARGB {
    float         a;                /*alpha channel*/
    float         r;                /*red component*/
    float         g;                /*green component*/
    float         b;                /*blue component*/
} TQ3ColorARGB;
```

QuickDraw 3D Color Utilities

```
TQ3ColorRGB *Q3ColorRGB_Set   (TQ3ColorRGB *color,
                               float r, float g, float b);

TQ3ColorARGB *Q3ColorARGB_Set (TQ3ColorARGB *color,
                               float a, float r, float g, float b);
```

```
TQ3ColorRGB *Q3ColorRGB_Add   (const TQ3ColorRGB *c1,
                               const TQ3ColorRGB *c2,
                               TQ3ColorRGB *result);

TQ3ColorRGB *Q3ColorRGB_Subtract (
                               const TQ3ColorRGB *c1,
                               const TQ3ColorRGB *c2,
                               TQ3ColorRGB *result);

TQ3ColorRGB *Q3ColorRGB_Scale (const TQ3ColorRGB *color,
                               float scale,
                               TQ3ColorRGB *result);

TQ3ColorRGB *Q3ColorRGB_Clamp (const TQ3ColorRGB *color,
                               TQ3ColorRGB *result);

TQ3ColorRGB *Q3ColorRGB_Lerp  (const TQ3ColorRGB *first,
                               const TQ3ColorRGB *last,
                               float alpha,
                               TQ3ColorRGB *result);

TQ3ColorRGB *Q3ColorRGB_Accumulate (
                               const TQ3ColorRGB *src, TQ3ColorRGB *result);

float *Q3ColorRGB_Luminance   (const TQ3ColorRGB *color, float *luminance);
```

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.**

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.

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 **pixmap, storage pixmap.**

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 angle button A button 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 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 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 A plane figure having the shape of an oval that closes the base of a cone or one end of a cylinder.

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.**

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 angle button, 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.**

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.**

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`).

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 angle button, 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.**

draw context space See **window coordinate system.**

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.**

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.**

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**.

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.**

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, general polygon, line, marker, mesh, NURB curve, NURB patch, point, polygon, triangle, trigrid.**

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.

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.**

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**.

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.**

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. Defined by the `TQ3MarkerData` data type.

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.

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.

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 angle button, 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.*

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, 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 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 $(0, \theta)$, 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.

polyhedron A solid figure composed of faces.

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 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 angle button, 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 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, 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.**

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.**

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 **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.*

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.

trigrid A grid composed of triangular facets. Defined by the `TQ3TriGridData` 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 **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.

viewing frustum The rectangular frustum defined by a perspective camera and the hither and yon clipping planes. Compare **viewing box.**

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 direction The direction of a view's camera. Also called the *camera vector* or the *viewing vector.*

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.

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.

***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 angle button, distance button, move button, rotate button.**

Index

Symbols

Numerals

A

B

C

D

E

F

G

H

I

J

K

L

N

O

P

Q3A–Q3C

Q3D–Q3G

Q3H–Q3L

Q3M

Q3N–Q3O

Q3P–Q3Q

Q3R–Q3S

Q3T–Q3U

Q3V–Q3W

Q

R

S

T

U

V

W, X

Y

Z

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Proof pages were created on an Apple LaserWriter NTX printer. Final page negatives were output directly from text files on an Agfa Large-Format Imagesetter. 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 Courier.

WRITER
Tim Monroe

DEVELOPMENTAL EDITORS
Antonio Padial, Jeanne Woodward

ILLUSTRATOR
Sandee Karr

PRODUCTION EDITOR
Gerri Gray

PROJECT MANAGER
Patricia Eastman

COVER ILLUSTRATOR
Graham Metcalfe

Special thanks to Kent Davidson, Tracey Davis, Robert Dierkes, Pablo Fernicola, Julian Gómez, Mark Halstead, Mike Kelley, Eiichiro Mikami, Brent Pease, Philip Schneider, Klaus Strelau, Nick Thompson, David Vasquez, Dan Venolia, Ingrid Voss, Kevin Wu.

Acknowledgments to George Corrick, Joe Flesch, Vicky Kaiser, Pete Litwinowicz, Charles Loop, Malcolm MacFail, Fábio Pettinati, Brian Rowe, Steve Rubin, Melissa Sleeter, Ken Turkowski, John Wang.