QuickDraw 3D1 .6

New API Features and Improvements

V5.3.99

QuickDraw 3D

Apple Computer, Inc.

Robert Dierkes (dierkes.r@apple.com)

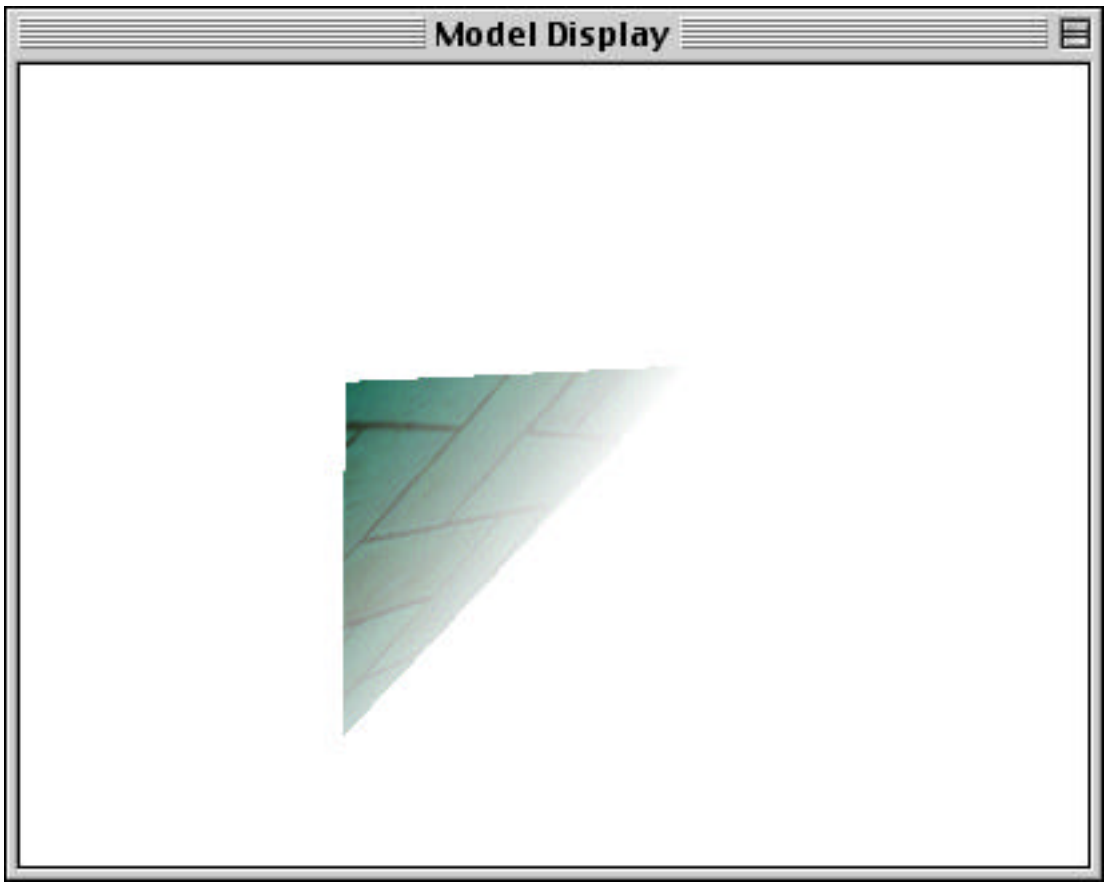
Stephen Luce (sluce@apple.com)

Brian Greenstone

TABLE OF CONTENTS

FOG STYLES

QuickDraw 3D 1.6 now has a Fog Style Objects which will allow you to do atmospheric effects in your 3D applications. Most 3D accelerator cards will work with the new Fog Style once they update their drivers for RAVE 1.6, and even the RAVE 1.6 software rasterizer can now render fogged scenes.



A single textured triangle rendered with white fog

WORKING WITH FOG STYLES

Fog Style Objects are created like any other Style Object, but they take the `TQ3FogStyleData` as input. The new fog types and structures can be found in `QD3DStyle.h`.

```
struct TQ3FogStyleData
{
    TQ3Switch       state;
    TQ3FogMode      mode;
    float           fogStart;
    float           fogEnd;
    float           density;
    TQ3ColorARGB    color;
}

enum TQ3FogMode
{
    kQ3FogModeLinear                = 0,
    kQ3FogModeExponential           = 1,
    kQ3FogModeExponentialSquared    = 2,
    kQ3FogModeAlpha                 = 3
};
```

The `state` field indicates whether the fog is on or off. The `mode` field indicates how the fog increases in density as you look into the distance.

`kQ3FogModeLinear`	fog = (end – z) / (end – start)
`kQ3FogModeExponential`	fog = exp(- density * z)
`kQ3FogModeExponentialSquared`	fog = exp(- density * z * density * z)
`kQ3FogModeAlpha`	fog = vertex alpha

`kQ3FogModeLinear`

The `fogStart` and `fogEnd` values determine the starting and ending points for the fog. Normally, you will want to set `fogEnd` to your camera's yon value, but the `fogStart` value can be any distance depending on the effect you want to achieve. Most applications will start the fog from 1/3 to 1/2 the distance between the camera and the yon plane. Most applications do not look

too good if you set the `fogStart` to your hither value. This causes too much fog and gives a "smoky room" appearance.

`kQ3FogModeExponential` *and* `kQ3FogModeExponentialSquared`

The `density` field determines how dense the fog will get at the `fogEnd` distance for these fog modes. To make a scene go completely to fog, you should set this to 1.0. But if you only want a partial fog for a misty effect, then set it to something lower like 0.5 or 0.8.

`kQ3FogModeAlpha`

This fog mode is a little different. Normally, the "fog" to apply to a particular vertex of a triangle is determined by how far away that vertex is from the camera. With `kQ3FogModeAlpha` the amount of fog to apply to the vertex is determined by the vertex's alpha value. So fog and alpha blending cannot be used with the alpha fog mode. This is mainly used to achieve special effects and not necessarily "fog".

In general, you will want to set your fog color to your clear color. Remember that the fog only affects drawn triangles, therefore, any pixels on the screen that are not occupied by triangles will still show the background color. If, for example, your clear color is black yet your fog is white, then your triangles will fog to white in the distance, but all the "blank" space around the triangles will still be black – not a desirable effect.

NEW API CALLS FOR FOG

Q3FogStyle_New

```
TQ3StyleObject  Q3FogStyle_New (const TQ3FogStyleData*data);
```


Input:	`data`	A pointer to a structure containing the fog parameters.
output:	`TQ3StyleObject`	A fog style object.
Info:	This function returns a reference to a new fog style object based on the input parameters.	

Q3FogStyle_Submit

```
TQ3Status  Q3FogStyle_Submit (
                              const TQ3FogStyleData  *data
                              TQ3ViewObject    view);
```

Input:	`data`	A pointer to a structure containing the fog parameters.
	`view`	A view Object to submit to.
output:	`TQ3Status`	`kQ3Success` if submit was successful.
Info:	This function submits Fog Style data to the input view.	

Q3FogStyle_GetData

```
TQ3Status  Q3FogStyle_GetData (
                              TQ3StyleObject styleObject
                              TQ3FogStyleData  *data);
```

Input: `styleObject` The Fog Style Object whose data you wish to get.

`data` A pointer to a structure to get the fog parameters.

output: `TQ3Status` `kQ3Success` if data was retrieved.

Info: This function gets the data from a Fog Style Object.

Q3FogStyle_SetData

```
TQ3Status  Q3FogStyle_SetData (
                              TQ3StyleObject   styleObject
                              const TQ3FogStyleData  *data);
```

Input: `styleObject` The Fog Style Object whose data you wish to set.

`data` A pointer to a structure containing the fog data.

output: `TQ3Status` `kQ3Success` if data was set successfully.

Info: This function sets the data in a Fog Style Object.

FOG SAMPLE CODE

```
TQ3ColorARGB    gClearColor = {1,1,1,1};   // clear color is white


TQ3StyleObject CreateMyFogStyleObject(void)
{
   TQ3FogStyleData    fogData;
   TQ3StyleObject     fogObj;

        /* SET MY PARAMETERS */

   fogData.state      = kQ3On;                // fog is on
   fogData.mode       = kQ3FogModeLinear;     // fog is linear
   fogData.fogStart   = MY_HITHER;            // start in front
   fogData.fogEnd     = MY_YON;               // end in back
   fogData.density    = 1.0;                  // goes totally foggy at yon
   fogData.color      = gClearColor;          // fog color = clear color


        /* MAKE NEW OBJECT */

   fogObj = Q3FogStyle_New(&fogData);

   return(fogObj);
}
```


WORLD RAY PICKING

World ray picking is a new pick object type introduced with QuickDraw 3D 1.6 which helps applications detect which geometric objects are close to each other in world-space with respect to a ray vector.

To demonstrate a common use of world ray picking, imagine an interactive 3D application where an object, perhaps a character, moves through a scene amongst a number of other objects. Here we need to know what objects are in the character's path so they can be avoided or just determine if something is close enough to be selected. Not all of these other neighboring objects may be visible from the camera's point of view, which precludes us from using the familiar window point pick type in this scenario. To address this a new kind of geometric entity is needed that can be placed anywhere in the scene, not just positioned at the camera's location. World ray picking provides this and allows us to locate a ray anywhere in world-space, like at key points on the character's extremities, pointing in any direction. When the ray pick object is submitted with the scene QuickDraw 3D computes which geometries intersect with the ray. With this information the application can then decide how to reposition the character or have it interact with other surrounding objects.

As you make notice world ray picking differs from window point and window rectangle picking in that it does not perform hit testing from the camera's point of view. Rather it uses an application controlled arbitrary ray that makes it a more generalized means of interacting with objects in a scene.

World ray picking supports the following geometry types.

Linear: Point, Line, PolyLine, and Ellipse

Polygonal: Triangle, Box, Polygon, General Polygon, Polyhedron, Mesh, trimmest, and TriGrid

Quadrics/Conics: Cone, Cylinder, Disk, Ellipsoid, and Torus

Parametric: NURB Curve and NURB Patch

Unsupported: Marker, Pixmap Marker *(ray picking doesn't make sense for these geometries)*

CREATING A WORLD RAY PICK OBJECT

The `TQ3WorldRayPickData` data structure is defined as:

```
struct TQ3WorldRayPickData
{
    TQ3PickData     data;
    TQ3Ray3D        ray;
    float           vertexTolerance;
    float           edgeTolerance;
};
typedef struct TQ3WorldRayPickData     TQ3WorldRayPickData;
```

The `data` field contains general picking information common to all pick types. Essentially it describes in what order hits should be sorted, the kinds of information to be calculated when an intersection is found (known as the "pick detail"), and the maximum number of hits to be accumulated and returned in a pick object's hit list.

The `ray` field holds the pick's ray origin point in world-space coordinates and a direction vector. An important thing to note about this ray direction vector is that it must be normalized before it's passed into the `Q3WorldRayPick_New` and `Q3WorldRayPick_SetRay` functions. The debug library will post a warning if the ray isn't normalized and return a failure status. Note however the optimized library will accept the ray but possible undesirable hits or erroneous hits may result. It is the responsibility of the application to ensure the ray is normalized before passing it to QuickDraw 3D.

The last fields, `vertexTolerance` and `edgeTolerance`, specify the maximum distance allowed between the projected pick ray and where it intersects the geometry. These tolerance values are specified in world-space coordinates.

THE PROCESS OF RAY PICKING

The process of ray picking is very similar to that of the other two pick types. During this process scene objects are submitted in a picking loop to find those geometries which intersect the world ray. The world ray is extended from its origin in the direction of the unit vector and geometries are tested to see if they fall with the given tolerance values or directly intersect the ray.

After submitting the scene an application gets the number of hits with `Q3Pick_GetNumHits`, uses `Q3Pick_GetPickDetailValidMask` to find which kinds of pick detail information are available for an individual hit, and then calls `Q3Pick_GetPickDetailData` to retrieve the pick detail data that was requested and is relevant for a given hit.

A special note: as is the case with all pick types requesting some kinds of pick detail hit information from `Q3Pick_GetPickDetailData` are objects that should be disposed by the

application to avoid leaked objects. Specifically, this applies to the hit path, the geometry object, and the shape part object.

TQ3PickDetailMasks	Means of disposing object
`kQ3PickDetailMaskPath`	`Q3HitPath_EmptyData`
`kQ3PickDetailMask Object`	`Q3Object_Dispose`
`kQ3PickDetailMask ShapePart`	`Q3Object_Dispose`

MORE ABOUT TOLERANCES

As mentioned earlier, edge and vertex tolerance values for world ray picking are specified in positive world-space. This coordinate space differs from window point picking and window rectangle picking, where vertex and edge tolerance values are measured in window space and correspond to screen pixels instead.

Vertex and edge tolerances apply to the geometry types shown below. With mesh geometries these tolerances are only enabled when the mesh is submitted together with a pick part style, i.e. when the `kQ3PickPartsMaskVertex` and/or `kQ3PickPartsMaskEdge` parts settings are used. Ray picking for all other geometries not shown in this table is done on a polygonal face basis.

	Tolerance Used		
Geometry	Vertex	Edge	Face
Point	•		
Line	•	•	
Ellipse	•	•	
NURB Curve	•	•	
Poly Line	•	•	
Mesh	•	•	
All other types			•

OTHER RAY PICKING GUIDELINES

When the `TQ3PickDetail` mask `kQ3PickDetailMaskDistance` is specified the distance returned is measured from the ray origin to the actual intersection on the geometry.

Generally the vertex tolerance should be larger than the edge tolerance if you wish hits on vertices to have precedence over edge hits. Too many multiple hits may be returned if the pick tolerances are too large and the `numHitsToReturn` setting is equal to 0 (return all hits) or greater than 1. This typically happens when:

- 2 or more geometries are close together and the tolerance(s) values are large enough to include several of these geometries.

- 2 or more separate line segments of the same geometry are close together or

adjacent segments and the tolerance value(s) are large. This is possible with polylines, ellipses, or NURB curves.

Therefore, it may be necessary for an application to use tolerance values appropriate for the size of a model's scale.

DATA STRUCTURES & API FUNCTIONS

TQ3WorldRayPickData

```
struct TQ3WorldRayPickData
{
     TQ3PickData     data;
     TQ3Ray3D        ray;
     float           vertexTolerance;
     float           edgeTolerance;
};
typedef struct TQ3WorldRayPickData TQ3WorldRayPickData;
```

Q3WorldRayPick_New

```
TQ3PickObject   Q3WorldRayPick_New (
                                    const  TQ3WorldRayPickData   *data);
```

Input:	`data`	A pointer to a structure containing general pick data to be calculated, the ray's origin and direction vector, and world vertex and edge tolerances.
Output:	`TQ3PickObject`	A world ray pick object is created and returned if the input data was valid.

Info: This function returns a reference to a new world pick ray object created from the given pick data, ray vector, and vertex and edge tolerances. The passed ray origin is in world-space coordinates and the direction vector must be normalized. The vertex and edge tolerance values are measured in world-space coordinates and should be positive values. See the earlier section on tolerance values for more details.

Q3WorldRayPick_GetRay

```
TQ3Status  Q3WorldRayPick_GetRay (
                                TQ3PickObject      pick,
                                TQ3Ray3D           *ray);
```

Input: `pick` A reference to a world ray pick object.

Output: `TQ3Status` Returns `kQ3Succcess` if the ray was copied from the pick object.

`ray` A pointer for the returned ray data.

Info: This function copies and returns the ray's origin point and direction vector into the passed `TQ3Ray3D` structure from the pick object.

Q3WorldRayPick_SetRay

```
TQ3Status  Q3WorldRayPick_SetRay (
                                TQ3PickObject        pick,
                                const TQ3Ray3D       *ray);
```


Input:	`pick`	A reference to a world ray pick object.
	`ray`	A pointer for the new ray data to be set on the pick object.
Output:	`TQ3Status`	Returns `kQ3Succcess` if the passed ray was copied from the pick object. With the debug library installed, `kQ3Failure` is returned if the ray passed in isn't normalized.

Info: This function replaces the ray's origin point and direction vector of the pick object with the given ray. The ray origin is assumed to be in world-space coordinates and the direction must be normalized.

Q3WorldRayPick_GetData

```
TQ3Status  Q3WorldRayPick_GetData (
                                   TQ3PickObject        pick,
                                   TQ3WorldRayPickData  *rayPickData);
```

Input:	`pick`	A reference to a world ray pick object.
Output:	`TQ3Status`	Returns `kQ3Succcess` if the ray pick data was retrieved the pick object.
	`rayPickData`	A pointer for the returned world ray pick data.

Info: This function copies all world ray pick data into the passed `TQ3WorldRayPickData` structure from the pick object.

Q3WorldRayPick_SetData

```
TQ3Status  Q3WorldRayPick_SetData (
                                TQ3PickObject               pick,
                                const TQ3WorldRayPickData   *rayPickData);
```

Input:

`pick`	A reference to a world ray pick object.
`rayPickData`	A pointer to world ray pick data.

Output:

`TQ3Status`	Returns `kQ3Succcess` if the ray pick data was valid and set on the pick object.

Info: This function replaces the data in the world ray pick object with the data from passed `TQ3WorldRayPickData` structure. The passed ray origin is specified in world-space coordinates. The data contained in the structure must be valid and the ray's direction must be normalized.

WORLD RAY PICKING SAMPLE CODE

The following example shows how to create a ray pick object and change the ray to move it through a scene. Using the view and model group passed to the main function it repositions the ray pick at various locations along a circle to find objects that fall in this circular path. In the HandleHit call the application could check if the object hit is at an acceptable distance by using the pick detail mask `kQ3PickDetailMask Distance`.

```
TQ3Status RayPickingSample(
   TQ3ViewObject      view,
   TQ3GroupObject     model)
{
   TQ3Status       status = kQ3Failure;
   TQ3Ray3D        worldRay;
   TQ3PickObject   pick = NULL;
   unsigned long   numHits = 0;
   float           angle = 0.0;

   #define    kRadius    10.0
   #define    kStep      15.0

   /* Set initial ray origin and direction */
   Q3Point3D_Set(&worldRay.origin, 0.0, 0.0, 0.0);
   Q3Vector3D_Set(&worldRay.direction, 1.0, 0.0, 0.0);

   pick = CreateWorldRayPick(&worldRay);
   if (pick == NULL) {
      return kQ3Failure;
   }

   /*
    * Move the ray's origin in a circular path in the XZ-plane
    * keeping the direction vector tangent to this circle.
    */
   for (angle = 0.0;
       angle <= Q3Math_DegreesToRadians(360.0);
       angle += Q3Math_DegreesToRadians(kStep)) {

      /* Position ray's origin */
      Q3Point3D_Set(&worldRay.origin,
                    cos(angle) * kRadius,
                    0.0,
                    sin(angle) * kRadius);

      /* Set a new ray direction as a tanget to the circle */
      worldRay.direction.x =  -sin(angle);
      worldRay.direction.z =   cos(angle);
```


```
      /* Change ray in pick */
      Q3Vector3D_Normalize(&worldRay.direction, &worldRay.direction);
      status = Q3WorldRayPick_SetRay(pick, &worldRay);

      /* Test if any scene objects intersect the ray */
      status = Pick_Model(view, model, pick, &numHits);
      if ((status == kQ3Success)  &&  (numHits > 1)) {
         HandleHit(pick, model);
      }
   }

   if (pick != NULL) {
      Q3Object_Dispose(pick);
      pick = NULL;
   }

   return kQ3Success;
}


/*
 * Sets up and create a world ray pick object
 */
TQ3PickObject CreateWorldRayPick(
   TQ3Ray3D        *pWorldRay)
{
   TQ3WorldRayPickData     rayPickData;
   TQ3PickObject           pickObject = NULL;

   rayPickData.data.sort =  kQ3PickSortNearToFar;

   rayPickData.data.mask =    kQ3PickDetailMaskObject      |
                              kQ3PickDetailMaskXYZ         |
                              kQ3PickDetailMaskDistance    |
                              kQ3PickDetailMaskNormal;
   rayPickData.data.numHitsToReturn = 1;

   /* Make sure ray is normalized  */
   Q3Vector3D_Normalize(&pWorldRay->direction, &pWorldRay->direction);
   rayPickData.ray = *pWorldRay;

   /* Tolerance values are measured in world space. */
   rayPickData.vertexTolerance     = 0.01;
   rayPickData.edgeTolerance       = 0.005;

   /* Create the new world ray pick object */
   pickObject = Q3WorldRayPick_New(&rayPickData);

   return pickObject;
}
```


```
/*
 * A basic picking submit loop
 */
TQ3Status Pick_Model(
   TQ3ViewObject      viewObj,
   TQ3GroupObject     modelObj,
   TQ3PickObject      pickObj,
   unsigned long      *pNumHits)
{
   TQ3Status       status     = kQ3Failure;
   TQ3ViewStatus   viewStatus = kQ3ViewStatusError;

   if (Q3View_StartPicking(viewObj, pickObj) == kQ3Success) {

      do {
         Q3DisplayGroup_Submit(modelObj, viewObj);
         viewStatus = Q3View_EndPicking(viewObj);
      } while (viewStatus == kQ3ViewStatusRetraverse);

      if (viewStatus == kQ3ViewStatusDone) {
         status = Q3Pick_GetNumHits(pickObj, pNumHits);
      }
   }

   return status;
}
```

RELATED API'S

```
Q3Pick_GetType
Q3Pick_GetData
Q3Pick_SetData
Q3Pick_GetNumHits
Q3Pick_GetPickDetailValidMask
Q3Pick_GetPickDetailData
Q3HitPath_EmptyData
Q3Pick_GetVertexTolerance
Q3Pick_SetVertexTolerance
Q3Pick_GetEdgeTolerance
Q3Pick_SetVertexTolerance
```


DISPLAY GROUP CULLING

One performance bottleneck with QuickDraw 3D has always been that it culls geometry on an object by object basis, and complex geometries such as Conics get culled very late in the pipeline which results in a tremendous amount of computation going into building a conic which might not even be visible. Even simple geometries such as TriMeshes take a speed hit. In the case of a model of a Jumbo Jet, which is made up of 20 different TriMeshes, 20 different culling tests are performed on this single model.

So, the solution has always been for coders to write their own model culling function. This is not a very friendly thing to ask a programmer to do when they're using a high level API like QD3D. Therefore, new in QD3D 1.6 is the ability to assign a Bounding Box to a Display Group. When the group is submitted for rendering, it will be cull tested and if it fails, then none of the geometry or other objects inside the group will be submitted for rendering. The speed boost from this ranges from 30% to 3500% or even higher for some conditions.

Using Group culling is a very powerful feature, but it must be used carefully. If you correctly assign a Bounding Box to a Group but then the geometry inside of the Group changes, the Group has no way to know that the Bounding Box may not be the right size anymore. The worst thing that can happen is that Groups will be culled prematurely. This happens when a Group's Bounding Box is smaller than it should be to encompass all of the geometry contained within the Group. If the Bounding Box is too large, then the Group may not get culled when it should and the contained geometries will get processed. Don't worry, however, because these geometries will then be cull-tested later down the pipeline when they would have been cull-tested anyway. No crashes or other serious problems will result from an incorrect Bounding Box, but you need

to be careful to avoid visual glitches.

API CALLS FOR DISPLAY GROUP CULLING

Q3DisplayGroup_SetAndUseBoundingBox

```
TQ3Status  Q3DisplayGroup_SetAndUseBoundingBox (
                                TQ3GroupObject    group,
                                TQ3BoundingBox    *bBox)
```

Input: `group` The Display Group to which you wish to assign a bounding box.

`bBox` A pointer to the Bounding Box you want to assign to the group.

Output: `TQ3Status` `kQ3Success` if the input bounding box was assigned to the group.

Info: This function will assign the input Bounding Box to the input Display Group. If the `isEmpty` field of the bounding box is `kQ3True`, then this function will return `kQ3Failure`. Only non-empty bounding boxes may be assigned to the Display Group.

In addition to assigning the bounding box to the Display Group, this function also sets the Group's `kQ3DisplayGroupStateMaskUseBoundingBox` state flag which indicates that QuickDraw 3D should use the assigned bounding box to perform culling. You can use `Q3DisplayGroup_SetState` to clear the

kQ3DisplayGroupStateMaskUseBoundingBox state flag which will cause QuickDraw 3D to ignore the assigned bounding box.

Note: in previous versions of QuickDraw 3D, the kQ3DisplayGroupStateMaskUseBoundingBox flag was always set by default on every new Group even though the flag did nothing. This is no longer the case in 1.6. Now, the flag is clear by default.

Q3DisplayGroup_CalcAndUseBoundingBox

```
TQ3Status  Q3DisplayGroup_CalcAndUse (
                                    TQ3GroupObject  group,
                                    TQ3ComputeBounds   computeBounds,
                                    TQ3ViewObject   view)
```

Input:	group	The Display Group who's Bounding Box you wish to calculate and use.
	computeBounds	Determines how accurate you would like your bounding box to be. Use kQ3ComputeBoundsExact or kQ3ComputeBoundsApproximate.
	view	The view to associate with calculating the bounding box.
Output:	TQ3Status	kQ3Success if the bounding box was successfully calculated and assigned to the group.

Info: This function takes the input Display Group and calculates the Bounding Box, which encloses all of the Group's geometries. The bounding box is then assigned to the group and the group's `kQ3DisplayGroupStateMaskUseBoundingBox` state flag is set.

This function may not be called inside of a Submit loop. It will correctly calculate the Bounding Box for the group in its current state, however, the programmer will still need to be aware that changes to the Group's contained objects may cause the bounding box to no longer be the correct size and thus Group culling may not function properly.

Q3DisplayGroup_GetBoundingBox

```
TQ3Status  Q3DisplayGroup_GetBoundingBox (
                                TQ3GroupObject    group,
                                TQ3BoundingBox    *bBox)
```

Input: `group` The Display Group who's Bounding Box you wish to get.

`bBox` A pointer to a Bounding Box structure to receive the groups bounding box data.

Output: `TQ3Status` kQ3Success if the bounding box was successfully retrieved from the group.

Info: This function will return the Group's currently assigned Bounding Box. If no Bounding Box is assigned to the Group, then the function returns `kQ3Failure`.

Q3DisplayGroup_RemoveBoundingBox

```
TQ3Status  Q3DisplayGroup_RemoveBoundingBox (TQ3GroupObject   group)
```

Input:	`group`	The Display Group who's Bounding Box you wish to remove.
Output:	`TQ3Status`	kQ3Success if the bounding box was successfully removed from the group.

Info: This function removed any assigned Bounding Box from the input Group. If there was no assigned Bounding Box, then the function does nothing and returns `kQ3Success`. However, if there was a Bounding Box, then it is removed from the Group and the Group's `kQ3DisplayGroupStateMaskUseBoundingBox` state flag is cleared.

Q3View_AllowAllGroupCulling

```
TQ3Status  Q3View_AllowAllGroupCulling (
                                TQ3ViewObject  view,
                                TQ3Boolean     allowCulling)
```


Input:	`view`	A View Object.
	`allowCulling`	A flag to turn group culling on or off. kQ3True will allow Group culling, kQ3False will turn it off for all Groups rendered by the view.
Output:	`TQ3Status`	kQ3Success if the flag was set.

Info: This function allows the application to deactivate Group Culling in the input View. By default, Group Culling is active in a View, but passing kQ3False to this function will disable it for all Groups being submitted. Passing `kQ3True` will re-enable Group culling.

In addition to adding the above function calls, there are also changes to the 3DMF output files. When a Display Group that has an assigned Bounding Box is written to a 3DMF file, additional data is written to the BeginGroup() definition:

```
BeginGroup (
   DisplayGroup ( )
   DisplayGroupState ( Inline
   )
   DisplayGroupBBox (
      -91.52821 -135.8437 -77.85556 91.52821 135.8437 77.85556 )
)
```

Previously, only the DisplayGroupState() data was written if any of the state flags were set.

If you are creating a Culling Display Group and it only contains geometry data (no transforms or attribute objects), then you should set the Group's `kQ3DisplayGroupStateMaskIsInline` state flag so that processing of the Group will be much more efficient. Any Group which is being used solely to contain data and not to define some hierarchical system should always have the inline state flag set.

DISPLAY GROUP CULLING SAMPLE CODE

The following code takes a list of Geometry Objects, puts them into a new Display Group, and then calculates and assigns a bounding box to the Display Group. When this Group is submitted, all of the enclosed geometries will be cull-tested up front which results in a huge performance boost.

```
/*****************  OPTIMIZE SOME MODELS **********************/
//
// INPUT:  numModels  = # models to put into culling group
//    models     = array of models
//

TQ3DisplayGroupObject OptimizeSomeModels(long numModels, TQ3GeometryObject *models)
{
    long                        i;
    TQ3DisplayGroupObject     theGroup;
    TQ3DisplayGroupState      state;

      /* MAKE A NEW DISPLAY GROUP TO PUT MODELS INTO */

    theGroup = Q3DisplayGroup_New();
    if (theGroup == nil)
      return(nil);


      /* PUT EACH MODEL INTO THE GROUP */

   for (i = 0; i < numModels; i++)
      Q3Group_AddObject(theGroup , models[i]);


      /* CALCULATE AND ACTIVATE THE BBOX */

   Q3DisplayGroup_CalcAndUseBoundingBox(theGroup,
```

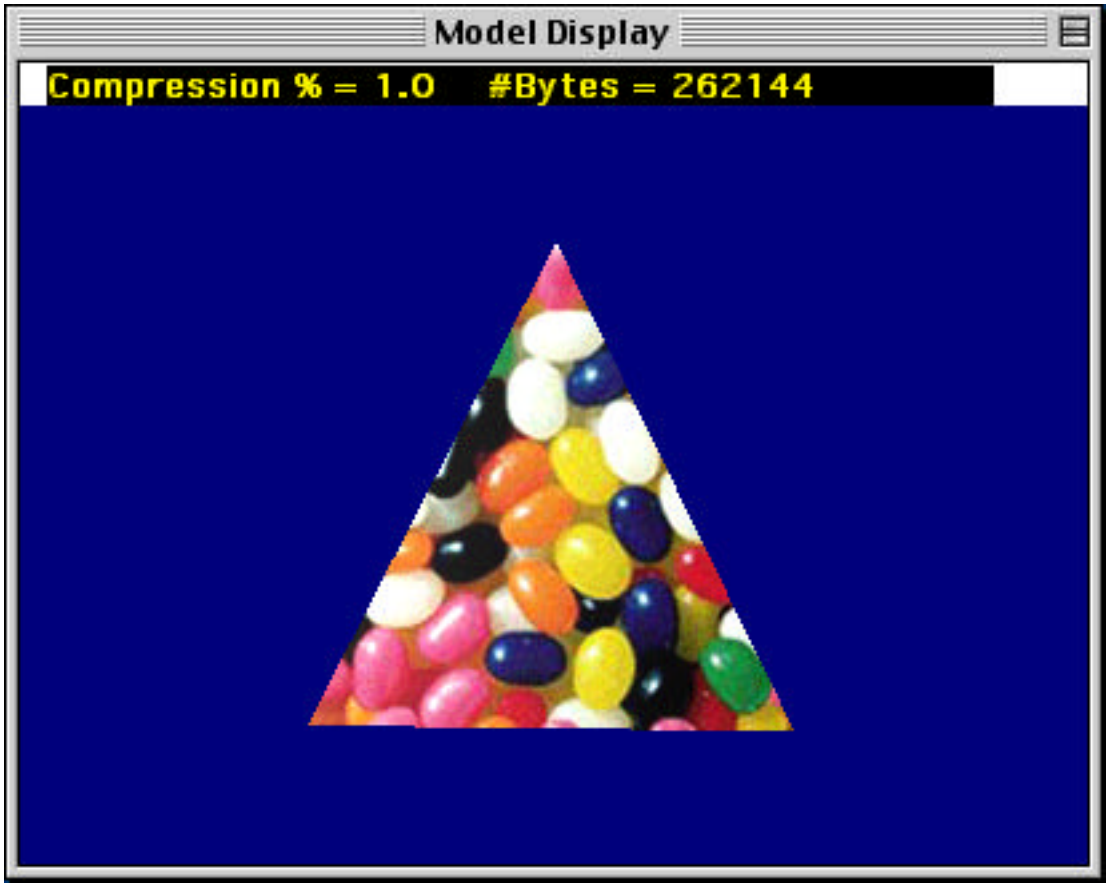

```
                                kQ3ComputeBoundsExact,
                                gMyViewObject);

    /* MAKE GROUP INLINE SINCE IT DOENST CONTAIN ATTRIBUTES OR TRANSFORMS */

    Q3DisplayGroup_GetState(theGroup ,&state);
    state |= kQ3DisplayGroupStateMaskIsInline;
    Q3DisplayGroup_SetState(theGroup ,state);

    return(theGroup);
}
```


COMPRESSED PIXMAP

New for 1.6 is the `TQ3CompressedPixmap` texture type. This texture type works just like the `Pixmap` and `Mipmap` types except that the texture pixel data is compressed with QuickTime. This new texture was designed to be easy to use, and it gives the user the flexibility of supplying their own QuickTime compressed image data or if they desire, QD3D will do the compression for them.

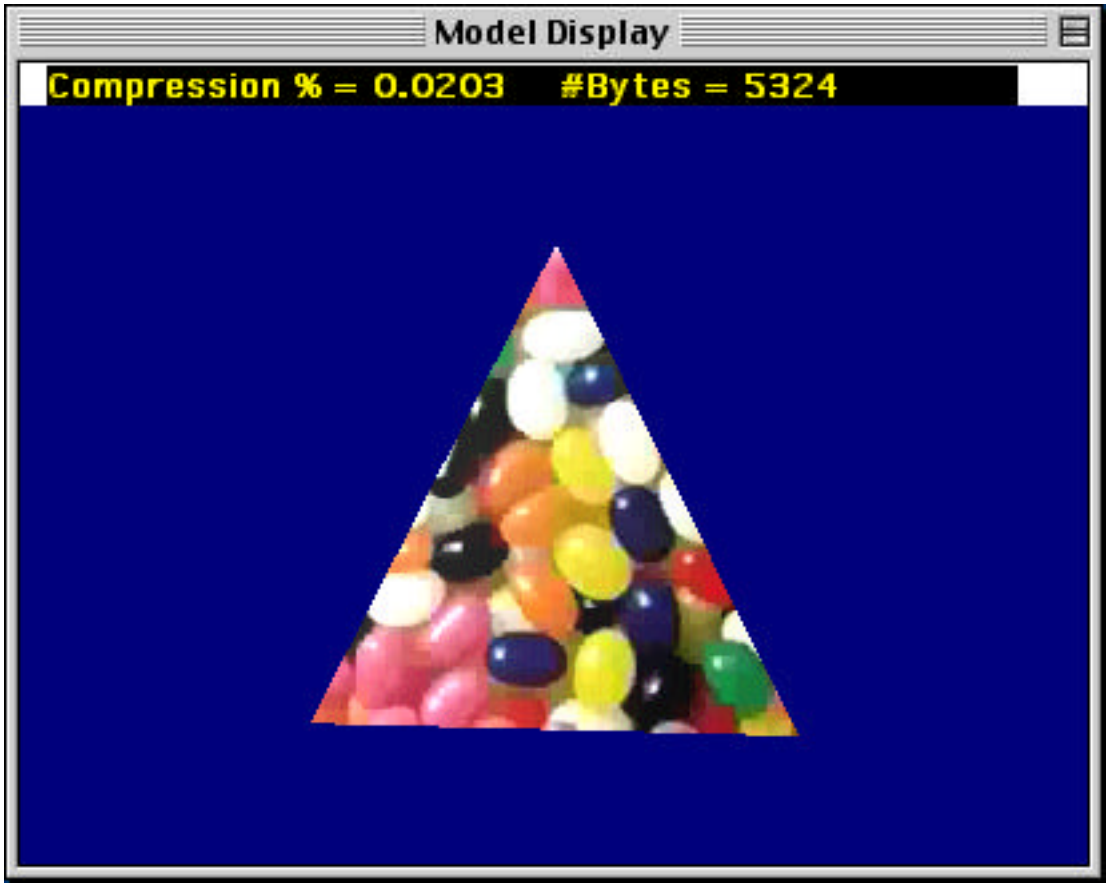
COMPRESSED PIXMAP SAMPLES

The following images show various types of compression applied to a texturemapped triangle. Notice how good the compression ratio is for JPEG and Sorenson, yet the degradation of the quality of the texture is almost unnoticeable. Sorenson compresses the texture to 2% of its original size, but you can barely tell any difference.



No Compression

JPEG Low Quality



Sorenson Least Quality

STRUCTURES AND API FUNCTIONS

```
typedef struct TQ3CompressedPixmap
{
   TQ3StorageObject   compressedImage;     /* contains  compressed  image data */
   TQ3Endian          imageDescByteOrder;  /* endianness of  data in the imageDesc */
   TQ3StorageObject   imageDesc;           /* contains  QT image description */
   TQ3Boolean         makeMipmaps;         /* kQ3True = will render with mipmaps */
   unsigned long      width;               /* width of texture */
   unsigned long      height;              /* height of texture */
   unsigned long      pixelSize;           /* pixel size 16 or 32 bits */
   TQ3PixelType       pixelType;           /* pixel type */
} TQ3CompressedPixmap;
```


Note that this structure does not contain much of the information (such as `rowBytes`) that the normal Pixmap structure has. Since the data is compressed, many parameters are simply not needed. QuickTime stores all of the information it needs to decode the image in the image description handle. The `imageDesc` record is not an Image Description handle, but rather a QuickDraw 3D container object, which contains the QuickTime Image Description Handle's data. In order to use this data in a QuickTime function call, you will need to convert this data into a real Handle to pass the QuickTime functions. Faking it with fake handles (i.e. &buffer) will not work – QuickTime will probably return an error.

The function calls for working with Compressed Pixmaps are identical to those for the other texture types. However, there is one new function designed to assist in creating the compressed data:

Q3CompressedPixmapTexture_CompressImage

```
TQ3Status  Q3CompressedPixmapTexture_CompressImage(
                                TQ3CompressedPixmap *compressedPixmap,
                                PixMapHandle        sourcePixMap,
                                CodecQ              codecQuality,
                                CodecType           codecType,
                                CodecComponent      codecComponent)
```

Input: `compressedPixmap` Pointer to the TQ3CompressedPixmap structure to receive the compressed image.

`SourcePixMap` `PixMapHandle` that contains the uncompressed image that you want to be compressed.

	`CodecQuality`	The QuickTime quality value to use to compress the image.
	`CodecType`	The QuickTime codec type to use to compress the image.
	`CodecComponent`	The QuickTime codec component to use to compress the image.
Output:	`TQ3Status`	`kQ3Success` if compression was successful.
	`CompressedPixmap`	The `compressedImage` and `imageDesc` fields will contain references to storage objects if compression was successful.

Info: This is a utility function for creating a Compressed Pixmap texture. Given various QuickTime parameters and a PixMapHandle containing the uncompressed source image, this function will use QuickTime to compress the image and then create two new storage objects that contain the compressed image data and the QuickTime image description data.

If the function succeeds in compressing the texture image, it saves the reference to the storage objects in the `compressedImage` and `imageDesc` fields of the `TQ3CompressedPixmap` structure. You still need to fill out the other fields of this since this function will not do it for you.

The QuickTime constants for the various compression parameters are found in the header file `ImageCompression.h`. Note that not all compressor CODECs can be used to compress a texture. Some compressors only decompress data. If you attempt to use a CODEC that cannot do compression, `Q3CompressedPixmapTexture_CompressImage` will return `kQ3Failure`. Your application may want to let the user select the compression parameters with the QuickTime function `SCRequestSequenceSettings`.

The other new functions for the CompressedPixmap texture type should look very familiar:

```
TQ3TextureObject Q3CompressedPixmapTexture_New(
                    const TQ3CompressedPixmap *compressedPixmap)

TQ3Status Q3CompressedPixmapTexture_GetCompressedPixmap(
                    TQ3TextureObject texture,
                    TQ3CompressedPixmap *compressedPixmap)

TQ3Status Q3CompressedPixmapTexture_SetCompressedPixmap(
                    TQ3TextureObject texture,
                    const TQ3CompressedPixmap *compressedPixmap)
```

COMPRESSED PIXMAP EXAMPLE

```
/***********  GWORLD TO COMPRESSED PIXMAP ******************/

TQ3TextureObject   GworldToCompressedPixmap(GworldPtr theGWorld)
{
PixMapHandle            hPixMap;
unsigned long           pictMapAddr;
unsigned long           pictRowBytes;
TQ3CompressedPixmap     pixmap;
```


```
TQ3TextureObject        texture;
TQ3Status               status;


          /* GET GWORLD ADDR & ROWBYTES */

   hPixMap = GetGWorldPixMap(theGWorld);
   pictMapAddr = (unsigned long )GetPixBaseAddr(hPixMap);
   pictRowBytes = (unsigned long)(**hPixMap).rowBytes & 0x3fff;


        /* FILL OUT COMPRESSED PIXMAP STRUCTURE */

   pixmap.compressedImage     = nil;
   pixmap.imageDesc           = nil;
   pixmap.makeMipmaps         = kQ3True;
   pixmap.width               = width;
   pixmap.height              = height;

   if (gCodecDepth == 32)
   {
      pixmap.pixelSize        = 32;
      pixmap.pixelType        = kQ3PixelTypeRGB32;
   }
   else
   {
      pixmap.pixelSize        = 16;
      pixmap.pixelType        = kQ3PixelTypeRGB16;
   }

        /* COMPRESS IMAGE AND FILL OUT REMAINING RECORDS IN STRUCTURE */
        //
        // Note:  gCodecType, gCodecComponent, gCodecDepth, and
        //         gCodecQuality are global variables we got from a
        //         previous call to SCRequestImageSettings() and
        //         SCGetInfo().
        //

   status = Q3CompressedPixmapTexture_CompressImage(&pixmap,
                                                    hPixMap,
                                                    gCodecType,
                                                    gCodecComponent,
                                                    gCodecDepth,
                                                    gCodecQuality);
   if (status == kQ3Failure)
      DoError("\pQ3CompressedPixmapTexture_CompressImage Failed!");


      /* MAKE NEW COMPRESSED PIXMAP TEXTUE OBJECT */

   texture = Q3CompressedPixmapTexture_New (&pixmap);
   if (texture == nil)
      DoError ("\pQ3CompressedPixmapTexture_New failed!");

   return(texture);
}
```


COMPRESSED PIXMAPS IN 3DMF FILES

Compressed Pixmaps get written out to 3DMF files just like other texture types. However, unlike other texture types who's texture data can be edited in a text 3DMF file, the Compressed Pixmap writes out compressed data to the 3DMF file and thus may appear as random patterns in the text 3DMF files. Do not attempt to edit the compressed data in a 3DMF file. This will likely lead to decompression corruption and possibly a crash in the QuickDraw 3D application attempting to view the file.

The Texture Shader for the above Jelly Bean texture which was compressed with Sorenson Least Quality looks like this:

```
Container (
      TextureShader ( )
      compressedpixmaptexture5:
      CompressedPixmapTexture (
         86 BigEndian 4996 False 256 256 16 RGB16
         0x00000056535651310000000000000000
         0x00020002535669730000000000000000
         0x01000100004800000048000000001384
         0x00010E536F72656E736F6E2056696465
         0x6F00000000000000000000000000000000
         0x00000018FFFF
         0x0000800017100100 3E0500000C6D528A
         0x0A7BA210DE56AC87FE79D48CBC267317
         0xFD148F92354B02AA37A0A11755909163
         0x9903E8282A7AF37B8E2E31A0C33A681D
         0xCC88A83F5A126CF430CF947E4A7C5532
         0xE50740E468FA17DCE801CE465A5F4CB6
         0xA84972F0D42479AA02D9E0B183F28B99
         ....
         ....
```

Let's see what all of this means:

The “86” is the size of the QuickTime Image Description Data. The next item “BigEndian” indicates the endianness of the Image Description Data. Following that is the number “4996” which indicates the size of the compressed texture data. “False” indicates no mipmapping. If this were set to “True” then that would indicate that this texture should be mipmapped. The next two numbers “256” and “256” are the width and height of the texture, and “16” and “RGB16” indicate the pixel size and type of the texture.

The next block of binary data is the actual QuickTime Image Description. Notice it is 86 bytes long as indicated above. Immediately following this data is the actual compressed texture itself. This data is actually 4996 bytes long, but we’ve truncated it above in an effort to save the rainforests.

GETTING THE RAVE DRAW CONTEXTS

There is now a way to get access to the Interactive Renderer's RAVE Draw Contexts from QuickDraw 3D. This gives you the ability to make RAVE calls directly to the 3D hardware or software rendering engine.

NEW API FUNCTIONS

Q3InteractiveRenderer_CountRAVEDrawContexts

```
TQ3Status  Q3InteractiveRenderer_CountRAVEDrawContexts(
                                    TQ3RendererObject  renderer,
                                    unsigned long      *numRAVEContexts)
```

Input: `renderer` a reference to a QD3D Renderer Object

Output: `numRAVEContext` the number of RAVE Draw Contexts owned by renderer.

Info: This function returns in `numRAVEContexts` the total number of RAVE Draw Contexts that the input renderer object *currently* owns. QuickDraw 3D's Interactive Renderer does not automatically create the RAVE Draw Contexts when the Renderer object is created or assigned to a View Object. The RAVE Draw Contexts are created when `Q3View_StartRendering` is called.the first time. Each time `Q3View_StartRendering` is called, QuickDraw 3D checks to see if the View has changed and if so, it deletes all of the RAVE Draw Contexts and recreates new ones.

To use this function before you have done any rendering, do the following:

```
Q3View_StartRendering(viewObject);
Q3InteractiveRenderer_CountRAVEDrawContexts(rendererObject,&num);
Q3View_Cancel(viewObject);
```

Q3InteractiveRenderer_GetRAVEDrawContexts

```
TQ3Status  Q3InteractiveRenderer_GetRAVEDrawContexts(
                              TQ3RendererObjectrenderer,
                              TQADrawContext    **raveDrawContextList,
                              TQAEngine         **raveDrawEngineList,

                              unsigned long  *numRAVEContexts,
                              void   (*raveDestroyCallback)(TQ3RendererObject))
```

Input:

`renderer`	a reference to a QD3D Renderer Object.
`raveDestroyCallback`	the callback function to call when the RAVE Draw Contexts become invalid.

Output:

`raveDrawContextList`	pointer to array of RAVE Draw Context pointers.
`RaveDrawEngineList`	pointer to array of RAVE Draw Engines.
`numRAVEContexts`	the number of RAVE Draw Contexts owned by renderer.

Info: This function returns a pointer to each RAVE Draw Context and Drawing Engine associated with the input Renderer Object. It also indicates the number of RAVE Draw Contexts returned.

As with `Q3InteractiveRenderer_CountRAVEDrawContexts`, this function will return 0 draw contexts if `Q3View_StartRendering` was not called earlier. QuickDraw 3D only creates the new RAVE Draw Contexts for the Renderer when `Q3View_StartRendering` is called.

Changing any parameters of the View associated with the renderer may result in the RAVE Draw Contexts being destroyed and recreated. Because of this, it is very important to be careful when using the returned `TQADrawContext` pointers. Some actions that will cause the RAVE Draw Contexts to be deleted and recreated are:

- Moving or resizing the View
- Changing the View's Clear Method
- Changing z-buffer depth.
- Changing the screen's color depth or resolution.

If you are simply creating a "static" View and never changing any View parameters, then you can be assured that the RAVE Draw Context pointers would remain valid.

The `raveDestroyCallback` parameter lets you assign a callback function to be called whenever the RAVE Draw Contexts become invalid. This way you do not need to guess when QuickDraw 3D may have disposed of them. The minimal callback function takes the following form:

```
void MyRaveInvalidateCallback(TQ3RendererObject  rendererRef)
{
```


```
}
```

** Note that your callback function must **not** dispose of the reference to the Renderer Object!

Your callback function can immediately call `Q3InteractiveRenderer_GetRAVEDrawContexts` to get the new RAVE Draw Contexts if any have been created. A callback function which does this should look like this:

```
void MyRaveInvalidateCallback(TQ3RendererObject  rendererRef)
{
    TQ3Status    status;

    status  = Q3View_StartRendering(gMyView);
    gNumContexts  = Q3InteractiveRenderer_GetRAVEDrawContexts (
                                        rendererRef,
                                        gContextList,
                                        gEngineList);
    if (status == kQ3Success)
        Q3View_Cancel(gMyView);
}
```

This code does something a little different. Since `Q3InteractiveRenderer_GetRAVEDrawContexts` needs to be called inside a rendering loop to be certain that the Contexts are up to date, we call `Q3View_StartRendering`. But remember that our callback function might have been called from within a render loop in which case `Q3View_StartRendering` would have already been called and active. Therefore, calling `Q3View_StartRendering` in our callback will return `kQ3Failure` because the View

is already in a rendering loop. We do not want to call `Q3View_Cancel` if this is the case, therefore, if `Q3View_StartRendering` fails, we do not call `Q3View_Cancel`.

Be aware that there are cases when your callback will be invoked to notify you that the Draw Contexts have been invalidated, but there might not necessarily be any new RAVE Draw Contexts assigned to the Renderer yet. Therefore, `Q3InteractiveRenderer_GetRAVEDrawContexts` will return a count of 0. This will occur when you dispose of the View or Renderer Objects entirely since the RAVE Draw Contexts will be invalidated and obviously no new ones will be created to replace them since you have nuked the View and/or Renderer.

If you do not wish to use a callback function to notify you when the RAVE Draw Contexts have become invalid, simply pass `NULL` for `raveDestroyCallback`.

As noted above, this function also returns the `TQAEngine` associated with each Draw Context. Note that multiple Draw Contexts may share the same Drawing Engine. Therefore, if you are going to make RAVE calls that take the `TQAEngine` as input such as `QABitmapNew`, be careful not to upload the bitmap to the same Drawing Engine multiple times. You may get two Draw Contexts which use the save Drawing Engine and the bitmap only needs to be uploaded to the Drawing Engine once for both Draw Context to use it.

If you are using the RAVE Draw Context to simply set “global” RAVE state variables like blending modes or whatever, then you should have no problems. However, if you are planning on making RAVE calls which normally require you to be inside a RenderStart / RenderEnd function such as drawing triangles or

textures, then you need to be very careful about something:

If your QD3D View has multiple RAVE Draw Contexts (because it crosses multiple monitors), then you need to know which Draw Context is the currently active context when you are inside the `Q3RenderStart` / `Q3RenderEnd` loop. You will loop through your "Submit" loop once for each monitor that the View touches. When you call Q3InteractiveRenderer_GetRAVEDrawContexts, it returns the RAVE Draw Contexts in the order that QuickDraw 3D processes them during rendering.

Therefore, if your View crosses 2 monitors then you will loop through your rendering loop two times. The first time through the loop, the active RAVE Draw Context will be the first context returned by `Q3InteractiveRenderer_GetRAVEDrawContexts`. The second time through the loop, the active RAVE Draw Context will be the second context returned by `Q3InteractiveRenderer_GetRAVEDrawContexts`.

You need to be careful to manage this correctly in your application because drawing a triangle with a RAVE Draw Context that is not currently active will cause an error to occur.

On a similar note, be careful not to submit textures, bitmaps, or triangles that are not clipped to the Draw Context's bounds. For example, suppose you have a View that is 100 pixels wide but crosses two monitors and only 20 pixels are on the left monitor. The other 80 pixels are on the right monitor. If you then call `QADrawBitmap` using a bitmap which is 30 pixels wide into the left Draw Context,

the RAVE Software Renderer will trash memory as it draws off the right side of that monitor. Unlike QuickDraw 3D which internally handles clipping for you, RAVE does not and it is very easy to trash memory if you are not careful about these kinds of situations.

In general, if you don't really know much about RAVE or you just plain don't know what you are doing, then only use these RAVE Draw Contexts you acquired from QuickDraw 3D to set RAVE State Variables. Do not use it to upload textures or draw triangles. Only seasoned veterans of RAVE should attempt to do this.

NEW VIEWER FUNCTIONS

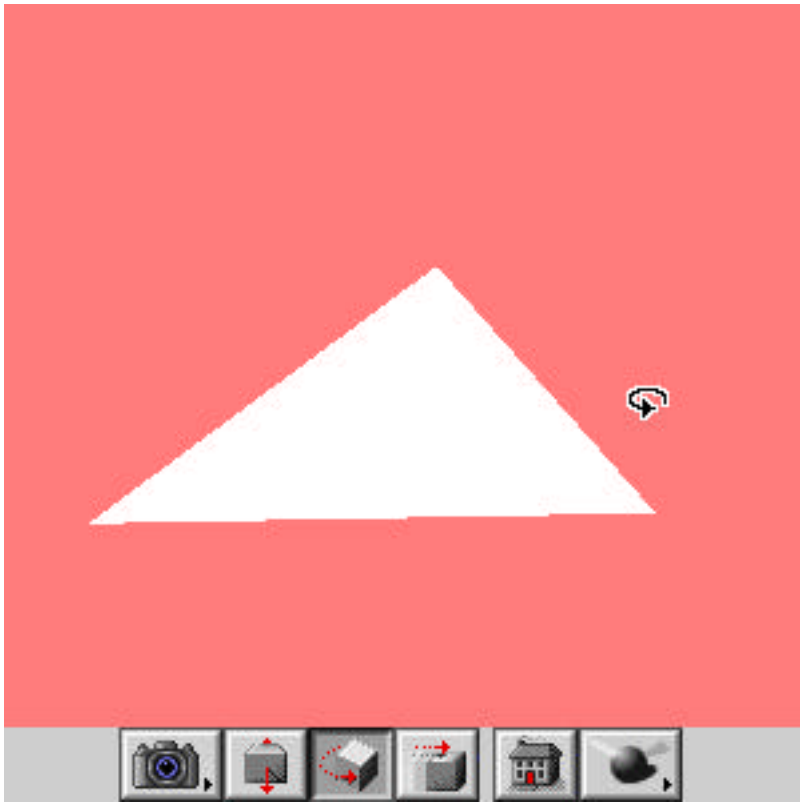
The new Viewer in QuickDraw 3D 1.6 is much improved over the old Viewer. User interaction of the 3D models is much more intuitive and has much better feedback. The visual appearance of the models is also greatly improved. Note that the default lighting in the Viewer is a standard studio lighting setup: 1 key light, 1 fill light, and 1 backlight.

FLY-THRU MODE

In the new Viewer's Camera menu, the first item now says Enter Fly-Thru Mode. When in this mode, the user's control of the camera changes. Moving the mouse up and down will move the camera forward and backward. Moving the mouse horizontally moves the camera sideways.

THE OPTIONS BUTTON

A new Options control strip button has been added to the Viewer which allows the user to modify the appearance of the rendered image in the Viewer.



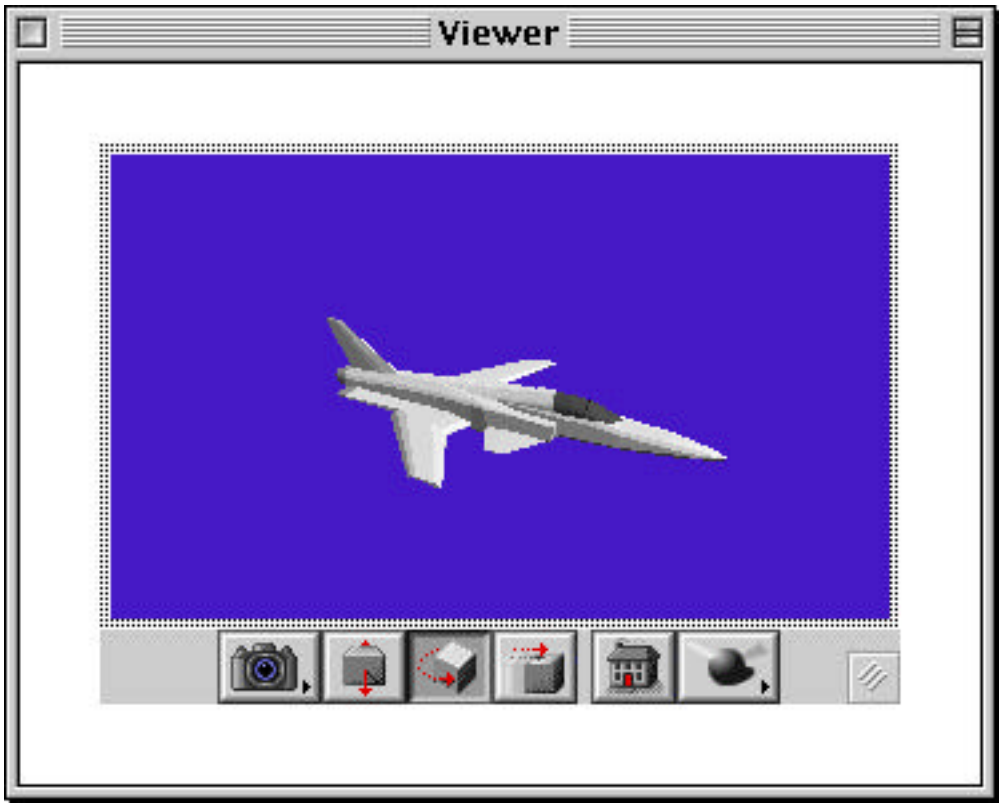
The icon on the far right is the new Options button

The Options button will pop open a sub-menu containing the following selections:

- **Renderer** The user can select from any of the installed Interactive Renderers for displaying the model.
- **Brightness** The user can change the brightness of the scene.
- **Background Color** The user can change the background color of the Viewer.
- **Remove Backfaces** Toggles backface removal on and off for better looking images and faster rendering.
- **Phong Shading** Toggles phong shading on and off to alter the appearance of the model.

RESIZING THE VIEWER PANE INSIDE THE WINDOW

Prior to QuickDraw 3D 1.6 it was only possible to have a window with a grow box that resized the Viewer's dimensions and always fill the entire window. This was done by setting the Viewer's `kQ3ViewerDrawGrowBox` flag. Some applications still need a resizable window and a resizable Viewer but wish to resize them independently. A new flag `kQ3ViewerPaneGrowBox` has been added so the Viewer can be resized in it's own pane. Setting this flag draws a grow box inside the Viewer's control strip which accepts mouse clicks and resized the Viewer when the user click in it. The diagram below shows the appearance of the Viewer with this new flag.



Viewer with `kQ3ViewerPaneGrowBox` flag setting

NEW API FUNCTIONS

In addition to giving the user access to the rendering options listed above, the application using the Viewer also has access to the same things via new function calls:

Q3ViewerSetRendererType

```
OSErr      Q3ViewerSetRendererType(
                                    TQ3ViewerObject  theViewer,
                                    TQ3ObjectType rendererType)
```

Input:

`theViewer`	the viewer object who's renderer type you wish to change.
r`endererType`	the renderer type you want the Viewer to use.

Info: Use this function to set the Renderer Type you want the Viewer to use when rendering an image. You should only pass in Interactive Renderer types since other types of renderers may impede the user's ability to work with the Viewer. The two types of Interactive Renderers built into QuickDraw 3D are `kQ3RendererTypeInteractive` and `kQ3RendererTypeWireFrame`.

Q3ViewerGetRendererType

```
OSErr      Q3ViewerGetRendererType(
                                    TQ3ViewerObject    theViewer,
                                    TQ3ObjectType      *rendererType)
```

Input: `theViewer` — the viewer object who's renderer type you wish to set.

Output: r`endererType` — a pointer to a `TQ3ObjectType` that will contain the current Renderer Type assigned to the Viewer when the function completes.

Info: Use this function to get the Renderer Type currently being used by the Viewer to render images.

Q3ViewerChangeBrightness

```
OSErr     Q3ViewerChangeBrightness(
                                    TQ3ViewerObject theViewer,
                                     float          brightness)
```

Input: `theViewer` — a viewer object.

`brightness` — the percentage of light dimming to apply. The value 0 will dim all lights to 0%. A value of 1.0 will set all lights to their original values. A value of 2.0 will brighten all lights by 200%.

Info: This function lets you change the brightness of a scene by altering the brightness values of each light in the scene. The brightness value is a scaling value that is multiplied against each light in the scene. There is no upper limit to the brightness value, but it must be >= 0.0. Any value over 1.0 may cause some lights to oversaturate, but that might be the effect you want. For example, if your View contains a light that already has a brightness of 0.9 and you apply a brightness value of 1.2, then the brightened light will have a brightness of 1.08 which is slightly oversaturated.

Q3ViewerSetRemoveBackfaces

```
OSErr   Q3ViewerSetRemoveBackfaces(
                                    TQ3ViewerObject    theViewer,
                                     TQ3Boolean        remove)
```

Input:

`theViewer`	a viewer object.
`remove`	a boolean where kQ3True indicates to remove backfaces, kQ3False indicates keep backfaces.

Info: Calling this function will allow you to turn backface removal in the Viewer on and off. By default, backfaces are shown in the Viewer. This may have detrimental effects on the appearance and rendering speed of many kinds of models, therefore, you may wish to turn backface removal on in your application.

Q3ViewerGetRemoveBackfaces

```
OSErr   Q3ViewerGetRemoveBackfaces(
                                    TQ3ViewerObject theViewer,
                                    TQ3Boolean      *remove)
```

Input: `theViewer` a viewer object.

Output: remove a pointer to a `TQ3Boolean` will contain kQ3True if backface removal is currently turned on.

Info: This function returns `kQ3True` if backface removal is currently active in the viewer. Otherwise, it returns `kQ3False`.

Q3ViewerSetPhongShading

```
OSErr   Q3ViewerSetPhongShading(
                                    TQ3ViewerObject theViewer,
                                     TQ3Boolean     phong)
```

Input: `theViewer` a viewer object.

`phong` a boolean where kQ3True tells the Viewer to use phong shading, kQ3False tells the Viewer not to use phong shading.

Info: This function allows you to set Phong or Lambert shading in the Viewer. Passing in `kQ3True` activates the Phong shader, or passing in `kQ3False` tells the Viewer to use Lambert shading instead.

Q3ViewerGetPhongShading

```
OSErr   Q3ViewerGetPhongShading(
                                    TQ3ViewerObject theViewer,
                                     TQ3Boolean     *phong)
```

Input: `theViewer` a viewer object.

Output: `phong` a pointer to a `TQ3Boolean` which will contain kQ3True if Phong shading is currently turned on.

Info: This function returns `kQ3True` if Phong shading is currently being used by the Viewer to render scenes. Otherwise, a value of `kQ3False` indicates that the Lambert shader is being used.

Phong shading is turned on by default in the Viewer, however, some models do not look very good with Phong's specular highlight. The Lambert shader does not render with a specular highlight and usually renders faster.

Q3ViewerSetWindowResizeCallback

```
OSErr   Q3ViewerSetWindowResizeCallback (
                    TQ3ViewerObject                         theViewer,
                    TQ3ViewerWindowResizeCallbackMethod     callbackMethod,
                    const void                              *data)
```


Input:	`theViewer`	a viewer object.
	`callbackMethod`	a pointer to an application defined window resize method.
	`data`	an optional pointer to any application specific data which is passed to the callback function.
Output:	`TQ3Status`	`kQ3Success` if method was installed successfully.

Info: This function installs a callback method that gets invoked when the user clicks and drags in the window's grow box. Use this function if your application needs to resize the Viewer window differently than the default resizing functionality provided by the Viewer when the kQ3ViewerDrawGrowBox flag is set. The callback should handle all mouse tracking and resizing and invalidating of the window. The Viewer will redraw itself after the callback finishes.

The optional `data` parameter is used if an application needs to reference other information that can't be obtained from the Viewer object passed in the callback. If no extra data needed this parameter can be set to NULL.

To disable and remove the window resize callback function from your Viewer application, call `Q3ViewerSetWindowResizeCallback`, with a value of `NULL`.

Q3ViewerSetPaneResizeNotifyCallback

```
OSErr   Q3ViewerSetPaneResizeNotifyCallback (
                    TQ3ViewerObject                         theViewer,
                    TQ3ViewerPaneResizeNotifyCallbackMethod callbackMethod,
                    const void                              *data)
```


Input:	`theViewer`	a viewer object.
	`callbackMethod`	a pointer to an application defined method for resizing the Viewer as a pane independently from it enclosing window.
	`data`	an optional pointer to any application specific data which is passed to the callback function.
Output:	`TQ3Status`	`kQ3Success` if method was installed successfully.

Info: Calling this function installs a callback method that gets invoked when the user clicks and drags in the Viewer's pane grow box to resize the Viewer. Use this function if your application needs to resize the Viewer's pane dimensions independently of the window.

This callback is only invoked when the `kQ3ViewerPaneGrowBox` flag is set and overrides the window resizing functionality of the `kQ3ViewerDrawGrowBox` flag. After the user clicks in the pane grow box and the Viewer handles resizing the callback function should then erase and update any affected areas of the window and call Q3ViewerDraw to redraw the Viewer. Use the Q3ViewerGetBounds call to find the Viewer's new pane dimensions.

The optional `data` parameter is used if an application needs to reference other information that can't be obtained from the Viewer object passed in the callback. If no extra data needed this parameter can be set to NULL.

To disable and remove the Viewer pane resize callback function in your Viewer

application, call `Q3ViewerSetWindowResizeCallback`, with a value of `NULL`.

Q3ViewerSetDrawingCallbackMethod

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

Input:	`theViewer`	a viewer object.
	`callbackMethod`	a pointer to an application defined function.
	`data`	an optional pointer to any application specific data.
Output:	`TQ3Status`	`kQ3Success` if method was installed successfully.

Info: This function sets a callback method called after the Viewer finishes rendering the model and drawing the control strip. The callback function is called when the Viewer window is updated or when `Q3ViewerDraw` is called. The callback can so other things like draw over the rendered model or perform some other post operation. (This API isn't actually new for 1.6 but it is documented here for comparison with the other new Viewer callback API's.)

The optional `data` parameter is used if an application needs to reference other information that can't be obtained from the Viewer object passed in the callback. If no extra data needed this parameter can be set to NULL.

To disable and remove the drawing callback function from your Viewer application, call `Q3ViewerSetDrawingCallbackMethod`, with a value of `NULL`.

MISCELLANEA

INTERACTIVE RENDERER MODIFICATIONS

1. TriMesh culling is faster. TriMeshes of any number of vertices will be cull-tested now. Previously, only TriMeshes of 40 or more vertices were cull tested which resulted in small models being transformed in a slower pipeline.

2. The Interactive Renderer's memory management has been optimized.

3. PowerMac's with the extended PowerPC floating point opcodes (the 603, 604, and G3 chips) now do some calculations faster. Vector Normalizations are significantly faster.

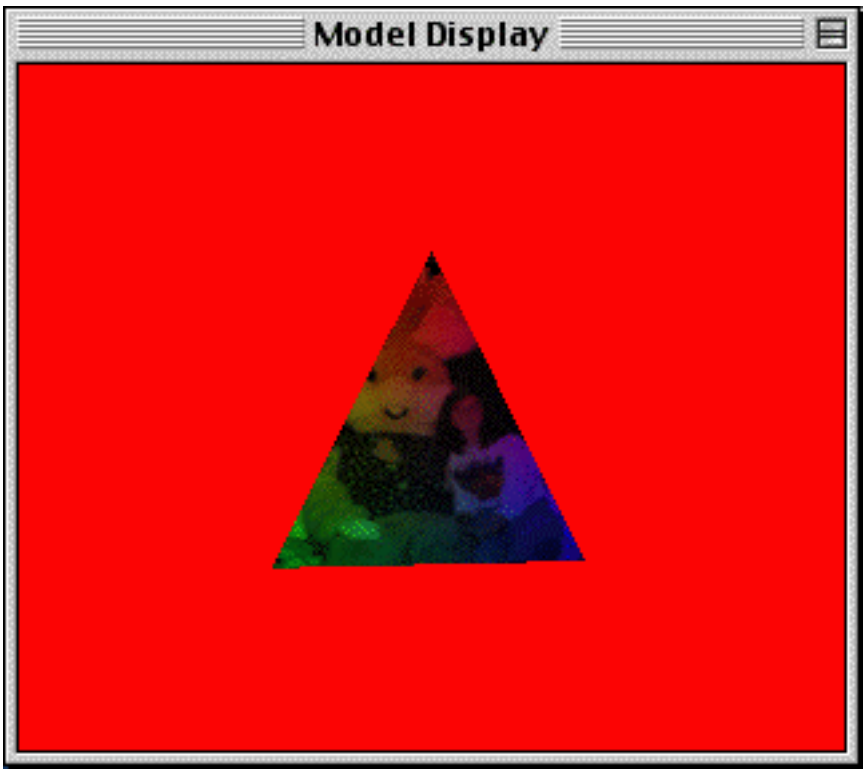
4. TriMesh transformation loops now specifically check for the Identity Matrix and if it exists, then no unnecessary transformation are calculated. Instead, data is simply copied from local space to world space. Therefore, it is more efficient to build your static geometries in world-space than to build them in local-space and use a transform to move them into world space. Building them in world-space allows QuickDraw 3D to avoid the local to world transform.

5. Many general optimizations to the transformation, lighting, and clipping code.

6. Changed the NULL Illumination model to always include vertex colors for shading

triangles. It makes sense that the NULL shader would use vertex colors since it has no way of calculating it's own vertex color values. Textured models can now also have vertex colors so you can essentially pre-light geometry for better performance. The RAVE Software rasterizer has also been updated to be able to render these textured-colored triangles.

7. Transparency now works correctly with the NULL shader using TriMeshes.



A single textured triangle with red, green, and blue diffuse colors applied to the vertices.

8. Fixed a bug that prevented a change in the View's Clear Method from working.

QUICKDRAW 3D MODIFICATIONS

1. Improved performance of Matrix Inversion code.

2. Improved performance of lots of low-level floating point math code.

3. Improved performance of most geometries. Conic decomposition is faster. Mesh decomposition is also faster.

4. Fixed `Q3InteractiveRenderer_SetRAVETextureFilter` so it actually works now.

5. The default diffuse color is now 1,1,1 instead of .5,.5,.5. Since 1.6 now blends diffuse colors with textures when the NULL shader is used (see #6 above), this default color needed to be changed to white instead of gray, otherwise textures would be 50% dimmed when the NULL shader is used.. If you are not using the NULL shader in your app and you want the default diffuse to be gray instead of white, then you can use the `Q3View_GetDefaultAttributeGet` and `Q3View_GetDefaultAttributeSet` calls to change the default diffuse RGB values.

6. Performance for window point picking has been improved for all geometries, notably trimesh, mesh, and trigrid. Initial test show improvements are 20-50%, 10-25% and 30-85% respectively, depending on the triangle count.