QuickDraw 3D RAVE

Contents

About QuickDraw 3D RAVE 1-8
- Drawing Engines 1-10
- Draw Contexts 1-12

Using QuickDraw 3D RAVE 1-13
- Specifying a Virtual Device 1-14
- Finding a Drawing Engine 1-16
- Creating and Configuring a Draw Context 1-17
- Drawing in a Draw Context 1-19
- Using A Draw Context as a Cache 1-20
- Using a Texture Map Alpha Channel 1-21
- Rendering With Antialiasing 1-22

Writing a Drawing Engine 1-23
- Writing Public Draw Context Methods 1-24
- Writing Private Draw Context Methods 1-26
- Handling Gestalt Selectors 1-27
- Registering a Drawing Engine 1-29
- Supporting OpenGL Hardware 1-30
  - Transparency 1-30
  - Texture Mapping 1-33

QuickDraw 3D RAVE Reference 1-34
- Constants 1-34
  - Version Values 1-35
  - Pixel Types 1-35
  - Color Lookup Table Types 1-37
  - Device Types 1-38
  - Clip Types 1-38
  - Tags for State Variables 1-38

Z Sorting Function Selectors 1-47
Antialiasing Selectors 1-48
Blending Operations 1-49
Z Perspective Selectors 1-50
Texture Filter Selectors 1-51
Texture Operations 1-51
CSG IDs 1-53
Texture Wrapping Values 1-54
Source Blending Values 1-54
Destination Blending Values 1-55
Buffer Drawing Operations 1-55
Vertex Modes 1-56
Gestalt Selectors 1-57
Gestalt Optional Features Response Masks 1-59
Gestalt Fast Features Response Masks 1-61
Vendor and Engine IDs 1-62
Triangle Flags Masks 1-63
Texture Flags Masks 1-64
Bitmap Flags Masks 1-64
Draw Context Flags Masks 1-65
Drawing Engine Method Selectors 1-65
Public Draw Context Method Selectors 1-67
Notice Method Selectors 1-68
Data Structures 1-69
Memory Device Structure 1-69
Rectangle Structure 1-70
Macintosh Device and Clip Structures 1-70
Windows Device and Clip Structures 1-71
Generic Device and Clip Structures 1-71
Device Structure 1-72
Clip Data Structure 1-72
Image Structure 1-73
Vertex Structures 1-73
Draw Context Structure 1-77
Indexed Triangle Structure 1-80
QuickDraw 3D RAVE Routines 1-81
Creating and Deleting Draw Contexts 1-81
`QADrawContextNew` 1-81

QADrawContextDelete 1-82
Creating and Deleting Color Lookup Tables 1-82
QAColorTableNew 1-83
QAColorTableDelete 1-84
Manipulating Textures and Bitmaps 1-85
QATextureNew 1-85
QATextureDetach 1-86
QATextureBindColorTable 1-87
QATextureDelete 1-87
QABitmapNew 1-88
QABitmapDetach 1-89
QABitmapBindColorTable 1-90
QABitmapDelete 1-90
Managing Drawing Engines 1-91
QADeviceGetFirstEngine 1-91
QADeviceGetNextEngine 1-92
QAEngineCheckDevice 1-92
QAEngineGestalt 1-93
QAEngineEnable 1-94
QAEngineDisable 1-94
Manipulating Draw Contexts 1-94
QAGetFloat 1-95
QASetFloat 1-95
QAGetInt 1-96
QASetInt 1-97
QAGetPtr 1-97
QASetPtr 1-98
QADrawPoint 1-98
QADrawLine 1-99
QADrawTriGouraud 1-99
QADrawTriTexture 1-100
QASubmitVerticesGouraud 1-101
QASubmitVerticesTexture 1-101
QADrawTriMeshGouraud 1-102
QADrawTriMeshTexture 1-103
QADrawVGouraud 1-104
QADrawVTexture 1-105
QADrawBitmap 1-106

QARenderStart 1-107
QARenderEnd 1-108
QARenderAbort 1-109
QAFlush 1-110
QASync 1-111
QAGetNoticeMethod 1-112
QASetNoticeMethod 1-112
Registering a Custom Drawing Engine 1-113
QARegisterEngine 1-113
QARegisterDrawMethod 1-114
Application-Defined Routines 1-115
Public Draw Context Methods 1-115
TQAGetFloat 1-116
TQASetFloat 1-116
TQAGetInt 1-117
TQASetInt 1-118
TQAGetPtr 1-119
TQASetPtr 1-119
TQADrawPoint 1-120
TQADrawLine 1-121
TQADrawTriGouraud 1-121
TQADrawTriTexture 1-122
TQASubmitVerticesGouraud 1-123
TQASubmitVerticesTexture 1-124
TQADrawTriMeshGouraud 1-125
TQADrawTriMeshTexture 1-126
TQADrawVGouraud 1-127
TQADrawVTexture 1-128
TQADrawBitmap 1-129
TQARenderStart 1-130
TQARenderEnd 1-131
TQARenderAbort 1-132
TQAFlush 1-133
TQASync 1-134
TQAGetNoticeMethod 1-134
TQASetNoticeMethod 1-135
Private Draw Context Methods 1-136
TQADrawPrivateNew 1-136

TQADrawPrivateDelete 1-137
TQAEngineCheckDevice 1-138
TQAEngineGestalt 1-138
Color Lookup Table Methods 1-139
TQAColorTableNew 1-139
TQAColorTableDelete 1-140
Texture and Bitmap Methods 1-141
TQATextureNew 1-141
TQATextureDetach 1-142
TQATextureBindColorTable 1-143
TQATextureDelete 1-143
TQABitmapNew 1-144
TQABitmapDetach 1-145
TQABitmapBindColorTable 1-145
TQABitmapDelete 1-146
Method Reporting Methods 1-146
TQAEngineGetMethod 1-147
Notice Methods 1-147
TQANoticeMethod 1-147
Summary of QuickDraw 3D RAVE 1-149
C Summary 1-149
Constants 1-149
Data Types 1-158
QuickDraw 3D RAVE Routines 1-163
Application-Defined Routines 1-166
Result Codes 1-170

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

Simultaneously published in the United States and Canada.

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.

This chapter describes the QuickDraw 3D Renderer Acceleration Virtual Engine (RAVE), the part of the Macintosh system software that controls 3D drawing engines (also known as 3D drivers). As explained more fully below, a drawing engine is software that supports the low-level rasterization operations required for interactive 3D rendering. To achieve interactive performance, a drawing engine is often associated with some hardware device designed specifically to accelerate the 3D rasterization process.

QuickDraw 3D RAVE is used internally by QuickDraw 3D, the 3D graphics library from Apple Computer, Inc. that you can use to create, configure, render, and interact with models of three-dimensional objects. For most 3D drawing and interaction, you should use the high-level application programming interfaces provided by QuickDraw 3D. In some cases, however, you might need to use the low-level services provided by QuickDraw 3D RAVE. You can use QuickDraw 3D RAVE if

- you are writing a specialized application (such as a game-development framework) that needs to take advantage of Apple's optimized software rasterizers and any available 3D acceleration hardware
- you are writing interactive software (such as a game or other entertainment software) that requires the extremely fast 3D rendering that can be achieved with a very low-level, lightweight graphics library
- you are developing 3D acceleration hardware or software that is to be accessed by any applications rendering 3D images

To use this chapter, you should already be familiar with QuickDraw 3D, described in *3D Graphics Programming With QuickDraw 3D*. You should also be familiar with low-level 3D rendering algorithms. The bibliography (page BI-171) lists a number of standard 3D reference books that document those algorithms.

This chapter begins by describing the basic capabilities of QuickDraw 3D RAVE. Then it shows how to use some of those capabilities to find the available drawing engines, select and configure a drawing engine, and use that drawing engine to draw 3D images. The section "Writing a Drawing Engine," beginning on page 1-23, shows how to add a new drawing engine to those already available for use by QuickDraw 3D RAVE. You need to read this section only if you are developing custom 3D acceleration hardware or software.

The section "QuickDraw 3D RAVE Reference," beginning on page 1-34, provides a complete reference to the constants, data structures, and functions provided by QuickDraw 3D RAVE.

IMPORTANT

QuickDraw 3D RAVE is used by the interactive renderer supplied as part of QuickDraw 3D version 1.0. However, the features described here that provide compatibility with **OpenGL**™ are not supported by that renderer and are subject to change in future versions of QuickDraw 3D RAVE. ▲

About QuickDraw 3D RAVE

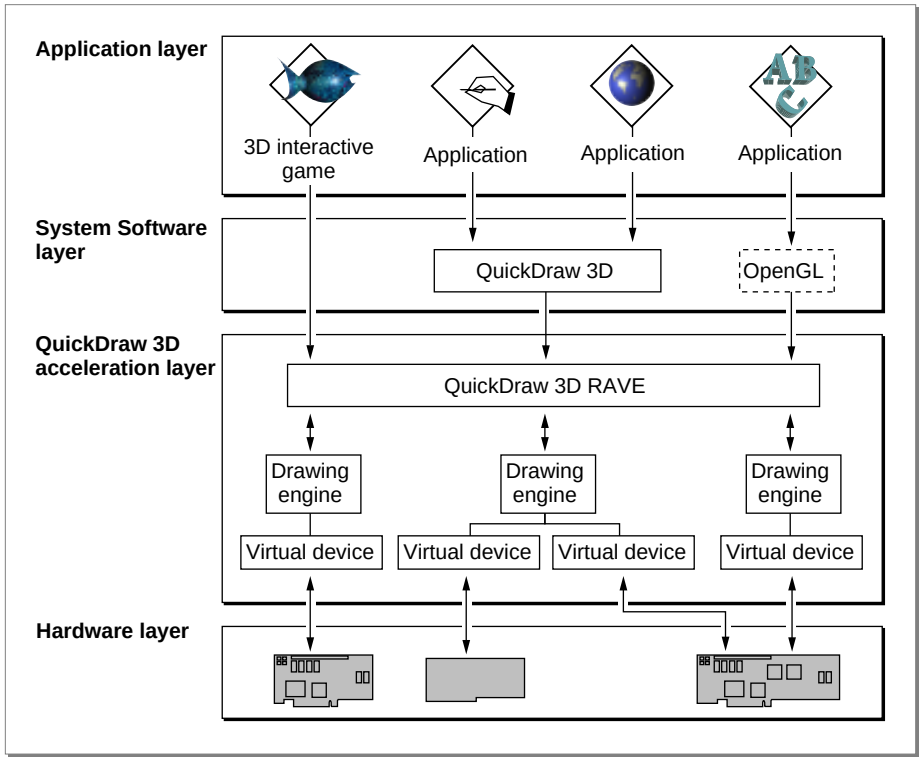
The **QuickDraw 3D Renderer Acceleration Virtual Engine** (or, more briefly, **QuickDraw 3D RAVE**) is the part of the Macintosh system software that controls 3D drawing engines. A **drawing engine** is software that supports low-level rasterization operations—that is, the process of determining values for pixels in an image on the screen or some other medium. You are probably already familiar with QuickDraw, which is a 2D drawing engine. The 3D drawing engines managed by QuickDraw 3D RAVE differ from 2D drawing engines in several important respects:

- A 3D drawing engine must support a *z* (or depth) value for **hidden surface removal** (removing any surfaces in a model that are hidden by opaque surfaces of objects).
- A 3D drawing engine typically supports **double buffering,** the use of two different buffers to store pixel images. Double buffering helps reduce the flashing caused by redrawing an image. Double buffering can also be used to avoid tearing artifacts caused by updating a window at high speed.
- A 3D drawing engine typically supports special rasterization modes, such as texture mapping or constructive solid geometry.

In almost all other respects, a 3D drawing engine operates just like a 2D drawing engine. You draw objects by sending drawing commands to the drawing engine, which interprets the commands and constructs a rasterized image. A 3D drawing engine is often associated with hardware designed specifically to accelerate the 3D rasterization process.

Figure 1-1 illustrates the position of QuickDraw 3D RAVE in relation to drawing engines, the clients that call it, and the devices driven by the drawing engines.

Figure 1-1 The position of QuickDraw 3D RAVE



QuickDraw 3D RAVE and all registered drawing engines with their associated devices comprise the **QuickDraw 3D Acceleration Layer**. As you can see, this layer operates as a hardware abstraction layer that insulates the system software (for instance, QuickDraw 3D) or other clients from the actual video display hardware and graphics acceleration hardware available on a particular Macintosh computer.

Most applications creating 3D images should use QuickDraw 3D, which determines the best drawing engine and associated output device to use to display an image. QuickDraw 3D calls that drawing engine, using functions provided by QuickDraw 3D RAVE. As a result, most applications do not need to know about the QuickDraw 3D Acceleration Layer. Instead, they should use

high-level 3D system software (such as QuickDraw 3D or OpenGL) to create and render 3D models.

Occasionally, however, some software (like the 3D game shown in Figure 1-1) needs interactive performance but only limited 3D rendering capabilities. In these cases, the software can call QuickDraw 3D RAVE functions directly, to find and configure a drawing engine and to issue drawing commands.

The QuickDraw 3D Acceleration Layer is intended to provide a basis for 3D rendering at interactive speeds. Accordingly, QuickDraw 3D RAVE is implemented in such a way as to minimize the overhead incurred by communication between an application and a drawing engine. In particular, a function call from an application to QuickDraw 3D RAVE does not require a context change. In addition, a function call from an application to a drawing engine does not require intermediate processing by QuickDraw 3D RAVE. Instead, drawing calls are implemented as C language macros that call directly into the code of a drawing engine. (See “Manipulating Draw Contexts” (page 1-94) for details.)

IMPORTANT

As a result of these two features, calling a drawing engine through QuickDraw 3D RAVE provides the same level of performance as linking the engine directly with the calling application. ▲

Drawing Engines

A drawing engine is a plug-in software module that accepts drawing commands and produces a rasterized image. QuickDraw 3D RAVE is designed to make it easy for you to add drawing engines to those already available. When you register a drawing engine, it thereby becomes available for use by any application or system software running on a Macintosh computer.

QuickDraw 3D RAVE expects that a drawing engine will have a certain minimum set of required features and possibly one or more optional features. Every drawing engine must provide these features:

- hidden surface removal (usually accomplished using z buffering with at least 16 bits per pixel)
- point and line drawing, with application-specifiable point and line widths
- drawing of Gouraud-shaded triangles

- drawing of bitmaps having depths of 1, 16, or 32 bits per pixel
- support for double buffering

In addition to the required features, a drawing engine may support one or more of these optional features:

- high-precision hidden surface removal (using z buffering with at least 24 bits per pixel)
- perspective-corrected hidden surface removal
- texture mapping
- triangle meshes (a memory and time optimization that allows rendered triangles to share vertices)
- transparency blending, with or without an alpha channel
- antialiasing
- z-sorted rendering of non-opaque objects
- support for OpenGL features (such as scissoring, multiple blending modes, area and line stipple patterns, and so forth)

The interactive renderer supplied as part of QuickDraw 3D uses a software-only drawing engine that can draw to any available device. In addition to the required features listed earlier, the drawing engine supplied with the interactive renderer supports these optional features:

- z buffering with 16 or 32 bits per pixel
- direct rendering at 16 or 32 bits per pixel (rendering at fewer than 16 bits per pixel is also supported, but with lower performance)
- perspective-corrected texture mapping

It's important to keep in mind that a drawing engine is a low-level 3D driver and hence does not support some features found in higher-level interfaces. The current programming interfaces to drawing engines do not support any of these features:

- transformations, shading, or clipping
- I/O support (such as reading and writing 3D metafiles)
- high-level primitives (such as curved surfaces)
- support for drawing to windows that straddle two or more devices

IMPORTANT

Because of these limitations, most applications should not use QuickDraw 3D RAVE directly. Instead, you should use the high-level programming interfaces provided by QuickDraw 3D or other system software that provides 3D capabilities. ▲

QuickDraw 3D RAVE does not require that a drawing engine be capable of drawing to all devices available on a particular computer. Rather, a particular drawing engine may support only a single output device. For example, a drawing engine that uses a frame buffer's built-in 3D acceleration hardware may be incapable of rendering to any other device. As a result, QuickDraw 3D RAVE won't allow some other device to be associated with that drawing engine. This means that QuickDraw 3D RAVE does not provide automatic support for drawing into windows that cross multiple devices. Instead, it is the application's responsibility to determine when a window does straddle devices and to construct multiple draw contexts (described next) for the output image.

Draw Contexts

Although a drawing engine may be capable of supporting more than one device, it cannot divide a raster across multiple devices. Instead, every drawing command sent to a drawing engine must be destined for a single device. QuickDraw 3D RAVE guarantees this by requiring a calling application to specify a draw context as a parameter for every drawing command. A **draw context** is a structure (of type `TQADrawContext`) that maintains state information and other data associated with a particular drawing engine and device.

As mentioned at the end of the previous section, you need to create several draw contexts if you want to draw into a window that spans several devices. Similarly, you need to create several draw contexts if you want to draw into several different windows on the same device. Each draw context maintains its own state information image buffers and is unaffected by any functions that operate on another draw context.

The state information associated with a draw context is maintained using a large number of **state variables.** For example, the background color of a draw context is specified by four state variables, designated by the four identifiers (or **tags**) `kQATag_ColorBG_a`, `kQATag_ColorBG_r`, `kQATag_ColorBG_g`, and `kQATag_ColorBG_b`. See "Creating and Configuring a Draw Context," beginning on page 1-17, for some sample code that reads and sets state variables, and

"Tags for State Variables," beginning on page 1-38, for a complete list of the available state variables.

A hardware device (such as a frame buffer or a video interface) is represented in QuickDraw 3D RAVE by a **virtual device,** a structure of type `TQADevice` that determines which one of a variety of types of hardware devices a draw context draws into. On Macintosh computers, QuickDraw 3D RAVE supports two kinds of virtual devices: memory devices and graphics devices. A **memory device** represents an area of memory, and a **graphics device** represents a video device (such as a plug-in video card or built-in video interface) that controls a screen, or an offscreen graphics world (which allows your application to build complex images off the screen before displaying them). In effect, a virtual device specifies the buffers into which all drawing commands associated with a draw context write pixels.

Using QuickDraw 3D RAVE

This section illustrates how to use QuickDraw 3D RAVE. In particular, it provides source code examples that show how you can

- specify a virtual device
- determine which drawing engines are available and what features they have
- create and configure a draw context
- draw objects in a draw context
- use a draw context as an image cache
- use a texture map's alpha channel for transparency or as a blend matte
- render with antialiasing

These are examples of operations that an application might need to perform. To learn how to write and register a new drawing engine, see the section "Writing a Drawing Engine," beginning on page 1-23.

Note
The code examples shown in this section provide only very rudimentary error handling. ◆

Specifying a Virtual Device

You send all drawing commands to a draw context. To create a draw context, you need to specify a virtual device and a drawing engine. This section shows how to initialize a virtual device. See the next section for information on specifying a drawing engine.

On Macintosh computers, a virtual device represents either an area of memory, a video device, or an offscreen graphics world. You specify a virtual device by filling in fields of a **device structure,** defined by the `TQADevice` data type.

```
typedef struct TQADevice {
    TQADeviceType                   deviceType;
    TQAPlatformDevice               device;
} TQADevice;
```

The `deviceType` field indicates the type of virtual device you want to draw into. Currently, you can pass either `kQADeviceMemory` or `kQADeviceGDevice` to select a Macintosh device type. The `device` field indicates a platform device data structure, which is either of type `TQADeviceMemory` for memory devices or `GDHandle` for graphics devices.

```
typedef union TQAPlatformDevice {
    TQADeviceMemory                 memoryDevice;
    GDHandle                        gDevice;
} TQAPlatformDevice;
```

To specify a memory device, you fill in the fields of a **memory device structure,** defined by the `TQADeviceMemory` data type.

```
typedef struct TQADeviceMemory {
    long                            rowBytes;
    TQAImagePixelType               pixelType;
    long                            width;
    long                            height;
    void                            *baseAddr;
} TQADeviceMemory;
```

Listing 1-1 shows how to initialize a memory device.

Listing 1-1 Initializing a memory device

```
TQADevice       myDevice;
long            myTargetMemory[100][100];

myDevice.deviceType = kQADeviceMemory;
myDevice.device.memoryDevice.rowBytes = 100 * sizeof(long);
myDevice.device.memoryDevice.pixelType = kQAPixel_ARGB32;
myDevice.device.memoryDevice.width = 100;
myDevice.device.memoryDevice.height = 100;
myDevice.device.memoryDevice.baseAddr = myTargetMemory;
```

Drawing to memory always occurs in the native pixel format of the platform. Note that not all drawing engines support drawing to memory. For information on determining what kinds of virtual devices a particular drawing engine supports, see “Finding a Drawing Engine” (page 1-16).

Listing 1-2 shows how to initialize a virtual graphics device.

Listing 1-2 Initializing a graphics device

```
TQADevice       myDevice;
GDHandle        gDeviceHandle;

/*create a GDHandle (perhaps by calling NewGDevice)*/
...
myDevice.deviceType = kQADeviceGDevice;
myDevice.device.gDevice = gDeviceHandle;
```

The code in Listing 1-2 assumes that the `gDeviceHandle` global variable has been assigned a handle to a `GDevice` record. See *Inside Macintosh: Imaging With QuickDraw* for complete information on creating and configuring graphics devices.

Note
A draw context can be associated with only a single virtual device and hence with only a single `GDevice`. Macintosh windows can straddle several screens, each associated with a different `GDevice`. It is your responsibility to determine which graphics devices a window straddles and to create a separate draw context for each one. ◆

Finding a Drawing Engine

Not all drawing engines are capable of drawing into all type of virtual devices. For example, some drawing engines might not support memory devices at all, and other drawing engines might support only a particular graphics device. As a result, once you've initialized a virtual device, you need to find a drawing engine that is capable of drawing into that device. You do this by finding the available drawing engines and selecting one that is capable of drawing into the desired virtual device. If more than one engine supports that device, you need to choose one of them.

QuickDraw 3D versions 1.1 and later provide a control panel that allows the user to select the drawing engine to use for each available monitor.

You can search through the list of available drawing engines by calling the `QADeviceGetFirstEngine` and `QADeviceGetNextEngine` functions. The `QADeviceGetFirstEngine` function returns the preferred drawing engine for the specified device; in most cases, this engine is the best engine to use for high performance rendering. However, you might need specific drawing features that are not supported by the preferred drawing engine. If so, you can use the `QAEngineGestalt` function to query the engine's capabilities, as shown in Listing 1-3.

Listing 1-3 Finding a drawing engine with fast texture mapping

```
TQAEngine *MyFindPreferredEngine (TQADevice *device)
{
    TQAEngine               *myEngine;
    unsigned long           fast;

    for (myEngine = QADeviceGetFirstEngine(device);
            myEngine;
```

```
            myEngine = QADeviceGetNextEngine(device, myEngine)) {
        if (QAEngineGestalt(myEngine, kQAGestalt_FastFeatures, &fast) == kQANoErr) {
            if (fast & kQAFast_Texture)
                return(myEngine);
        }
    }
    return(NULL);
}
```

The `MyFindPreferredEngine` function defined in Listing 1-3 calls the `QADeviceGetFirstEngine` function to get the preferred drawing engine for the specified device. Then it calls `QAEngineGestalt`, passing the `kQAGestalt_FastFeatures` selector, to determine which (if any) features are accelerated by that engine. If the engine supports accelerated texture mapping, the `MyFindPreferredEngine` function returns that drawing engine. Otherwise, the `MyFindPreferredEngine` function loops through all engines capable of drawing into the specified device until it finds one that does support fast texture mapping. If none is found, `MyFindPreferredEngine` returns the value `NULL`.

Note
See "Gestalt Selectors" (page 1-57) for a complete description of the selectors you can pass to the `QAEngineGestalt` function. ◆

Creating and Configuring a Draw Context

Once you've initialized a virtual device and selected a drawing engine capable of drawing to that device, you can call the `QADrawContextNew` function to create a new draw context. You pass the device and engine to that function, along with a drawing rectangle, a clipping region, and a set of draw context flags. The flags specify features of the new draw context. Listing 1-4 illustrates how to create a double-buffered draw context with z buffering.

Listing 1-4 Creating a draw context

```
TQADrawContext          *myDrawContext;

if (QADrawContextNew(&myDevice, &myRect, &myClip, myEngine,
```

```
            kQAContext_DoubleBuffer, &myDrawContext) != kQANoErr) {
    /*Error! Could not create new draw context.*/
}
```

If `QADrawContextNew` succeeds, it returns the result code `kQANoErr` and sets the `myDrawContext` parameter to the new draw context. Otherwise, if an error occurs, `QADrawContextNew` returns some other result code and sets the `myDrawContext` parameter to the value `NULL`.

Note
When you are finished using the new draw context, you should free the memory and other resources it uses by calling the `QADrawContextDelete` function. ◆

QuickDraw 3D RAVE does not provide a function to reposition an existing draw context. If a window associated with a draw context is moved on the screen, you need to delete the existing draw context and create a new draw context at the new location. Similarly, QuickDraw 3D RAVE does not provide a function to change the clipping region of a draw context. If you want to change a clipping region, you need to delete the existing draw context and create a new draw context with the desired clipping region.

However, you can change a number of other features of a draw context without having to delete an existing draw context and create a new one. The features you can change are indicated by the state variables of the draw context. For example, to change the background color of a draw context to opaque black, you can use the code shown in Listing 1-5.

Listing 1-5 Setting a draw context state variable

```
void MySetBackgroundToBlack (TQADrawContext *drawContext);
{
    QASetFloat(drawContext, kQATag_ColorBG_a, 1.0);
    QASetFloat(drawContext, kQATag_ColorBG_r, 0.0);
    QASetFloat(drawContext, kQATag_ColorBG_g, 0.0);
    QASetFloat(drawContext, kQATag_ColorBG_b, 0.0);
}
```

The `QASetFloat` function sets a draw context state variable that has a floating-point value. QuickDraw 3D RAVE provides functions to get and set state variables with floating-point, long integer, or pointer values.

Note
See "Tags for State Variables," beginning on page 1-38, for a complete description of the available draw context state variables. ◆

The `QASetFloat` function is defined using a C language macro:

```
#define QASetFloat(drawContext,tag,newValue) \
                (drawContext)->setFloat (drawContext,tag,newValue)
```

During compilation, the `QASetFloat` call is replaced by code that directly calls the drawing engine's floating-point setting method. This allows you to achieve the highest possible performance when configuring a draw context.

Drawing in a Draw Context

QuickDraw 3D RAVE allows you to draw five kinds of objects in a draw context: points, lines, triangles, triangle meshes, and bitmaps. You draw by calling a function to draw the desired type of object. For instance, to draw a single point, you can call the `QADrawPoint` function, as follows:

```
QADrawPoint(myDrawContext, myPoint);
```

Here, the `myPoint` parameter specifies the point to draw. All objects that a drawing engine can draw (except for bitmaps) are defined by points or vertices. QuickDraw 3D RAVE supports two different types of vertices: Gouraud vertices and texture vertices. You use **Gouraud vertices** for drawing Gouraud-shaded triangles, and also for drawing points and lines. A Gouraud vertex is defined by the `TQAVGouraud` data structure, which specifies the position, depth, color, and transparency information.

You use **texture vertices** to define triangles to which a texture is to be mapped. A texture vertex is defined by the `TQAVTexture` data structure, which specifies the position, depth, transparency, and texture mapping information.

IMPORTANT

QuickDraw 3D RAVE does not currently support clipping to a draw context. All triangles and other objects drawn to a draw context must lie entirely within the draw context. ◆

Using a Draw Context as a Cache

QuickDraw 3D RAVE supports draw context caching, a technique that allows you to improve rendering performance when a large number of the objects in a scene don't change from frame to frame. A **draw context cache** is simply a draw context that contains an image and is designated as the initial context in a call to `QARenderStart`. The contents of that context are drawn into the destination draw context before any other objects.

To create a draw context cache, you first create a draw context by calling the `QADrawContextNew` function, where the `flags` parameter has the `QAContext_Cache` flag set. Then you draw the unchanging objects into the draw context cache. For example, suppose that you want to draw a series of frames in which two triangles remain constant from frame to frame but a third triangle changes every frame. Listing 1-6 shows how to do this.

Listing 1-6 Creating and using a draw context cache

```
TQAVGouraud                 tri1[3], tri2[3], tri3[3];
TQADrawContext              *myCache, *myDest;

/*Create draw context cache and destination draw context.*/
QADrawContextNew(myDev, rect, NULL, myEng, QAContext_Cache, &myCache);
QADrawContextNew(myDev, rect, NULL, myEng, QAContext_DoubleBuffer, &myDest);

/*Set up the image in the cache context.*/
QARenderStart(myCache, NULL, NULL);
QADrawTriGouraud(myCache, &tri1[0], &tri1[1], &tri1[2], kQATriFlags_None);
QADrawTriGouraud(myCache, &tri2[0], &tri2[1], &tri2[2], kQATriFlags_None);
QARenderEnd(myCache, NULL);

/*Render frames using the cache and moving tri3 only.*/
while (gStillMovingTriangle3) {
    MyMoveTri(tri3);
    QARenderStart(myDest, NULL, myCache);
```

```
    QADrawTriGouraud(myDest, &tri3[0], &tri3[1], &tri3[2], kQATriFlags_None);
    QARenderEnd(myDest, NULL);
}
```

Not all drawing engines support draw context caching. If a drawing engine does not support caching, it should return the value `NULL` whenever you pass the `QAContext_Cache` flag to `QADrawContextNew`.

IMPORTANT

All draw context caches must be single buffered, and they must be created using the same device and rectangle as the destination draw contexts with which they will be used. ▲

Using a Texture Map Alpha Channel

Texture maps whose pixel type is either `kQAPixel_ARGB16` or `kQAPixel_ARGB32` contain an alpha channel value for each pixel in the map. You can use the alpha channel value to control the transparency of an object on a pixel-by-pixel basis, or you can use the alpha channel value as a blend matte that exposes only certain portions of an image.

To use the alpha channel to control transparency, you should set the drawing engine's transparency blending mode to `kQABlend_Premultiply`. (You specify an engine's transparency blending mode by assigning a value to its `kQATag_Blend` state variable.) For pixels of type `kQAPixel_ARGB16`, the alpha channel value occupies bit 15; when the value is 1, the pixel is opaque; when the value is 0, the pixel is completely transparent. For pixels of type `kQAPixel_ARGB32`, the alpha channel value occupies bits 31 through 24; when the value is 255, the pixel is opaque; when the value is 0, the pixel is completely transparent.

IMPORTANT

The `kQABlend_Premultiply` transparency model assumes that the diffuse color of a pixel has been premultiplied by the alpha channel value. As a result, every pixel of the texture map must be premultiplied by its associated alpha channel value before you create the texture map by calling `QATextureNew`. ▲

Note that the specular highlight is unaffected by the diffuse transparency of an object. As a result, setting an object's alpha channel value to 0 when using the

kQABlend_Premultiply transparency blending mode does not cause the object to vanish. The specular highlight is still rendered.

To use the alpha channel as a **blend matte** to cut out certain portions of a rendered object, you should set the drawing engine's transparency blending mode to kQABlend_Interpolate. If the alpha channel value of all pixels in an object is 0, neither the object nor its specular highlight is rendered. This effectively eliminates the object from the rendered image.

IMPORTANT

The kQABlend_Interpolate transparency model assumes that the diffuse color of a pixel has *not* been premultiplied by the alpha channel value. This multiplication is performed by the blending operation. ▲

The kQABlend_Interpolate blending mode cannot render a transparent surface as accurately as the kQABlend_Premultiply mode, because the specular highlight is scaled by the alpha value. In some cases, you can compensate for this behavior by increasing the brightness of the specular highlight.

Rendering With Antialiasing

A drawing engine may support an **antialiasing mode** that determines the kind of antialiasing applied to a drawing context. (**Antialiasing** is the smoothing of jagged edges on a displayed shape by modifying the transparencies of individual pixels along the shape's edge.)

Note

The antialiasing mode state variable is optional; it must be supported only when a drawing engine supports the kQAOptional_Antialias feature. ◆

You specify an engine's antialiasing mode by assigning a value to its kQATag_Antialias state variable. QuickDraw 3D RAVE provides these constants for antialiasing modes:

```
#define kQAAntiAlias_Off                    0
#define kQAAntiAlias_Fast                   1
#define kQAAntiAlias_Mid                    2
#define kQAAntiAlias_Best                   3
```

The interpretation of these values is specific to each drawing engine. For example, a drawing engine might be able to support antialiased line drawing with no performance penalty but that same engine might incur a 50 percent slowdown when drawing antialiased triangles. Accordingly, this engine might interpret the `kQAAntiAlias_Fast` antialiasing mode as rendering antialiased lines only, and it might interpret the `kQAAntiAlias_Mid` mode as rendering both antialiased lines and triangles.

Note
QuickDraw 3D RAVE interprets antialiasing modes independently of the transparency blending modes, unlike some other rendering technologies. For instance, with OpenGL you must select specific blending modes when antialiasing is enabled. ◆

Writing a Drawing Engine

This section shows how to write a new drawing engine and add it to the QuickDraw 3D Acceleration Layer.

IMPORTANT
You need to read this section only if you are developing custom 3D acceleration hardware or software. If you simply want to create a draw context and draw into it using low-level drawing functions, see "Using QuickDraw 3D RAVE," beginning on page 1-13. ▲

To develop a new drawing engine and add it to the QuickDraw 3D Acceleration Layer, you need to perform these seven steps:

1. Write methods for the public functions pointed to by the fields of a draw context structure (for example, `setInt`). These methods are described in detail in the section "Public Draw Context Methods," beginning on page 1-115.
2. Write methods for the `TQADrawPrivateNew` and `TQADrawPrivateDelete` function prototypes. These functions are called internally by the `QADrawContextNew` and `QADrawContextDelete` functions, respectively. You use these methods to allocate and release any private data (such as state variables) maintained by

your drawing engine. These methods are described in detail in the section "Private Draw Context Methods," beginning on page 1-136.

3. Write methods for any texture and bitmap functions supported by your drawing engine (`TQATextureNew`, `TQATextureDetach`, `TQATextureDelete`, `TQABitmapNew`, `TQABitmapDetach`, and `TQABitmapDelete`). These functions are called by their public counterparts (for example, `QABitmapNew`). These methods are described in detail in the section "Texture and Bitmap Methods," beginning on page 1-141.
4. Write a method to handle the `QAEngineGestalt` function when your drawing engine is the target engine. This method is described in detail on (page 1-138).
5. Write a method to handle the `QAEngineCheckDevice` function when your drawing engine is the target engine. QuickDraw 3D RAVE calls this method to determine which devices your drawing engine supports. This method is described in detail on (page 1-138).
6. Write a method for the `TQAEngineGetMethod` function prototype. QuickDraw 3D RAVE calls this method to get some of your engine's methods during engine registration. This method is described in detail on (page 1-147).
7. Build your code as a shared library. The initialization routine of the shared library should register your drawing engine with QuickDraw 3D RAVE by calling the `QARegisterEngine` function.

The following sections describe some of these steps in more detail. The section "Supporting OpenGL Hardware," beginning on page 1-30 contains information that is useful if you are implementing a drawing engine to support hardware that is based on an OpenGL rasterization model.

Writing Public Draw Context Methods

As you've seen, the draw context structure (of type `TQADrawContext`) contains function pointers to the public draw context methods supported by your drawing engine. These methods are called whenever an application calls one of the public functions provided by QuickDraw 3D RAVE. For example, when an application calls the `QADrawPoint` function for a draw context associated with

your drawing engine, your engine's `TQADrawPoint` method (pointed to by the `drawPoint` field) is called. The `TQADrawPoint` method is declared like this:

```
typedef void (*TQADrawPoint) (
                    const TQADrawContext *drawContext,
                    const TQAVGouraud *v);
```

A draw context structure is passed as the first parameter to all the public draw context methods you need to define. This allows your methods to find the private data associated with the draw context (which is pointed to by the `drawPrivate` field).

Notice that the function prototype for a point-drawing method passes the draw context as a `const` parameter. This indicates that your method should not alter any of the fields of the draw context structure passed to it. Only three draw context methods (namely `TQASetInt`, `TQASetFloat`, and `TQASetPtr`) are allowed to alter the draw context.

Listing 1-7 shows a sample definition for a point-drawing method.

Listing 1-7 A `TQADrawPoint` method

```
void MyDrawPoint (const TQADrawContext *drawContext, const TQAVGouraud *v)
{
    MyPrivateData               *myData;    /*our actual private data type*/

    /*Cast generic drawPrivate pointer to our actual private data type.*/
    myData = (MyPrivateData *) drawContext->drawPrivate;

    /*Call our z-buffered pixel drawing function with xyz and argb, and
      also pass it the current zfunction, which is stored in the private draw
      context data structure. Note that this isn't a complete implementation!
      (We should be using kQATag_Width, for example.)*/

    MyDrawPixelWithZ(v->x, v->y, v->z, v->a, v->r, v->g, v->b,
                                    myData->stateVariable[kQATag_ZFunction]);
}
```

Note
See "Public Draw Context Methods," beginning on page 1-115 for complete information on the public draw context methods your drawing engine must define. ◆

Once you've defined the necessary public draw context methods, you need to insert pointers to those methods into a draw context structure. You accomplish this step in your `TQADrawPrivateNew` method, described in the next section.

Writing Private Draw Context Methods

Once you've written the public draw context methods supported by your drawing engine, you need to write several private draw context methods. In particular, you need to write a `TQADrawPrivateNew` method to initialize a draw context and a `TQADrawPrivateDelete` method to delete a draw context. The `TQADrawPrivateNew` method is called whenever an application creates a new draw context by calling the `QADrawContextNew` function. Listing 1-8 illustrates a sample `TQADrawPrivateNew` method.

Listing 1-8 A `TQADrawPrivateNew` method

```
TQAError MyDrawPrivateNew (
    TQADrawContext      *drawContext,
    const TQADevice     *device,
    const TQARect       *rect,
    const TQAClip       *clip,
    unsigned long       flags)
{
    MyPrivateData       *myData;

    /*Allocate a new MyPrivateData structure and store it in draw context.*/
    myData = MyDataNew(...);
    drawContext->drawPrivate = (TQADrawPrivate *) myData;
    if (!myData)
        return (kQAOutOfMemory);

    /*Set the method pointers of drawContext to point to our draw methods.*/
    newDrawContext->setFloat = MySetFloat;
    newDrawContext->setInt = MySetInt;
```

```
    ...
    return(kQANoErr);
}
```

As you can see, the `MyDrawPrivateNew` function defined in Listing 1-8 allocates space for its private data, installs a pointer to that data in the `drawPrivate` field of the draw context structure, and then installs pointers to all the public draw context methods supported by the drawing engine into the draw context structure.

Your `TQADrawPrivateDelete` method should simply undo any work done by your `TQADrawPrivateNew` method. In this case, the delete method just needs to release the private storage allocated by the `TQADrawPrivateNew` method. Listing 1-9 shows a sample `TQADrawPrivateDelete` method.

Listing 1-9 A `TQADrawPrivateDelete` method

```
void MyDrawPrivateDelete (TQADrawPrivate *drawPrivate)
{
    MyDataDelete((MyPrivateData *) drawPrivate);
}
```

You register your private draw context methods with QuickDraw 3D RAVE using another private method, the `TQAEngineGetMethod` method. See “Registering a Drawing Engine,” beginning on page 1-29 for details.

Handling Gestalt Selectors

To support calls to the public function `QAEngineGestalt`, your drawing engine must define a `TQAEngineGestalt` method. This method returns information about the capabilities of your drawing engine. For example, suppose that your drawing engine supports texture mapping and accelerates both Gouraud shading and line drawing. Suppose further that you have been assigned a vendor ID of 5, and that the engine ID of your engine is 1001. In that case, you could define a method like the one shown in Listing 1-10.

Listing 1-10 A `TQAEngineGestalt` method

```
TQAError MyEngineGestalt (TQAGestaltSelector selector, void *response)
{
    const static char       *myEngineName = "SurfDraw 3D";

    switch (selector) {
        case kQAGestalt_OptionalFeatures:
            *((unsigned long *) response) = kQAOptional_Texture;
            break;
        case kQAGestalt_FastFeatures:
            *((unsigned long *) response) = kQAFast_Line | kQAFast_Gouraud;
            break;
        case kQAGestalt_VendorID:
            *((long *) response) = 5;
            break;
        case kQAGestalt_EngineID:
            *((long *) response) = 1001;
            break;
        case kQAGestalt_Revision:
            *((long *) response) = 0;
            break;
        case kQAGestalt_ASCIINameLength:
            *((long *) response) = strlen(myEngineName);
            break;
        case kQAGestalt_ASCIIName:
            strcpy(response, myEngineName);
            break;
        default:                        /*must flag unrecognized selectors*/
            return (kQAParamErr);
    }
    return (kQANoErr);
}
```

If two different drawing engines should return identical vendor and engine IDs, QuickDraw 3D RAVE chooses the one that returns the most recent revision number (that is, the value returned for the `kQAGestalt_Revision` selector). The larger number is considered newer.

You register your `TQAEngineGestalt` method with QuickDraw 3D RAVE using the `TQAEngineGetMethod` method, described in the next section.

Registering a Drawing Engine

Once you written all the necessary public and private draw context methods, as well as methods to handle textures and bitmaps, you must write a `TQAEngineGetMethod` method that reports the addresses of some of those methods to QuickDraw 3D RAVE. Listing 1-11 shows a sample `TQAEngineGetMethod` method. Notice that this method returns the addresses only of the private draw context methods and the methods to handle textures and bitmaps. The pointers for the public draw context methods are assigned directly to the fields of a draw context structure by your `TQADrawPrivateNew` method (as shown in Listing 1-8).

Listing 1-11 A `TQAEngineGetMethod` method

```
TQAError MyEngineGetMethod (TQAEngineMethodTag methodTag, TQAEngineMethod *method)
{
    switch (methodTag) {
        case kQADrawPrivateNew:
            method->drawPrivateNew = MyDrawPrivateNew;
            break;
        case kQADrawPrivateDelete:
            method->drawPrivateDelete = MyDrawPrivateDelete;
            break;
        case kQAEngineCheckDevice:
            method->engineCheckDevice = MyEngineCheckDevice;
            break;
        case kQAEngineGestalt:
            method->engineGestalt = MyEngineGestalt;
            break;
        case kQABitmapNew:
            method->bitmapNew = MyBitmapNew;
            break;
        case kQABitmapDetach:
            method->bitmapDetach = MyBitmapDetach;
            break;
        case kQABitmapDelete:
            method->bitmapDelete = MyBitmapDelete;
            break;
        default:
            return(kQANotSupported);
```

```
    }
    return(kQANoErr);
}
```

Finally, you register your drawing engine by passing the address of your `TQAEngineGetMethod` method to the `QARegisterEngine` function:

```
QARegisterEngine(&MyEngineGetMethod);
```

You can call `QARegisterEngine` in two ways. During product development, you can link your drawing engine code directly with a test application, in which case you should call `QARegisterEngine` from your application's initialization code. Alternatively, once you've completed development, you should build your engine's code into a shared library of type `'tnsl'`. In this case, you should call `QARegisterEngine` from the initialization routine of the shared library. When the shared library containing QuickDraw 3D RAVE is loaded, it searches for and loads any drawing engines contained in shared libraries in the current folder or in the Extensions folder.

Supporting OpenGL Hardware

This section contains information that is useful if you are implementing a drawing engine to support hardware that is based on an OpenGL rasterization model. It describes special considerations for handling transparency and texture mapping.

Transparency

QuickDraw 3D RAVE supports three transparency models: the premultiplied, interpolated, and OpenGL transparency models. Support for the OpenGL transparency model (indicated by the `kQABlend_OpenGL` constant) should be automatic for hardware that is based on the OpenGL rasterization model. The other two models, indicated by the `kQABlend_PreMultiply` and `kQABlend_Interpolate` constants) may require emulation by your drawing engine.

For example, consider the premultiplied blending function, specified by these equations:

$$a = 1 - ((1 - a_s) \times (1 - a_d))$$
$$r = r_s + ((1 - a_s) \times r_d)$$
$$g = g_s + ((1 - a_s) \times g_d)$$
$$b = b_s + ((1 - a_s) \times b_d)$$

(Here, the factors a_s, r_s, g_s, and b_s represent the alpha, red, green and blue components of a source pixel; the factors a_d, r_d, g_d, and b_d represent the alpha, red, green and blue components of a destination pixel.)

Note
A complete description of how transparent objects are blended together with each of these models is provided in "Blending Operations" (page 1-49). ◆

OpenGL directly supports the premultiplied transparency blending function (and the interpolated transparency blending function) for the RGB components only. In other words, the alpha channel component (which is the same for both blending operations) cannot be directly implemented in OpenGL-compliant hardware. It is possible, however, to emulate these two transparency modes on OpenGL hardware, using several different methods. You can blend the RGB values only, or you can blend the ARGB values using a multipass algorithm. Which of these emulations you use depends on whether your drawing engine is associated with a frame buffer that stores an alpha channel or not.

If your drawing engine is associated with a frame buffer that doesn't store an alpha channel value, you can implement the premultiplied and interpolated blending functions by simply ignoring the alpha channel component. These functions are then equivalent to OpenGL blending modes. The premulitplied blending function, with its alpha channel ignored, can be emulated by this function:

```
gBlendFunc(GL_ONE, GL_ONE_MINUS_SRC_ALPHA);
```

Similarly, the interpolated blending function, with its alpha channel ignored, can be emulated by this function:

```
gBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

IMPORTANT
A drawing engine that uses this method of emulating the QuickDraw 3D RAVE blending functions on OpenGL hardware should not set the `kQAOptional_BlendAlpha` flag of the `kQAGestalt_OptionalFeatures` selector to the `QAEngineGestalt` function. ▲

To achieve a more complete blending, you can have your drawing engine rasterize each transparent object more than once, altering in each pass the blending mode, object alpha channel, and buffer write masks. The first pass should perform RGB blending. Accordingly, you should disable writing any alpha channel or z buffer data during this pass.

```
/*first pass*/
glColorMask(TRUE, TRUE, TRUE, FALSE);          /*disable alpha channel*/
glDepthMask(FALSE);                            /*disable Z buffer*/
if (premultpliedTransparency)
    glBlendFunc(GL_ONE, GL_ONE_MINUS_SRC_ALPHA);
else
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
/*render the object here*/
```

On the second pass, you should set the frame buffer alpha channel value to $(1–a_s)\times(1–a_d)$. To do this, you need to render the object again, with a different alpha value, as follows:

```
/*second pass*/
glColorMask(FALSE, FALSE, FALSE, TRUE);        /*enable alpha channel*/
glDepthMask(FALSE);                            /*disable Z buffer*/
glBlendFunc(GL_ONE_MINUS_DST_ALPHA, GL_ZERO);
/*render the object with alpha replaced with 1-a*/
```

Finally, the third pass should replace the value in the alpha channel with the final value $1–((1–a_s)\times(1–a_d))$. To do this, you need to render the object again, with its alpha value set to 1, as follows:

```
/*third pass*/
glColorMask(FALSE, FALSE, FALSE, TRUE);        /*enable alpha channel*/
glDepthMask(TRUE);                             /*enable Z buffer*/
glBlendFunc(GL_ONE_MINUS_DST_ALPHA, GL_ZERO);
/*render the object with alpha replaced with 1*/
```

After the third pass, the frame buffer contains the correctly blended object.

Texture Mapping

QuickDraw 3D RAVE supports several texture mapping operations, which are controlled by the flags in the `kQATag_TextureOp` state variable. Currently these flags are defined:

```
#define kQATextureOp_Modulate                   (1 << 0)
#define kQATextureOp_Highlight                  (1 << 1)
#define kQATextureOp_Decal                      (1 << 2)
#define kQATextureOp_Shrink                     (1 << 3)
```

Note
A complete description of texture mapping operations is provided in “Texture Operations” (page 1-51). ◆

To support the `kQATextureOp_Modulate` mode on an OpenGL-compliant rasterizer, you can use the `GL_MODULATE` mode, where the `kd_r`, `kd_g`, and `kd_b` fields of a texture vertex specify the modulating color. Note, however, that `GL_MODULATE` does not allow these color values to be greater than 1.0, whereas QuickDraw 3D RAVE does allow them to be greater than 1.0. Values greater than 1.0 can provide improved image realism, and new hardware should support them. A more reasonable maximum modulation amplitude is 2.0.

You can support the `kQATextureOp_Highlight` mode by performing two rendering passes. The first pass should render the texture-mapped object (possibly also with modulation, as just described), and the second pass should add the specular highlight value.

```
/*first pass*/
glDepthMask(FALSE);                             /*disable Z buffer*/
/*render the texture-mapped object here*/

/*second pass*/
glDepthMask(TRUE);                              /*enable Z buffer*/
glBlendFunc(GL_ONE, GL_ONE);                    /*add highlight color*/
/*render the highlight color as a Gouraud-shaded object here*/
```

On the second pass, you should render the highlight color, using the `ks_r`, `ks_g`, and `ks_b` fields of a texture vertex, as a Gouraud-shaded object.

If the kQATextureOp_Modulate flag is clear (that is, is no texture map color modulation is to be performed), you can support the kQATextureOp_Decal mode using the OpenGL GL_DECAL mode. If, in addition, the kQATextureOp_Highlight flag is set, you need to perform two rendering passes, as just described.

IMPORTANT
There is currently no known method of accurately rendering to OpenGL-compliant hardware when *both* the kQATextureOp_Decal and the kQATextureOp_Modulate flags are set. You should determine the best method of implementing this mode correctly on your hardware. If your hardware cannot handle both modes at once, you should ignore the kQATextureOp_Modulate mode whenever kQATextureOp_Decal is set. ▲

QuickDraw 3D RAVE Reference

This section describes the constants, data structures, and routines provided by QuickDraw 3D RAVE. It also describes the functions you must define in order to write a drawing engine.

The application programming interfaces of QuickDraw 3D RAVE follow these simple naming conventions:

- All names of constants begin with the prefix kQA (for example, kQATextureFilter_Fast).
- All names of data types begin with the prefix TQA (for example, TQADrawContext).
- All names of functions begin with the prefix QA (for example, QADrawContextNew).

Constants

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

Version Values

The `version` field of a draw context structure (of type `TQADrawContext`) specifies the current version of QuickDraw 3D RAVE. This field contains one of these constants:

```
typedef enum TQAVersion {
    kQAVersion_Prerelease               = 0,
    kQAVersion_1_0                      = 1,
    kQAVersion_1_0_5                    = 2,
    kQAVersion_1_1                      = 3
} TQAVersion;
```

Constant descriptions

`kQAVersion_Prerelease`
A prerelease version.

`kQAVersion_1_0` Version 1.0. This is the version that supports the interactive renderer included with QuickDraw 3D version 1.0.

`kQAVersion_1_0_5` Version 1.0.5. This version supports triangle meshes and color lookup tables.

`kQAVersion_1_1` Version 1.1. This version supports notice methods, texture compression flags, and the `kQAGestalt_AvailableTexMem` selector for the `QAEngineGestalt` function.

Pixel Types

The `pixelType` field of a memory device structure (of type `TQADeviceMemory`) specifies a pixel format (that is, the size and organization of the memory associated with a single pixel in a memory pixmap). You use these constants to assign a value to that field and also to parameters to the `QATextureNew` and `QABitmapNew` functions.

```
typedef enum TQAImagePixelType {
    kQAPixel_Alpha1                     = 0,
    kQAPixel_RGB16                      = 1,
    kQAPixel_ARGB16                     = 2,
    kQAPixel_RGB32                      = 3,
    kQAPixel_ARGB32                     = 4,
```

```
    kQAPixel_CL4                        = 5,
    kQAPixel_CL8                        = 6
} TQAImagePixelType;
```

Constant descriptions

`kQAPixel_Alpha1` A pixel occupies 1 bit of memory, which is interpreted as an alpha channel value. This value is relevant only for the `QABitmapNew` function. When a bit is 1, it is opaque and is rendered in the color passed to the `QADrawBitmap` function; when the bit is 0, it is completely transparent.

`kQAPixel_RGB16` A pixel occupies 16 bits of memory, with the red component in bits 14 through 10, the green component in bits 9 through 5, and the blue component in bits 4 through 0. There is no per-pixel alpha channel value. As a result, the pixmap (perhaps defining a texture) is treated as opaque. (You can, however, apply transparency to the pixmap using the alpha channel values of a triangle vertex, for instance.)

`kQAPixel_ARGB16` A pixel occupies 16 bits of memory, with the red component in bits 14 through 10, the green component in bits 9 through 5, and the blue component in bits 4 through 0. In addition, the pixel's alpha channel value is in bit 15. When the alpha value is 1, the pixmap is opaque; when the alpha value is 0, the pixmap is completely transparent.

`kQAPixel_RGB32` A pixel occupies 32 bits of memory, with the red component in bits 23 through 16, the green component in bits 15 through 8, and the blue component in bits 7 through 0. There is no per-pixel alpha channel value. As a result, the pixmap (perhaps defining a texture) is treated as opaque. (You can, however, apply transparency to the pixmap using the alpha channel values of a triangle vertex, for instance.)

`kQAPixel_ARGB32` A pixel occupies 32 bits of memory, with the red component in bits 23 through 16, the green component in bits 15 through 8, and the blue component in bits 7 through 0. In addition, the pixel's alpha channel value is in bits 31 through 24. When the alpha value is 255, the pixmap is opaque; when the alpha value is 0, the pixmap is completely transparent.

`kQAPixel_CL4` A pixel value is an index into a 4-bit color lookup table. This color lookup table is always big-endian (that is, the high 4 bits affect the leftmost pixel). This pixel type is valid only as a parameter for the `QATextureNew` and `QABitmapNew` functions. Not all drawing engines support this pixel type; it is supported only when a drawing engine supports the `kQAOptional_CL4` feature.

`kQAPixel_CL8` A pixel value is an index into a 8-bit color lookup table. This pixel type is valid only as a parameter for the `QATextureNew` and `QABitmapNew` functions. Not all drawing engines support this pixel type; it is supported only when a drawing engine supports the `kQAOptional_CL8` feature.

Color Lookup Table Types

The `tableType` parameter of the `QAColorTableNew` function specifies a color lookup table type. QuickDraw 3D RAVE currently supports these types of color lookup tables:

```
typedef enum TQAColorTableType {
    kQAColorTable_CL8_RGB32             = 0,
    kQAColorTable_CL4_RGB32             = 1
} TQAColorTableType;
```

Constant descriptions

`kQAColorTable_CL8_RGB32`
The color lookup table contains 256 colors, and each color occupies 32 bits of memory, with the red component in bits 23 through 16, the green component in bits 15 through 8, and the blue component in bits 7 through 0.

`kQAColorTable_CL4_RGB32`
The color lookup table contains 16 colors, and each color occupies 32 bits of memory, with the red component in bits 23 through 16, the green component in bits 15 through 8, and the blue component in bits 7 through 0.

Device Types

The `deviceType` field of a device data structure (of type `TQADevice`) specifies a device type. You use these constants to assign a value to that field.

```
typedef enum TQADeviceType {
    kQADeviceMemory                     = 0,
    kQADeviceGDevice                    = 1,
    kQADeviceWin32DC                    = 2,
    kQADeviceDDSurface                  = 3
} TQADeviceType;
```

Constant descriptions

`kQADeviceMemory` A memory device.

`kQADeviceGDevice` A graphics device (of type `GDevice`).

`kQADeviceWin32DC` A Windows 32 device.

`kQADeviceDDSurface` A Windows direct draw surface.

Clip Types

The `clipType` field of a clip data structure (of type `TQAClip`) specifies a clip type. You use these constants to assign a value to that field.

```
typedef enum TQAClipType {
    kQAClipRgn                          = 0,
    kQAClipWin32Rgn                     = 1
} TQAClipType;
```

Constant descriptions

`kQAClipRgn` A clipping region.

`kQAClipWin32Rgn` A Windows 32 clipping region.

Tags for State Variables

A drawing engine maintains a large number of state variables that determine how the engine draws into a device. Each state variable has a state value, which is either an unsigned long integer, a floating-point value, or a pointer. You can read and write state values by calling QuickDraw 3D RAVE functions. (For instance, you can set a state value by calling `QASetInt`, `QASetFloat`, or

QASetPtr.) You specify which state variable to get or set using a state tag, a unique identifier associated with that variable.

Note
All tag values greater than 0 and less than kQATag_EngineSpecific_Minimum are reserved for use by QuickDraw 3D RAVE. If you need to define engine-specific tags, you should assign them tag values greater than or equal to kQATag_EngineSpecific_Minimum. ◆

Here are the tags for state variables having unsigned long integer values:

```
typedef enum TQATagInt {
    kQATag_ZFunction                    = 0,    /*required variables*/
    kQATag_Antialias                    = 8,    /*optional variables*/
    kQATag_Blend                        = 9,
    kQATag_PerspectiveZ                 = 10,
    kQATag_TextureFilter                = 11,
    kQATag_TextureOp                    = 12,
    kQATag_CSGTag                       = 14,
    kQATag_CSGEquation                  = 15,
    kQATagGL_DrawBuffer                 = 100,  /*OpenGL variables*/
    kQATagGL_TextureWrapU               = 101,
    kQATagGL_TextureWrapV               = 102,
    kQATagGL_TextureMagFilter           = 103,
    kQATagGL_TextureMinFilter           = 104,
    kQATagGL_ScissorXMin                = 105,
    kQATagGL_ScissorYMin                = 106,
    kQATagGL_ScissorXMax                = 107,
    kQATagGL_ScissorYMax                = 108,
    kQATagGL_BlendSrc                   = 109,
    kQATagGL_BlendDst                   = 110,
    kQATagGL_LinePattern                = 111,
    kQATagGL_AreaPattern0               = 117,
    kQATagGL_AreaPattern31              = 148,
    kQATag_EngineSpecific_Minimum       = 1000
} TQATagInt;
```

Constant descriptions

kQATag_ZFunction — The z sorting function of the drawing engine. This function determines which surfaces are to be removed during

hidden surface removal. See "Z Sorting Function Selectors" (page 1-47) for a description of the available z sorting functions. The default value for a drawing engine that is z buffered is `kQAZFunction_LT`; the default value for a draw context that is not z buffered is `kQAZFunction_None`. The z sorting function state variable must be supported by all drawing engines.

`kQATag_Antialias` The antialiasing mode of the drawing engine. This mode determines how, if at all, antialiasing is applied to the draw context. See "Antialiasing Selectors" (page 1-48) for a description of the available antialiasing modes. The default value for a drawing engine that supports antialiasing is `kQAAntiAlias_Fast`. The antialiasing mode state variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_Antialias` feature.

`kQATag_Blend` The transparency blending function of the drawing engine. See "Blending Operations" (page 1-49) for a description of the available transparency blending functions. The default value for a drawing engine that supports blending is `kQABlend_Premultiply`. The transparency blending function state variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_Blend` feature.

`kQATag_PerspectiveZ`

The z perspective control of the drawing engine. This control determines how a drawing engine performs hidden surface removal. See "Z Perspective Selectors" (page 1-50) for a description of the available z perspective controls. The default value for a drawing engine that supports z perspective is `kQAPerspectiveZ_Off`. The z perspective control state variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_PerspectiveZ` feature.

`kQATag_TextureFilter`

The texture mapping filter mode of the drawing engine. This mode determines how a drawing engine performs texture mapping. See "Texture Filter Selectors" (page 1-51) for a description of the available texture mapping filter modes. The default value for a drawing engine that supports texture mapping is `kQATextureFilter_Fast`. The texture mapping filter state variable is optional; it must be

supported only when a drawing engine supports the `kQAOptional_Texture` feature.

`kQATag_TextureOp` The texture mapping operation of the drawing engine. This mode determines the current texture mapping operation of a drawing engine. See "Texture Operations" (page 1-51) for a description of the available texture mapping operations. The default value for a drawing engine that supports texture mapping is `kQATextureOp_None`. The texture mapping operation variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_Texture` feature.

`kQATag_CSGTag` The CSG ID of triangles subsequently submitted to the drawing engine. See "CSG IDs" (page 1-53) for a description of the available CSG IDs. The default value for a drawing engine that supports CSG operations is `kQACSGTag_None`. The CSG ID variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_CSG` feature.

`kQATag_CSGEquation` The CSG equation for the drawing engine, which determines the manner in which triangles with CSG IDs are combined into solid objects. See the book *3D Graphics Programming With QuickDraw 3D* for an explanation of how to specify a CSG equation. The CSG equation variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_CSG` feature.

`kQATagGL_DrawBuffer`

The OpenGL color buffer of the drawing engine. This determines where a drawing engine draws when writing colors to a frame buffer. See "Buffer Drawing Operations" (page 1-55) for a description of the buffer drawing modes. The default value of this variable for a drawing engine that supports OpenGL buffering is `kQAGL_DrawBuffer_Front` for single-buffered contexts and `kQAGL_DrawBuffer_Back` for double-buffered contexts. The OpenGL color buffer state variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_OpenGL` feature.

`kQATagGL_TextureWrapU`

The OpenGL texture *u* wrapping mode of the drawing engine. See "Texture Wrapping Values" (page 1-54) for a description of the wrapping modes. The default value of

this variable for a drawing engine that supports OpenGL texture wrapping is `kQAGL_Repeat`. The OpenGL texture *u* wrapping mode state variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_OpenGL` feature.

`kQATagGL_TextureWrapV`

The OpenGL texture *v* wrapping mode of the drawing engine. See "Texture Wrapping Values" (page 1-54) for a description of the wrapping modes. The default value of this variable for a drawing engine that supports OpenGL texture wrapping is `kQAGL_Repeat`. The OpenGL texture *v* wrapping mode state variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_OpenGL` feature.

`kQATagGL_TextureMagFilter`

The OpenGL **texture magnification function** of the drawing engine. This function is called when a pixel being textured maps to an area that is less than or equal to one texture element. See **[to be supplied]** for a description of the available magnification functions. The default value of this variable for a drawing engine that supports OpenGL texture magnification is `kQAGL_Linear`. The OpenGL texture magnification function state variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_OpenGL` feature.

`kQATagGL_TextureMinFilter`

The OpenGL **texture minifying function** of the drawing engine. This function is called when a pixel being textured maps to an area that is greater than one texture element. See **[to be supplied]** for a description of the available minifying functions. The default value of this variable for a drawing engine that supports OpenGL texture minifying is `kQAGL_ToBeSupplied`. The OpenGL texture minifying function state variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_OpenGL` feature.

`kQATagGL_ScissorXMin`

The minimum *x* value of the **scissor box,** a rectangle that determines which pixels can be modified by drawing commands. This state variable is optional; it must be

supported only when a drawing engine supports the `kQAOptional_OpenGL` feature.

`kQATagGL_ScissorYMin`
: The minimum *y* value of the scissor box. This state variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_OpenGL` feature.

`kQATagGL_ScissorXMax`
: The maximum *x* value of the scissor box. This state variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_OpenGL` feature.

`kQATagGL_ScissorYMax`
: The maximum *y* value of the scissor box. This state variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_OpenGL` feature.

`kQATagGL_BlendSrc` The source blending operation of the drawing engine. This control determines how a drawing engine computes the red, green, blue, and alpha source-blending factors when performing transparency blending. The source blending operation state variable is optional; it must be supported only when a drawing engine supports both the `kQAOptional_Blend` and `kQAOptional_OpenGL` features.

`kQATagGL_BlendDst` The destination blending operation of the drawing engine. This control determines how a drawing engine computes the red, green, blue, and alpha destination-blending factors when performing transparency blending. The destination blending operation state variable is optional; it must be supported only when a drawing engine supports both the `kQAOptional_Blend` and `kQAOptional_OpenGL` features.

`kQATagGL_LinePattern`
: The OpenGL **line stipple pattern** of the drawing engine. This pattern specifies which bits in a line are to be drawn and which are masked out.

`kQATagGL_AreaPattern0`
: The first of 32 registers that specify an **area stipple pattern.**

`kQATagGL_AreaPattern31`
: The last of 32 area stipple pattern registers.

`kQATag_EngineSpecific_Minimum`
: The minimum tag value to be used for variables that are specific to a particular drawing engine. Any custom

variables you support must have tag values greater than or equal to this value. Note that you should use engine-specific tags only in exceptional circumstances, because the operations determined by the associated state variables are not generally accessible.

Here are the tags for state variables having floating-point values:

```
typedef enum TQATagFloat {
    kQATag_ColorBG_a                    = 1,    /*required variables*/
    kQATag_ColorBG_r                    = 2,
    kQATag_ColorBG_g                    = 3,
    kQATag_ColorBG_b                    = 4,
    kQATag_Width                        = 5,
    kQATag_ZMinOffset                   = 6,
    kQATag_ZMinScale                    = 7,
    kQATagGL_DepthBG                    = 112,  /*OpenGL variables*/
    kQATagGL_TextureBorder_a            = 113,
    kQATagGL_TextureBorder_r            = 114,
    kQATagGL_TextureBorder_g            = 115,
    kQATagGL_TextureBorder_b            = 116
} TQATagFloat;
```

Constant descriptions

`kQATag_ColorBG_a` The alpha channel value of a drawing engine's background color. This value must be greater than or equal to 0.0 and less than or equal to 1.0. The default value for the background color alpha channel is 0.0. The background color alpha channel state variable must be supported by all drawing engines.

`kQATag_ColorBG_r` The red component of a drawing engine's background color. This value must be greater than or equal to 0.0 and less than or equal to 1.0. The default value for the background color red component is 0.0. The background color red component state variable must be supported by all drawing engines.

`kQATag_ColorBG_g` The green component of a drawing engine's background color. This value must be greater than or equal to 0.0 and less than or equal to 1.0. The default value for the background color green component is 0.0. The background

color green component state variable must be supported by all drawing engines.

`kQATag_ColorBG_b` The blue component of a drawing engine's background color. This value must be greater than or equal to 0.0 and less than or equal to 1.0. The default value for the background color blue component is 0.0. The background color blue component state variable must be supported by all drawing engines.

`kQATag_Width` The width (in pixels) of points or lines drawn by the drawing engine. This value must be greater than or equal to 0.0 and less than or equal to `kQAMaxWidth` (currently defined as 128.0). The default value for the width is 1.0. The width state variable must be supported by all drawing engines.

`kQATag_ZMinOffset` The minimum *z* offset that must be performed to guarantee that a drawn object passes the `kQAZFunction_LT` hidden surface test. This variable is read-only; you cannot set its value. In general, a drawing engine that employs fixed-point values for the *z* coordinate returns a small negative value (for example, 1/65536) for the minimum offset; a drawing engine that employs floating-point values for the *z* coordinate returns 0.0 for the minimum offset.

`kQATag_ZMinScale` The minimum *z* scale factor that must be applied to guarantee that a drawn object passes the `kQAZFunction_LT` hidden surface test. This variable is read-only; you cannot set its value. In general, a drawing engine that employs fixed-point values for the *z* coordinate returns 1.0 for the minimum scale factor; a drawing engine that employs floating-point values for the *z* coordinate returns a value slightly less than 1.0 (for example, 0.9999) for the minimum scale factor.

`kQATagGL_DepthBG` The OpenGL background z of the drawing engine. The default value of this variable for a drawing engine that supports OpenGL texture magnification is `kQAGL_Linear`. The OpenGL background z state variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_OpenGL` feature.

`kQATagGL_TextureBorder_a`
The alpha component of a drawing engine's texture border

color. This value must be greater than or equal to 0.0 and less than or equal to 1.0. The default value for the texture border color alpha component is 0.0. The texture border color alpha component state variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_OpenGL` feature.

`kQATagGL_TextureBorder_r`
The red component of a drawing engine's texture border color. This value must be greater than or equal to 0.0 and less than or equal to 1.0. The default value for the texture border color red component is 0.0. The texture border color red component state variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_OpenGL` feature.

`kQATagGL_TextureBorder_g`
The green component of a drawing engine's texture border color. This value must be greater than or equal to 0.0 and less than or equal to 1.0. The default value for the texture border color green component is 0.0. The texture border color green component state variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_OpenGL` feature.

`kQATagGL_TextureBorder_b`
The blue component of a drawing engine's texture border color. This value must be greater than or equal to 0.0 and less than or equal to 1.0. The default value for the texture border color blue component is 0.0. The texture border color blue component state variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_OpenGL` feature.

Here are the tags for state variables having pointer values:

```
typedef enum TQATagPtr {
    kQATag_Texture                      = 13
} TQATagPtr;
```

Constant descriptions

`kQATag_Texture` A pointer to the current texture map of the drawing engine, as created by the `QATextureNew` function. The default value for the texture map pointer is `NULL`. The

texture map pointer state variable is optional; it must be supported only when a drawing engine supports the `kQAOptional_Texture` feature.

Z Sorting Function Selectors

A drawing engine must support a **z sorting function** that determines which surfaces are to be removed during hidden surface removal. You specify an engine's z sorting function by assigning a value to its `kQATag_ZFunction` state variable. The default value of this variable for a drawing engine that is z buffered is `kQAZFunction_LT`; the default value (and also the only possible value) for a draw context that is not z buffered is `kQAZFunction_None`.

IMPORTANT

If a drawing engine supports `kQAOptional_PerspectiveZ` and if the state variable `kQATag_PerspectiveZ` is set to the value `kQAPerspectiveZ_On`, then the state variable `kQATag_ZFunction` should be interpreted so that it yields the same result as when the value of `kQATag_PerspectiveZ` is `kQAPerspectiveZ_Off`. ▲

```
#define kQAZFunction_None                       0
#define kQAZFunction_LT                         1
#define kQAZFunction_EQ                         2
#define kQAZFunction_LE                         3
#define kQAZFunction_GT                         4
#define kQAZFunction_NE                         5
#define kQAZFunction_GE                         6
#define kQAZFunction_True                       7
```

Constant descriptions

`kQAZFunction_None` The z value is neither tested nor written.

`kQAZFunction_LT` A new z value is visible if it is less than the value in the z buffer.

`kQAZFunction_EQ` A new z value is visible if it is equal to the value in the z buffer. This selector should be passed only to drawing engines that support the optional OpenGL features.

`kQAZFunction_LE` A new z value is visible if it is less than or equal to the value in the z buffer. This selector should be passed only to

	drawing engines that support the optional OpenGL features.
kQAZFunction_GT	A new z value is visible if it is greater than the value in the z buffer. This selector should be passed only to drawing engines that support the optional OpenGL features.
kQAZFunction_NE	A new z value is visible if it is not equal to the value in the z buffer. This selector should be passed only to drawing engines that support the optional OpenGL features.
kQAZFunction_GE	A new z value is visible if it is greater than or equal to the value in the z buffer. This selector should be passed only to drawing engines that support the optional OpenGL features.
kQAZFunction_True	A new z value is always visible.

Antialiasing Selectors

You specify an engine's antialiasing mode by assigning a value to its kQATag_Antialias state variable. The default value of this variable for a drawing engine that supports antialiasing is kQAAntiAlias_Fast.

```
#define kQAAntiAlias_Off                    0
#define kQAAntiAlias_Fast                   1
#define kQAAntiAlias_Mid                    2
#define kQAAntiAlias_Best                   3
```

Constant descriptions

kQAAntiAlias_Off	Antialiasing is off.
kQAAntiAlias_Fast	The drawing engine performs whatever level of antialiasing it can do with no speed penalty. This often means that antialiasing is turned off.
kQAAntiAlias_Mid	The drawing engine performs a medium level of antialiasing. You should use this antialiasing mode when you want to perform antialiasing interactively.
kQAAntiAlias_Best	The drawing engine performs the highest level of antialiasing it can. This mode may be unsuitable for interactive rendering.

Blending Operations

A drawing engine may support a **transparency blending function** that determines the kind of transparency blending applied to a drawing context when combining new (“source”) pixels with the pixels already in a frame buffer (“destination”). You specify an engine’s transparency blending function by assigning a value to its `kQATag_Blend` state variable. The default value of this variable for a draw context that supports transparency blending is `kQABlend_PreMultiply`.

In the equations below, the factors a_s, r_s, g_s, and b_s represent the alpha, red, green and blue components of a source pixel; the factors a_d, r_d, g_d, and b_d represent the alpha, red, green and blue components of a destination pixel.

```
#define kQABlend_PreMultiply                    0
#define kQABlend_Interpolate                    1
#define kQABlend_OpenGL                         2
```

Constant descriptions

`kQABlend_PreMultiply`

The drawing engine uses a premultiplied blending function. The components of a pixel written to the frame buffer are computed using these equations:

$$a = 1 - ((1 - a_s) \times (1 - a_d))$$
$$r = r_s + ((1 - a_s) \times r_d)$$
$$g = g_s + ((1 - a_s) \times g_d)$$
$$b = b_s + ((1 - a_s) \times b_d)$$

In general, you should use the premultiplied blending function for rendering shaded transparent 3D primitives (such as triangles). The premultiplied function does not scale the source color components by the alpha value a_s; as a result, this function allows a transparent object to have a specular highlight value that is greater than its alpha channel value. For example, a sheet of glass might allow 99% of the light behind it to pass though (indicating an alpha channel value of 0.01). However, that same sheet of glass might have a specular highlight value much greater than 0.01. The premultiplied function allows the drawing engine to render this object correctly.

`kQABlend_Interpolate`

The drawing engine uses an interpolated blending function. The components of a pixel written to the frame buffer are computed using these equations:

$$a = 1 - ((1 - a_s) \times (1 - a_d))$$
$$r = (r_s \times a_s) + ((1 - a_s) \times r_d)$$
$$g = (g_s \times a_s) + ((1 - a_s) \times g_d)$$
$$b = (b_s \times a_s) + ((1 - a_s) \times b_d)$$

The interpolated blending function is not entirely suitable for rendering shaded transparent objects, but it is very effective for compositing bitmap images.

`kQABlend_OpenGL` The drawing engine uses the OpenGL blending function determined by the values of the `kQATagGL_BlendSrc` and `kQATagGL_BlendDest` state variables. For complete information on OpenGL blending functions, consult the description of the `glBlendFunc` function in *OpenGL™ Reference Manual*. OpenGL blending functions are supported only by drawing engines that support the `kQAOptional_OpenGL` feature.

Z Perspective Selectors

A drawing engine may support a **z perspective control** that determines whether the `z` or the `invW` field of a vertex (of type `TQAVGouraud` or `TQAVTexture`) is to be used for hidden surface removal. You specify an engine's z perspective control by assigning a value to its `kQATag_PerspectiveZ` state variable. The default value of this variable for a drawing engine that supports z perspective is `kQAPerspectiveZ_Off`.

```
#define kQAPerspectiveZ_Off                     0
#define kQAPerspectiveZ_On                      1
```

Constant descriptions

`kQAPerspectiveZ_Off`

The drawing engine performs hidden surface removal using `z` values, as is standard.

`kQAPerspectiveZ_On` The drawing engine performs hidden surface removal using `invW` values, which results in perspective-correct hidden surface removal.

Texture Filter Selectors

A drawing engine may support a **texture mapping filter mode** that determines how a drawing engine performs texture mapping. You specify an engine's texture filter by assigning a value to its `kQATag_TextureFilter` state variable. The default value of this variable for a drawing engine that supports texture mapping is `kQATextureFilter_Fast`.

```
#define kQATextureFilter_Fast                   0
#define kQATextureFilter_Mid                    1
#define kQATextureFilter_Best                   2
```

Constant descriptions

`kQATextureFilter_Fast`
The drawing engine performs whatever level of texture filtering it can do with no speed penalty. This often means that no texture filtering is performed.

`kQATextureFilter_Mid`
The drawing engine performs a medium level of texture filtering. You should use this texture mapping filter mode when you want to perform texture mapping interactively.

`kQATextureFilter_Best`
The drawing engine performs the highest level of texture filtering it can. This mode may be unsuitable for interactive rendering.

Texture Operations

A drawing engine may support a **texture mapping operation** that determines how a drawing engine performs texture mapping. You specify an engine's texture mapping operation by assigning a value to its `kQATag_TextureOp` state variable. The default value of this variable for a drawing engine that supports texture mapping is `kQATextureOp_None`.

You can use the following masks to specify a texture mapping operation. The bits are ORed together to determine the desired operation.

```
#define kQATextureOp_None                       0
#define kQATextureOp_Modulate                   (1 << 0)
#define kQATextureOp_Highlight                  (1 << 1)
#define kQATextureOp_Decal                      (1 << 2)
#define kQATextureOp_Shrink                     (1 << 3)
```

Constant descriptions

`kQATextureOp_None` The drawing engine supports no special texture mapping operations. The drawing engine simply replaces an object's color with the texture map color. This mode results in a flat-looking image with no lighting effects, which is most useful when the texture mapping engine is used as a 2D warping engine (for example, for video effects). The texture map's alpha channel values control the transparency of a rendered object on a per-pixel basis. The alpha channel value of a particular pixel is the product of texture map's alpha channel value and the vertex alpha channel value (which is interpolated from the `TQAVTexture` data structure).

`kQATextureOp_Modulate`
The texture map color is modulated with the interpolated diffuse colors (from the `kd_r`, `kd_g`, and `kd_b` fields of a texture vertex).

`kQATextureOp_Highlight`
The interpolated specular colors (from the `ks_r`, `ks_g`, and `ks_b` fields of a texture vertex) are added to the texture map color.

`kQATextureOp_Decal` The texture map alpha channel value is used to blend the texture map color and the interpolated decal colors (from the `r`, `g`, and `b` fields of a texture vertex). When the texture map alpha channel value is 0, the texture map color is replaced with the interpolated `r`, `g`, and `b` values.

`kQATextureOp_Shrink`
The drawing engine modifies any *u* and *v* values so that they always lie in the range 0.0 to 1.0 inclusive. This guarantees that wrapping not occur. In theory, *u* and *v* values in the range [0.0, 1.0] should never cause wrapping. In practice, however, errors that occur during *uv* interpolation can cause *uv* overflow or underflow, which

can result in occasional one pixel texture wrapping at the 0.0 and 1.0 boundaries.

IMPORTANT

The clamping specified by the `kQATextureOp_Shrink` flag is not the same type of OpenGL texture clamping specified by the `kQATagGL_TextureWrapU` and `kQATagGL_TextureWrapV` state variables (see "Texture Wrapping Values" (page 1-54)). OpenGL clamping is designed to accept arbitrary *uv* values, while clamping specified by the `kQATextureOp_Shrink` flag operates only on *uv* values in the range 0.0 to 1.0. The `kQATextureOp_Shrink` clamping is therefore less expensive to implement (perhaps simply by compressing the range of *u* and *v* slightly before beginning interpolation). Any drawing engine that does support OpenGL clamping can use that code to support `kQATextureOp_Shrink` clamping. ▲

CSG IDs

A drawing engine may support **CSG IDs** that determine what number a drawing engine assigns to triangles submitted for drawing. You specify an engine's CSG ID by assigning a value to its `kQATag_CSGTag` state variable. The default value of this variable for a drawing engine that supports CSG is `kQACSGTag_None`. You can use the following constants to specify a CSG ID.

```
#define kQACSGTag_None                  0xffffffffUL
#define kQACSGTag_0                     0
#define kQACSGTag_1                     1
#define kQACSGTag_2                     2
#define kQACSGTag_3                     3
#define kQACSGTag_4                     4
```

Constant descriptions

`kQACSGTag_None`	Do not assign CSG IDs to submitted triangles.
`kQACSGTag_0`	Submitted triangles have the CSG ID 0.
`kQACSGTag_1`	Submitted triangles have the CSG ID 1.
`kQACSGTag_2`	Submitted triangles have the CSG ID 2.
`kQACSGTag_3`	Submitted triangles have the CSG ID 3.

kQACSGTag_4 Submitted triangles have the CSG ID 4.

Texture Wrapping Values

A drawing engine may support OpenGL texture wrapping, in which case you might need to specify a texture wrapping mode in the *u* and *v* parametric directions. You specify an engine's texture wrapping modes by assigning a value to its kQATagGL_TextureWrapU and kQATagGL_TextureWrapV state variables. The default value of both these variables for a drawing engine that supports OpenGl texture wrapping is kQAGL_Repeat. You can use the following constants to specify a texture wrapping mode.

```
#define kQAGL_Repeat                    0
#define kQAGL_Clamp                     1
```

Constant descriptions

kQAGL_Repeat The integer part of a *u* or *v* coordinate is ignored, thereby causing a texture to be repeated across the surface of an object.

kQAGL_Clamp The *u* or *v* coordinates are clamped to the range [0, 1]. This mode prevents wrapping artifacts from occurring when a single texture is mapped onto an object.

Source Blending Values

When a drawing engine's transparency blending function is set to the value kQABlend_OpenGL, the state variable kQATagGL_BlendSrc must be set to a value to indicate the red, green, blue, and alpha source blending factors. You can use these constants to define the source blending factors.

```
#define kQAGL_SourceBlend_XXX           0
```

Constant descriptions

kQAGL_SourceBlend_XXX
[To be supplied.]

Destination Blending Values

When a drawing engine's transparency blending function is set to the value `kQABlend_OpenGL`, the state variable `kQATagGL_BlendDst` must be set to a value to indicate the red, green, blue, and alpha destination blending factors. You can use these constants to define the destination blending factors.

```
#define kQAGL_DestBlend_XXX                         0
```

Constant descriptions

`kQAGL_DestBlend_XXX`
[To be supplied.]

Buffer Drawing Operations

A drawing engine may support an OpenGL buffer drawing mode that determines which color buffers a drawing engine draws into. You specify one or more buffers by assigning a value to the `kQATagGL_DrawBuffer` state variable of that engine. The default value of this variable for a drawing engine that supports OpenGL buffering is `kQAGL_DrawBuffer_Front` for single-buffered contexts and `kQAGL_DrawBuffer_Back` for double-buffered contexts. You can use the following masks to specify a buffer drawing mode.

```
#define kQAGL_DrawBuffer_None                       0
#define kQAGL_DrawBuffer_FrontLeft                  (1 << 0)
#define kQAGL_DrawBuffer_FrontRight                 (1 << 1)
#define kQAGL_DrawBuffer_BackLeft                   (1 << 2)
#define kQAGL_DrawBuffer_BackRight                  (1 << 3)
#define kQAGL_DrawBuffer_Front                      \
        (kQAGL_DrawBuffer_FrontLeft | kQAGL_DrawBuffer_FrontRight)
#define kQAGL_DrawBuffer_Back                       \
        (kQAGL_DrawBuffer_BackLeft | kQAGL_DrawBuffer_BackRight)
```

Constant descriptions

`kQAGL_DrawBuffer_None`
The drawing engine draws into no color buffer.

`kQAGL_DrawBuffer_FrontLeft`
The drawing engine draws into the front left buffer only.

`kQAGL_DrawBuffer_FrontRight`
The drawing engine draws into the front right buffer only.

kQAGL_DrawBuffer_BackLeft
: The drawing engine draws into the back left buffer only.

kQAGL_DrawBuffer_BackRight
: The drawing engine draws into the back right buffer only.

kQAGL_DrawBuffer_Front
: The drawing engine draws into the front left and right buffers only.

kQAGL_DrawBuffer_Back
: The drawing engine draws into the back left and right buffers only.

Vertex Modes

The `vertexMode` parameter for the `QADrawVGouraud` and `QADrawVTexture` functions specifies a **vertex mode,** which determines how the drawing engine interprets and draws an array of vertices.

```
typedef enum TQAVertexMode {
    kQAVertexMode_Point                 = 0,
    kQAVertexMode_Line                  = 1,
    kQAVertexMode_Polyline              = 2,
    kQAVertexMode_Tri                   = 3,
    kQAVertexMode_Strip                 = 4,
    kQAVertexMode_Fan                   = 5
} TQAVertexMode;
```

Constant descriptions

kQAVertexMode_Point
: Draw points. Each vertex in the array is drawn as a point. The engine draws `nVertices` points (where `nVertices` is the number of vertices in the vertex array).

kQAVertexMode_Line
: Draw line segments. Each successive pair of vertices in the array determines a single line segment. The engine draws `nVertices`/2 line segments.

kQAVertexMode_Polyline
: Draw connected line segments. Each vertex in the array and its predecessor determine a line segment. The engine draws `nVertices`–1 line segments.

`kQAVertexMode_Tri` Draw triangles. Each successive triple of vertices in the array determines a single triangle. The engine draws `nVertices`/3 triangles.

`kQAVertexMode_Strip`
Draw a strip of triangles. The first three vertices in the array determine a triangle, and each successive vertex and its two predecessors determine a triangle that abuts the existing strip of triangles. The engine draws `nVertices`–2 triangles.

`kQAVertexMode_Fan` Draw a fan. The first three vertices in the array determine a triangle; each successive vertex, its immediate predecessor, and the first vertex in the array determine a triangle that abuts the existing fan. The engine draws `nVertices`–2 triangles.

Gestalt Selectors

You can use the `QAEngineGestalt` function to get information about a drawing engine. You pass `QAEngineGestalt` a selector that determines the kind of information about the engine you want to receive and a pointer to a buffer into which the information is to be copied. The selectors are defined by constants. Note that your application must allocate space for the buffer (pointed to by the `response` parameter) into which the information is copied.

```
typedef enum TQAGestaltSelector {
    kQAGestalt_OptionalFeatures         = 0,
    kQAGestalt_FastFeatures             = 1,
    kQAGestalt_VendorID                 = 2,
    kQAGestalt_EngineID                 = 3,
    kQAGestalt_Revision                 = 4,
    kQAGestalt_ASCIINameLength          = 5,
    kQAGestalt_ASCIIName                = 6,
    kQAGestalt_AvailableTexMem          = 7
} TQAGestaltSelector;
```

Constant descriptions

`kQAGestalt_OptionalFeatures`
`QAEngineGestalt` returns a value whose bits encode the optional features supported by the drawing engine. The `response` parameter must point to a buffer of type

unsigned long. See “Gestalt Optional Features Response Masks” (page 1-59) for a description of the meaning of the bits in the returned value.

kQAGestalt_FastFeatures

QAEngineGestalt returns a value whose bits encode the features supported by the drawing engine that are accelerated. The response parameter must point to a buffer of type unsigned long. See “Gestalt Fast Features Response Masks” (page 1-61) for a description of the meaning of the bits in the returned value.

kQAGestalt_VendorID

QAEngineGestalt returns the vendor ID of the drawing engine. The response parameter must point to a buffer of type long. See “Vendor and Engine IDs” (page 1-62) for a list of the currently defined vendor IDs.

kQAGestalt_EngineID

QAEngineGestalt returns the engine ID of the drawing engine. The response parameter must point to a buffer of type long. See “Vendor and Engine IDs” (page 1-62) for a list of the currently defined engine IDs.

kQAGestalt_Revision

QAEngineGestalt returns the revision number of the drawing engine. (Larger numbers indicate more recent revisions.) The response parameter must point to a buffer of type long.

kQAGestalt_ASCIINameLength

QAEngineGestalt returns the number of characters in the ASCII name of the drawing engine. The response parameter must point to a buffer of type long.

kQAGestalt_ASCIIName

QAEngineGestalt returns the ASCII name of the drawing engine. The response parameter must point to a C string whose length you have determined by passing the kQAGestalt_ASCIINameLength selector to QAEngineGestalt.

kQAGestalt_AvailableTexMem

QAEngineGestalt returns the size, in bytes, of the memory available for storing texture maps. Note that the amount of memory required to hold a particular texture map depends on the texture flags of that texture map—in particular, on

the texture compression and mipmapping flags. (See “Texture Flags Masks” (page 1-64) for details.) As a result, the size returned by `QAEngineGestalt` is only a rough indication of the number of texture maps that can be created. The `response` parameter must point to a buffer of type `Size`.

Gestalt Optional Features Response Masks

When you pass the `kQAGestalt_OptionalFeatures` selector to the `QAEngineGestalt` function, `QAEngineGestalt` returns (through its `response` parameter) a value that indicates the optional features supported by a drawing engine. You can use these masks to test that value for a specific feature. The bits corresponding to supported features are ORed together to determine the returned value.

Note
A drawing engine may support an optional feature in software only (that is, unaccelerated). You can use the `kQAGestalt_FastFeatures` selector to determine which, if any, features are accelerated by a drawing engine. ◆

```
#define kQAOptional_None                    0
#define kQAOptional_DeepZ                   (1 << 0)
#define kQAOptional_Texture                 (1 << 1)
#define kQAOptional_TextureHQ               (1 << 2)
#define kQAOptional_TextureColor            (1 << 3)
#define kQAOptional_Blend                   (1 << 4)
#define kQAOptional_BlendAlpha              (1 << 5)
#define kQAOptional_Antialias               (1 << 6)
#define kQAOptional_ZSorted                 (1 << 7)
#define kQAOptional_PerspectiveZ            (1 << 8)
#define kQAOptional_OpenGL                  (1 << 9)
#define kQAOptional_NoClear                 (1 << 10)
#define kQAOptional_CSG                     (1 << 11)
#define kQAOptional_BoundToDevice           (1 << 12)
#define kQAOptional_CL4                     (1 << 13)
#define kQAOptional_CL8                     (1 << 14)
```

Constant descriptions

`kQAOptional_None` This value is returned if the drawing engine supports no optional features.

`kQAOptional_DeepZ` This bit is set if the drawing engine supports **deep z buffering** (that is, z buffering with a resolution of greater than or equal to 24 bits per pixel).

`kQAOptional_Texture`
This bit is set if the drawing engine supports texture mapping.

`kQAOptional_TextureHQ`
This bit is set if the drawing engine supports **high-quality texture mapping** (that is, texture mapping using trilinear interpolation or an equivalent algorithm).

`kQAOptional_TextureColor`
This bit is set if the drawing engine supports full color texture modulation and highlighting.

`kQAOptional_Blend` This bit is set if the drawing engine supports transparency blending.

`kQAOptional_BlendAlpha`
This bit is set if the drawing engine supports transparency blending that uses an alpha channel.

`kQAOptional_Antialias`
This bit is set if the drawing engine supports antialiasing.

`kQAOptional_ZSorted`
This bit is set if the drawing engine supports z sorted rendering (for example, for transparency). If this bit is clear, an application must submit transparent objects for rendering in back-to-front z order (or the blending functions will not yield correct results). In general, an application should submit opaque objects first, followed by any transparent objects in back-to-front z order.

`kQAOptional_PerspectiveZ`
This bit is set if the drawing engine supports perspective-corrected hidden surface removal.

`kQAOptional_OpenGL` This bit is set if the drawing engine supports the extended OpenGL feature set.

`kQAOptional_NoClear`
This bit is set if the drawing engine doesn't clear the buffer

before drawing (so that double-buffering might not be required in some applications).

`kQAOptional_CSG` This bit is set if the drawing engine supports CSG.

`kQAOptional_BoundToDevice`
This bit is set if the drawing engine is tightly bound to a specific graphics device.

`kQAOptional_CL4` This bit is set if the drawing engine supports the `kQAPixel_CL4` pixel type.

`kQAOptional_CL8` This bit is set if the drawing engine supports the `kQAPixel_CL8` pixel type.

Gestalt Fast Features Response Masks

When you pass the `kQAGestalt_FastFeatures` selector to the `QAEngineGestalt` function, `QAEngineGestalt` returns (through its `response` parameter) a value that indicates which, if any, features supported by a drawing engine are accelerated. You can use these masks to test that value for a specific feature. The bits corresponding to accelerated features are ORed together to determine the returned value.

Note
A feature is considered accelerated if it is performed substantially faster by the drawing engine than it would be if performed in software only. ◆

```
#define kQAFast_None                    0
#define kQAFast_Line                    (1 << 0)
#define kQAFast_Gouraud                 (1 << 1)
#define kQAFast_Texture                 (1 << 2)
#define kQAFast_TextureHQ               (1 << 3)
#define kQAFast_Blend                   (1 << 4)
#define kQAFast_Antialiasing            (1 << 5)
#define kQAFast_ZSorted                 (1 << 6)
#define kQAFast_CL4                     (1 << 7)
#define kQAFast_CL8                     (1 << 8)
```

Constant descriptions

`kQAFast_None` This value is returned if the drawing engine accelerates no features.

kQAFast_Line	This bit is set if the drawing engine accelerates line drawing.
kQAFast_Gouraud	This bit is set if the drawing engine accelerates Gouraud shading.
kQAFast_Texture	This bit is set if the drawing engine accelerates texture mapping.
kQAFast_TextureHQ	This bit is set if the drawing engine accelerates high-quality texture mapping.
kQAFast_Blend	This bit is set if the drawing engine accelerates transparency blending.
kQAFast_Antialiasing	This bit is set if the drawing engine accelerates antialiasing.
kQAFast_ZSorted	This bit is set if the drawing engine accelerates z sorted rendering.
kQAFast_CL4	This bit is set if the drawing engine accelerates kQAPixel_CL4 pixel type rendering.
kQAFast_CL8	This bit is set if the drawing engine accelerates kQAPixel_CL8 pixel type rendering.

Vendor and Engine IDs

QuickDraw 3D RAVE defines constants for vendor IDs. You pass a vendor ID as a parameter to the QAEngineEnable and QAEngineDisable functions, and you receive a vendor ID when you pass the kQAGestalt_VendorID selector to the QAEngineGestalt function.

```
#define kQAVendor_BestChoice                    (-1)
#define kQAVendor_Apple                         0
#define kQAVendor_ATI                           1
#define kQAVendor_Radius                        2
#define kQAVendor_Mentor                        3
#define kQAVendor_Matrox                        4
#define kQAVendor_Yarc                          5
```

Constant descriptions

kQAVendor_BestChoice	The best drawing engine available for the target device. You should use this value as the default.
kQAVendor_Apple	The vendor is Apple Computer, Inc.

`kQAVendor_ATI` The vendor is ATI Technologies Inc.

`kQAVendor_Radius` The vendor is Radius.

`kQAVendor_Mentor` The vendor is Mentor Software, Inc.

`kQAVendor_Matrox` The vendor is Matrox.

`kQAVendor_Yarc` The vendor is YARC Systems.

For the vendor `kQAVendor_Apple`, QuickDraw 3D RAVE defines these constants for engine IDs.

```
#define kQAEngine_AppleSW                       0
#define kQAEngine_AppleHW                       (-1)
#define kQAEngine_AppleHW2                      1
```

Constant descriptions

`kQAEngine_AppleSW` The default software rasterizer.

`kQAEngine_AppleHW` The Apple 3D accelerator.

`kQAEngine_AppleHW2` Another Apple 3D accelerator.

Triangle Flags Masks

The `flags` parameter for the `QADrawTriGouraud` and `QADrawTriTexture` functions specifies a **triangle mode,** which determines how the drawing engine draws a triangle. You can use these masks to set the `flags` parameter.

```
#define kQATriFlags_None                        0
#define kQATriFlags_Backfacing                  (1 << 0)
```

Constant descriptions

`kQATriFlags_None` Pass this value for no triangle flags. The triangle is frontfacing or has an unspecified orientation.

`kQATriFlags_Backfacing`
The triangle is backfacing. You should set this bit for all triangles known to be backfacing (to help the drawing engine resolve ambiguous hidden surface removal situations).

Texture Flags Masks

The `flags` parameter for the `QATextureNew` function specifies a **texture mode,** which determines certain features of the new texture map. You can use these masks to set the `flags` parameter.

```
#define kQATexture_None                 0
#define kQATexture_Lock                 (1 << 0)
#define kQATexture_Mipmap               (1 << 1)
#define kQATexture_NoCompression        (1 << 2)
#define kQATexture_HighCompression      (1 << 3)
```

Constant descriptions

`kQATexture_None` Pass this value for no texture features.

`kQATexture_Lock` The new texture map should remain locked in memory and not be swapped out. You should set this flag for texture maps that are heavily used during rendering. Note, however, that this flag is usually ignored by software-based drawing engines.

`kQATexture_Mipmap` The new texture map is mipmapped.

`kQATexture_NoCompression`
The new texture map should not be compressed.

`kQATexture_HighCompression`
The new texture map should be compressed (even if doing so takes a considerable amount of time).

Bitmap Flags Masks

The `flags` parameter passed to the `QABitmapNew` function specifies a set of bit flags that control features of the new bitmap. You can use these masks to configure a `flags` parameter.

```
#define kQABitmap_None                  0
#define kQABitmap_Lock                  (1 << 1)
#define kQABitmap_NoCompression         (1 << 2)
#define kQABitmap_HighCompression       (1 << 3)
```

Constant descriptions

`kQABitmap_None` Pass this value for no bitmap features.

`kQABitmap_Lock` The new bitmap should remain locked in memory and not be swapped out. You should set this flag for bitmap that are heavily used during rendering. Note, however, that this flag is usually ignored by software-based drawing engines.

`kQABitmap_NoCompression`
The new bitmap should not be compressed.

`kQABitmap_HighCompression`
The new bitmap should be compressed (even if doing so takes a considerable amount of time).

Draw Context Flags Masks

The `flags` parameter passed to the `QADrawContextNew` function specifies a set of bit flags that control features of the new draw context. You can use these masks to configure a `flags` parameter.

```
#define kQAContext_None                 0
#define kQAContext_NoZBuffer            (1 << 0)
#define kQAContext_DeepZ                (1 << 1)
#define kQAContext_DoubleBuffer         (1 << 2)
#define kQAContext_Cache                (1 << 3)
```

Constant descriptions

`kQAContext_None` Pass this value for no draw context features.

`kQAContext_NoZBuffer`
The new draw context should not be z buffered.

`kQAContext_DeepZ` The new draw context should have a z buffer with at least 24 bits of precision.

`kQAContext_DoubleBuffer`
The new draw context should be double buffered.

`kQAContext_Cache` The new draw context should be used for a draw context cache.

Drawing Engine Method Selectors

To determine the addresses of some of the methods defined by a drawing engine, QuickDraw 3D RAVE calls the engine's `TQAEngineGetMethod` function,

passing a **method selector** in the `methodTag` parameter. This selector indicates of which method the engine should return the address in the `method` parameter.

```
typedef enum TQAEngineMethodTag {
    kQADrawPrivateNew                       = 0,
    kQADrawPrivateDelete                    = 1,
    kQAEngineCheckDevice                    = 2,
    kQAEngineGestalt                        = 3,
    kQATextureNew                           = 4,
    kQATextureDetach                        = 5,
    kQATextureDelete                        = 6,
    kQABitmapNew                            = 7,
    kQABitmapDetach                         = 8,
    kQABitmapDelete                         = 9,
    kQAColorTableNew                        = 10,
    kQAColorTableDelete                     = 11,
    kQATextureBindColorTable                = 12,
    kQABitmapBindColorTable                 = 13
} TQAEngineMethodTag;
```

Constant descriptions

`kQADrawPrivateNew` The `TQADrawPrivateNew` method.

`kQADrawPrivateDelete`
The `TQADrawPrivateDelete` method.

`kQAEngineCheckDevice`
The `TQAEngineCheckDevice` method.

`kQAEngineGestalt` The `TQAEngineGestalt` method.

`kQATextureNew` The `TQATextureNew` method.

`kQATextureDetach` The `TQATextureDetach` method.

`kQATextureDelete` The `TQATextureDelete` method.

`kQABitmapNew` The `TQABitmapNew` method.

`kQABitmapDetach` The `TQABitmapDetach` method.

`kQABitmapDelete` The `TQABitmapDelete` method.

`kQAColorTableNew` The `TQAColorTableNew` method.

`kQAColorTableDelete`
The `TQAColorTableDelete` method.

`kQATextureBindColorTable`
The `TQATextureBindColorTable` method.

`kQABitmapBindColorTable`
The `TQABitmapBindColorTable` method.

Public Draw Context Method Selectors

The `methodTag` parameter passed to the `QARegisterDrawMethod` function specifies a type of public draw context method. QuickDraw 3D RAVE defines these constants for method selectors.

```
typedef enum TQADrawMethodTag {
    kQASetFloat                                 = 0,
    kQASetInt                                   = 1,
    kQASetPtr                                   = 2,
    kQAGetFloat                                 = 3,
    kQAGetInt                                   = 4,
    kQAGetPtr                                   = 5,
    kQADrawPoint                                = 6,
    kQADrawLine                                 = 7,
    kQADrawTriGouraud                           = 8,
    kQADrawTriTexture                           = 9,
    kQADrawVGouraud                             = 10,
    kQADrawVTexture                             = 11,
    kQADrawBitmap                               = 12,
    kQARenderStart                              = 13,
    kQARenderEnd                                = 14,
    kQARenderAbort                              = 15,
    kQAFlush                                    = 16,
    kQASync                                     = 17,
    kQASubmitVerticesGouraud                    = 18,
    kQASubmitVerticesTexture                    = 19,
    kQADrawTriMeshGouraud                       = 20,
    kQADrawTriMeshTexture                       = 21,
    kQASetNoticeMethod                          = 22,
    kQAGetNoticeMethod                          = 23
} TQADrawMethodTag;
```

Constant descriptions

`kQASetFloat` The `TQASetFloat` method.

`kQASetInt` The `TQASetInt` method.

`kQASetPtr` The `TQASetPtr` method.

`kQAGetFloat`	The `TQAGetFloat` method.
`kQAGetInt`	The `TQAGetInt` method.
`kQAGetPtr`	The `TQAGetPtr` method.
`kQADrawPoint`	The `TQADrawPoint` method.
`kQADrawLine`	The `TQADrawLine` method.
`kQADrawTriGouraud`	The `TQADrawTriGouraud` method.
`kQADrawTriTexture`	The `TQADrawTriTexture` method.
`kQADrawVGouraud`	The `TQADrawVGouraud` method.
`kQADrawVTexture`	The `TQADrawVTexture` method.
`kQADrawBitmap`	The `TQADrawBitmap` method.
`kQARenderStart`	The `TQARenderStart` method.
`kQARenderEnd`	The `TQARenderEnd` method.
`kQARenderAbort`	The `TQARenderAbort` method.
`kQAFlush`	The `TQAFlush` method.
`kQASync`	The `TQASync` method.
`kQASubmitVerticesGouraud`	The `TQASubmitVerticesGouraud` method.
`kQASubmitVerticesTexture`	The `TQASubmitVerticesTexture` method.
`kQADrawTriMeshGouraud`	The `TQADrawTriMeshGouraud` method.
`kQADrawTriMeshTexture`	The `TQADrawTriMeshTexture` method.
`kQASetNoticeMethod`	The `TQASetNoticeMethod` method.
`kQAGetNoticeMethod`	The `TQAGetNoticeMethod` method.

Notice Method Selectors

The `method` parameter passed to the `QAGetNoticeMethod` and `QASetNoticeMethod` functions specifies a type of notice method. QuickDraw 3D RAVE defines these constants for method selectors.

```
typedef enum TQAMethodSelector {
    kQAMethod_RenderCompletion                  = 0,
    kQAMethod_DisplayModeChanged                = 1
} TQAMethodSelector;
```

Constant descriptions

`kQAMethod_RenderCompletion`
The renderer has finished rendering an image.

`kQAMethod_DisplayModeChanged`
The display mode has changed.

Data Structures

This section describes the data structures provided by QuickDraw 3D RAVE.

Memory Device Structure

You specify a memory device using a **memory device structure,** defined by the `TQADeviceMemory` data type.

```
typedef struct TQADeviceMemory {
    long                                rowBytes;
    TQAImagePixelType                   pixelType;
    long                                width;
    long                                height;
    void                                *baseAddr;
} TQADeviceMemory;
```

Field descriptions

`rowBytes` The distance, in bytes, from the beginning of one row of the memory device to the beginning of the next row of the memory device.

`pixelType` A value that specifies the size and organization of the memory associated with a pixel in the pixmap. See “Pixel Types” (page 1-35) for information on the values you can assign to this field.

`width` The width, in pixels, of the memory device.

`height` The height, in pixels, of the memory device.

`baseAddr` A pointer to the beginning of the memory device.

Rectangle Structure

You specify a rectangular region of memory (for instance, to define the area into which a drawing engine is to draw) using a **rectangle structure,** defined by the `TQARect` data type. All values are interpreted to be in device coordinates.

```
typedef struct TQARect {
    long                                left;
    long                                right;
    long                                top;
    long                                bottom;
} TQARect;
```

Field descriptions

`left`	The left side of the rectangle.
`right`	The right side of the rectangle.
`top`	The top side of the rectangle.
`bottom`	The bottom side of the rectangle.

Macintosh Device and Clip Structures

QuickDraw 3D RAVE supports two types of devices and one type of clipping on the Macintosh Operating System. The available devices and clipping are defined by unions of type `TQAPlatformDevice` and `TQAPlatformClip`.

```
typedef union TQAPlatformDevice {
    TQADeviceMemory                     memoryDevice;
    GDHandle                            gDevice;
} TQAPlatformDevice;
```

Field descriptions

`memoryDevice`	A memory device data structure.
`gDevice`	A handle to a graphics device (of type `GDevice`).

```
typedef union TQAPlatformClip {
    RgnHandle                           clipRgn;
} TQAPlatformClip;
```

Field descriptions

`clipRgn` A handle to a clipping region.

Windows Device and Clip Structures

QuickDraw 3D RAVE supports two types of devices and one type of clipping on Windows 32 systems. The available devices and clipping are defined by unions of type `TQAPlatformDevice` and `TQAPlatformClip`.

```
typedef union TQAPlatformDevice {
    TQADeviceMemory                     memoryDevice;
    HDC                                 hdc;
    struct {
        LPDIRECTDRAW                    lpDirectDraw;
        LPDIRECTDRAWSURFACE             lpDirectDrawSurface;
    };
} TQAPlatformDevice;
```

Field descriptions

`memoryDevice` A memory device data structure.

`hdc` A handle to a draw context.

`lpDirectDraw` [To be supplied.]

`lpDirectDrawSurface` [To be supplied.]

```
typedef union TQAPlatformClip {
    HRGN                                clipRgn;
} TQAPlatformClip
```

Field descriptions

`clipRgn` A handle to a clipping region.

Generic Device and Clip Structures

QuickDraw 3D RAVE supports one type of device and one type of clipping on generic operating systems. The available device and clipping are defined by unions of type `TQAPlatformDevice` and `TQAPlatformClip`.

```
typedef union TQAPlatformDevice {
    TQADeviceMemory                     memoryDevice;
} TQAPlatformDevice;
```

Field descriptions

`memoryDevice` A memory device data structure.

```
typedef union TQAPlatformClip {
    void                                *region;
} TQAPlatformClip;
```

Field descriptions

`region` **[To be supplied.]**

Device Structure

You specify a device (for example, when creating a new draw context with the `QADrawContextNew` function) by filling in a device structure, defined by the `TQADevice` data type.

```
typedef struct TQADevice {
    TQADeviceType                       deviceType;
    TQAPlatformDevice                   device;
} TQADevice;
```

Field descriptions

`deviceType` The device type. See "Device Types" (page 1-38) for information on the types of devices that are currently supported.

`device` A platform device data structure.

Clip Data Structure

You specify a clipping region (for example, when creating a new draw context with the `QADrawContextNew` function) by filling in a clip data structure, defined by the `TQAClip` data type. The clipping region determines which pixels are drawn to a device.

```
typedef struct TQAClip {
    TQAClipType                 clipType;
    TQAPlatformClip             clip;
} TQAClip;
```

Field descriptions

`clipType`	The clip type. See "Clip Types" (page 1-38) for the values you can assign to this field.
`clip`	A platform clip data structure.

Image Structure

Texture maps and bitmaps are defined using pixel images (or pixmaps). To specify a pixel image, you fill in an **image structure,** defined by the `TQAImage` data structure.

```
struct TQAImage {
    long                        width;
    long                        height;
    long                        rowBytes;
    void                        *pixmap;
};
typedef struct TQAImage TQAImage;
```

Field descriptions

`width`	The width, in pixels, of the pixmap.
`height`	The height, in pixels, of the pixmap.
`rowBytes`	The distance, in bytes, from the beginning of one row of the image data to the beginning of the next row of the image data. (For some low-cost accelerators, setting the value in this field to the product of the value in the `width` field and the pixel size improves performance.)
`pixmap`	A pointer to the image data.

Vertex Structures

QuickDraw 3D RAVE supports two different types of vertices: Gouraud vertices and texture vertices. You use **Gouraud vertices** for drawing Gouraud-shaded triangles, and also for drawing points and lines. A Gouraud

vertex is defined by the TQAVGouraud data structure, which specifies the position, depth, color, and transparency information.

```
typedef struct TQAVGouraud {
    float                               x;
    float                               y;
    float                               z;
    float                               invW;
    float                               r;
    float                               g;
    float                               b;
    float                               a;
} TQAVGouraud;
```

Field descriptions

x	The *x* coordinate of the vertex relative to the upper-left corner of the draw context rectangle (that is, the rectangle passed to the QADrawContextNew function). The value of this field is a floating-point value that specifies a number of pixels.
y	The *y* coordinate of the vertex relative to the upper-left corner of the draw context rectangle (that is, the rectangle passed to the QADrawContextNew function). The value of this field is a floating-point value that specifies a number of pixels.
z	The depth of the vertex. The value of this field is a floating-point number between 0.0 and 1.0 inclusive, where lower numbers specify points closer to the origin.
invW	The inverse *w* value (that is, the value 1/*w*, where *w* is the homogeneous correction factor). This field is valid only for drawing engines that support the kQAOptional_PerspectiveZ feature. When the state variable kQATag_PerspectiveZ is set to kQAPerspectiveZ_On, hidden surface removal is performed using the value in this field rather than the value in the z field, thereby causing the hidden surface removal to be perspective corrected.
r	The red component of the vertex color.
g	The green component of the vertex color.
b	The blue component of the vertex color.

`a` The alpha channel value of the vertex, where 1.0 represents opacity and 0.0 represents complete transparency.

You use **texture vertices** to define triangles to which a texture is to be mapped. A texture vertex is defined by the `TQAVTexture` data structure, which specifies the position, depth, transparency, and texture mapping information.

Note
Not all the fields of a `TQAVTexture` data structure need to be filled out. Many of these fields are used only when texture mapping operations are in force (that is, when the `kQATag_TextureOp` state variable has some value other than `kQATextureOp_None`). ◆

```
typedef struct TQAVTexture {
    float                               x;
    float                               y;
    float                               z;
    float                               invW;
    float                               r;
    float                               g;
    float                               b;
    float                               a;
    float                               uOverW;
    float                               vOverW;
    float                               kd_r;
    float                               kd_g;
    float                               kd_b;
    float                               ks_r;
    float                               ks_g;
    float                               ks_b;
} TQAVTexture;
```

Field descriptions

`x` The *x* coordinate of the vertex relative to the upper-left corner of the draw context rectangle (that is, the rectangle passed to the `QADrawContextNew` function). The value of this field is a floating-point value that specifies a number of pixels.

`y` The *y* coordinate of the vertex relative to the upper-left corner of the draw context rectangle (that is, the rectangle

	passed to the `QADrawContextNew` function). The value of this field is a floating-point value that specifies a number of pixels.
`z`	The depth of the vertex. The value of this field is a floating-point number between 0.0 and 1.0 inclusive, where lower numbers specify points closer to the origin.
`invW`	The inverse *w* value (that is, the value 1/*w*, where *w* is the homogeneous correction factor). This field must contain a value. For drawing engines that support the `kQAOptional_PerspectiveZ` feature and when the state variable `kQATag_PerspectiveZ` is set to `kQAPerspectiveZ_On`, hidden surface removal is performed using the value in this field rather than the value in the `z` field. For non-perspective rendering, this field should be set to 1.0.
`r`	The red component of the decal color. The value in this field is used only when the `kQATextureOp_Decal` texture mapping operation is enabled.
`g`	The green component of the decal color. The value in this field is used only when the `kQATextureOp_Decal` texture mapping operation is enabled.
`b`	The blue component of the decal color. The value in this field is used only when the `kQATextureOp_Decal` texture mapping operation is enabled.
`a`	The alpha channel value of the vertex, where 1.0 represents opacity and 0.0 represents complete transparency.
`uOverW`	The perspective-corrected *u* coordinate of the vertex.
`vOverW`	The perspective-corrected *v* coordinate of the vertex.
`kd_r`	The red component of the diffuse color of the vertex. The value in this field is used only when the `kQATextureOp_Modulate` texture mapping operation is enabled. The value in this field can be greater than 1.0 to more accurately render scenes with high light intensities.
`kd_g`	The green component of the diffuse color of the vertex. The value in this field is used only when the `kQATextureOp_Modulate` texture mapping operation is enabled. The value in this field can be greater than 1.0 to more accurately render scenes with high light intensities.

kd_b	The blue component of the diffuse color of the vertex. The value in this field is used only when the kQATextureOp_Modulate texture mapping operation is enabled. The value in this field can be greater than 1.0 to more accurately render scenes with high light intensities.
ks_r	The red component of the specular color of the vertex. The value in this field is used only when the kQATextureOp_Highlight texture mapping operation is enabled.
ks_g	The green component of the specular color of the vertex. The value in this field is used only when the kQATextureOp_Highlight texture mapping operation is enabled.
ks_b	The blue component of the specular color of the vertex. The value in this field is used only when the kQATextureOp_Highlight texture mapping operation is enabled.

IMPORTANT

A drawing engine may choose to use a single modulation value instead of the three values kd_r, kd_g, and kd_b. This change is transparent to applications, except that colored lights applied to a texture appear white. As a result, a drawing engine that uses this simplification must negate the kQAOptional_TextureColor bit in the optional features value returned by QAEngineGestalt. Similarly, a drawing engine may choose to use a single highlight value instead of the three values ks_r, ks_g, and ks_b. This change is transparent to applications, except that a texture-mapped object's specular highlight appears white, not colored. As a result, a drawing engine that uses this simplification must negate the kQAOptional_TextureColor bit in the optional features value. ▲

Draw Context Structure

QuickDraw 3D RAVE drawing routines operate on a draw context, which maintains state information and other data associated with a drawing engine. You access a draw context using a **draw context structure,** defined by the TQADrawContext data type.

IMPORTANT

You should not directly access the fields of a draw context structure. Instead, you should use the draw context manipulation macros defined by QuickDraw 3D RAVE. See "Manipulating Draw Contexts," beginning on page 1-94 for more information. ▲

```
struct TQADrawContext {
    TQADrawPrivate                  *drawPrivate;
    const TQAVersion                version;
    TQASetFloat                     setFloat;
    TQASetInt                       setInt;
    TQASetPtr                       setPtr;
    TQAGetFloat                     getFloat;
    TQAGetInt                       getInt;
    TQAGetPtr                       getPtr;
    TQADrawPoint                    drawPoint;
    TQADrawLine                     drawLine;
    TQADrawTriGouraud               drawTriGouraud;
    TQADrawTriTexture               drawTriTexture;
    TQADrawVGouraud                 drawVGouraud;
    TQADrawVTexture                 drawVTexture;
    TQADrawBitmap                   drawBitmap;
    TQARenderStart                  renderStart;
    TQARenderEnd                    renderEnd;
    TQARenderAbort                  renderAbort;
    TQAFlush                        flush;
    TQASync                         sync;
    TQASubmitVerticesGouraud        submitVerticesGouraud;
    TQASubmitVerticesTexture        submitVerticesTexture;
    TQADrawTriMeshGouraud           drawTriMeshGouraud;
    TQADrawTriMeshTexture           drawTriMeshTexture;
    TQASetNoticeMethod              setNoticeMethod;
    TQAGetNoticeMethod              getNoticeMethod;
};
typedef struct TQADrawContext TQADrawContext;
```

Field descriptions

`drawPrivate` A pointer to the private data for the drawing engine associated with this draw context.

`version`	The version of QuickDraw 3D RAVE. This field is initialized when you call `QADrawContextNew`. See "Version Values" (page 1-35) for the currently defined version numbers.
`setFloat`	A function pointer to the drawing engine's method for setting floating-point state variables.
`setInt`	A function pointer to the drawing engine's method for setting unsigned long integer state variables.
`setPtr`	A function pointer to the drawing engine's method for setting pointer state variables.
`getFloat`	A function pointer to the drawing engine's method for getting floating-point state variables.
`getInt`	A function pointer to the drawing engine's method for getting unsigned long integer state variables.
`getPtr`	A function pointer to the drawing engine's method for getting pointer state variables.
`drawPoint`	A function pointer to the drawing engine's method for drawing points.
`drawLine`	A function pointer to the drawing engine's method for drawing lines.
`drawTriGouraud`	A function pointer to the drawing engine's method for drawing triangles with Gouraud shading.
`drawTriTexture`	A function pointer to the drawing engine's method for drawing texture-mapped triangles.
`drawVGouraud`	A function pointer to the drawing engine's method for drawing vertices with Gouraud shading.
`drawVTexture`	A function pointer to the drawing engine's method for drawing texture-mapped vertices.
`drawBitmap`	A function pointer to the drawing engine's method for drawing a bitmap.
`renderStart`	A function pointer to the drawing engine's method for initializing in preparation for rendering.
`renderEnd`	A function pointer to the drawing engine's method for completing a rendering operation and displaying an image.
`renderAbort`	A function pointer to the drawing engine's method for canceling the current rendering operation and flushing any queued operations.

`flush`	A function pointer to the drawing engine's method for starting to render all queued drawing commands.
`sync`	A function pointer to the drawing engine's method for waiting until all queued drawing commands have been processed.
`submitVerticesGouraud`	A function pointer to the drawing engine's method for submitting Gouraud vertices.
`submitVerticesTexture`	A function pointer to the drawing engine's method for submitting texture vertices.
`drawTriMeshGouraud`	A function pointer to the drawing engine's method for drawing triangle meshes with Gouraud shading.
`drawTriMeshTexture`	A function pointer to the drawing engine's method for drawing texture-mapped triangle meshes.
`setNoticeMethod`	A function pointer to the drawing engine's method for setting a notice method.
`getNoticeMethod`	A function pointer to the drawing engine's method for getting a notice method.

Indexed Triangle Structure

The `QADrawTriMeshGouraud` and `QADrawTriMeshTexture` functions draw triangle meshes defined by an array of indexed triangles. An indexed triangle is represented by a data structure of type `TQAIndexedTriangle` that defines three vertices and a set of triangle flags.

```
typedef struct TQAIndexedTriangle {
    unsigned long                       triangleFlags;
    unsigned long                       vertices[3];
} TQAIndexedTriangle;
```

Field descriptions

`triangleFlags`	A set of triangle flags. See "Triangle Flags Masks," beginning on page 1-63 for a complete description of the available flags.

vertices	An array of three indices into the array of vertices submitted by the most recent call to QASubmitVerticesGouraud or QASubmitVerticesTexture.

QuickDraw 3D RAVE Routines

This section describes the routines provided by QuickDraw 3D RAVE.

Creating and Deleting Draw Contexts

QuickDraw 3D RAVE provides routines that you can use to create and delete draw contexts.

QADrawContextNew

You can use the QADrawContextNew function to create a new draw context.

```
TQAError QADrawContextNew (
                    const TQADevice *device,
                    const TQARect *rect,
                    const TQAClip *clip,
                    const TQAEngine *engine,
                    unsigned long flags,
                    TQADrawContext **newDrawContext);
```

device	A device.
rect	The rectangular region (specified in device coordinates) of the specified device that can be drawn into by the drawing engine associated with the new draw context.
clip	The two-dimensional clipping region for the new draw context, or NULL if no clipping is desired. This parameter must be set to NULL for devices of type kQADeviceMemory.
engine	A drawing engine.

`flags` A set of bit flags specifying features of the new draw context. See “Draw Context Flags Masks” (page 1-65) for complete information.

`newDrawContext`
On entry, the address of a pointer variable. On exit, that variable points to a new draw context. If a new draw context cannot be created, `*newDrawContext` is set to the value `NULL`.

DESCRIPTION

The `QADrawContextNew` function returns, through the `newDrawContext` parameter, a new draw context associated with the device specified by the `device` parameter and the drawing engine specified by the `engine` parameter.

QADrawContextDelete

You can use the `QADrawContextDelete` function to delete a draw context.

```
void QADrawContextDelete (TQADrawContext *drawContext);
```

`drawContext` A draw context.

DESCRIPTION

The `QADrawContextDelete` function deletes the draw context specified by the `drawContext` parameter. Any memory and other resources associated with that draw context are released.

Creating and Deleting Color Lookup Tables

QuickDraw 3D RAVE provides routines that you can use to create and dispose of color lookup tables.

QAColorTableNew

You can use the QAColorTableNew function to create a new color lookup table.

```
TQAError QAColorTableNew (
                    const TQAEngine *engine,
                    TQAColorTableType tableType,
                    void *pixelData,
                    long transparentIndexFlag,
                    TQAColorTable **newTable);
```

`engine` A drawing engine.

`tableType` The type of the new color lookup table. See "Color Lookup Table Types" (page 1-37) for information on the available color lookup table types.

`pixelData` A pointer to the color lookup table entries.

`transparentIndexFlag`
A long integer, interpreted as a Boolean value, that indicates whether the color lookup table entry at index 0 is completely transparent (`TRUE`) or not (`FALSE`).

`newTable` On entry, the address of a pointer variable. On exit, that variable points to a new color lookup table. If a new color lookup table cannot be created, `*newTable` is set to the value `NULL`.

DESCRIPTION

The QAColorTableNew function returns, through the newTable parameter, a new color lookup table associated with the drawing engine specified by the engine parameter. The table entries for the new color lookup table are copied from the block of data pointed to by the pixelData parameter; if QAColorTableNew completes successfully, you can dispose of that block of memory. The data in that block of memory is interpreted according to the format specified by the tableType parameter. For example, if tableType is kQAColorTable_CL8_RGB32, then pixelData should point to a block of data that is at least 1024 bytes long and in which each 32-bit quantity is an RGB color value.

IMPORTANT

Currently, QuickDraw 3D RAVE supports only 32-bit RGB color lookup table entries. The specified drawing engine might reduce the size of individual color lookup table entries to fit into its on-board memory. ▲

Not all drawing engines support color lookup tables, and QuickDraw 3D RAVE does not provide color lookup table emulation for engines that do not support them.

SEE ALSO

Use the `QAColorTableDelete` function (next) to delete a color lookup table. Use the `QATextureBindColorTable` function (page 1-87) to bind a color lookup table to a texture map. Use the `QABitmapBindColorTable` function (page 1-90) to bind a color lookup table to a bitmap.

QAColorTableDelete

You can use the `QAColorTableDelete` function to delete a color lookup table.

```
void QAColorTableDelete (
                    const TQAEngine *engine,
                    TQAColorTable *colorTable);
```

`engine` A drawing engine.

`colorTable` A color lookup table.

DESCRIPTION

The `QAColorTableDelete` function deletes the color lookup table specified by the `colorTable` parameter. Any memory and other resources associated with that color lookup table are released.

SEE ALSO

Use the `QAColorTableNew` function (page 1-83) to create a color lookup table.

Manipulating Textures and Bitmaps

QuickDraw 3D RAVE provides routines that you can use to create and dispose of texture maps and bitmaps. It also provides routines that you can use to bind color lookup tables to texture maps and bitmaps.

QATextureNew

You can use the `QATextureNew` function to create a new texture map.

```
TQAError QATextureNew (
                    const TQAEngine *engine,
                    unsigned long flags,
                    TQAImagePixelType pixelType,
                    const TQAImage images[],
                    TQATexture **newTexture);
```

`engine` A drawing engine.

`flags` A set of bit flags specifying features of the new texture map. See “Texture Flags Masks” (page 1-64) for complete information.

`pixelType` The type of pixels in the new texture map. See “Pixel Types” (page 1-35) for a description of the values you can pass in this parameter.

`images` An array of pixel images to use for the new texture map. The values in the `width` and `height` fields of these structures must be an even power of 2.

`newTexture` On entry, the address of a pointer variable. On exit, that variable points to a new texture map. If a new texture map cannot be created, `*newTexture` is set to the value `NULL`.

DESCRIPTION

The `QATextureNew` function returns, through the `newTexture` parameter, a new texture map associated with the drawing engine specified by the `engine` parameter. You can use the returned texture map to set the value of the `kQATag_Texture` state variable.

The `flags` parameter specifies a set of texture map features. If the `kQATexture_Lock` bit in that parameter is set but the drawing engine cannot guarantee that the texture will remain locked in memory, the `QATextureNew` function returns an error.

If the `kQATexture_Mipmap` bit of the `flags` parameter is clear, the `images` parameter points to a single pixel image that defines the texture map. If the `kQATexture_Mipmap` bit is set, the `images` parameter points to an array of pixel images of varying pixel depths. The first element in the array must be the mipmap page having the highest resolution, with a width and height that are even powers of 2. Each subsequent pixel image in the array should have a width and height that are half those of the previous image (with a minimum width and height of 1).

SPECIAL CONSIDERATIONS

`QATextureNew` does not automatically copy the pixmap data pointed to by the `images` parameter. As a result, you should not release or reuse the storage occupied by the pixel images until you've called `QATextureDetach`. Note, however, that `QATextureNew` does copy all of the information contained in the `TQAImage` structures in the array, so you can free or reuse that memory after `QATextureNew` completes successfully.

QATextureDetach

You can use the `QATextureDetach` function to detach a texture map from a drawing engine.

```
TQAError QATextureDetach (const TQAEngine *engine, TQATexture *texture);
```

`engine` A drawing engine.

`texture` A texture map.

DESCRIPTION

The `QATextureDetach` function causes the drawing engine specified by the `engine` parameter to copy the data associated with the texture map specified by

the texture parameter. Once the data are copied, you can reuse or dispose of the memory you originally specified in a call to QATextureNew.

QATextureBindColorTable

You can use the QATextureBindColorTable function to bind a color lookup table to a texture map.

```
TQAError QATextureBindColorTable (
                    const TQAEngine *engine,
                    TQATexture *texture,
                    TQAColorTable *colorTable);
```

engine	A drawing engine.
texture	A texture map.
colorTable	A color lookup table (as returned by a previous call to QAColorTableNew).

DESCRIPTION

The QATextureBindColorTable function binds the color lookup table specified by the colorTable parameter to the texture map specified by the texture parameter. Before you can draw any texture map whose pixel type is either kQAPixel_CL4 or kQAPixel_CL8, you must bind a color lookup table to it. In addition, the type of the specified color lookup table must match that of the pixel type of the texture map to which it is bound. For example, a color lookup table of type kQAColorTable_CL8_RGB32 can be bound only to a texture map whose pixel type is kQAPixel_CL8.

QATextureDelete

You can use the QATextureDelete function to delete a texture map.

```
void QATextureDelete (const TQAEngine *engine, TQATexture *texture);
```

engine	A drawing engine.
texture	A texture map.

DESCRIPTION

The QATextureDelete function deletes the texture map specified by the texture parameter from the drawing engine specified by the engine parameter.

QABitmapNew

You can use the QABitmapNew function to create a new bitmap.

```
TQAError QABitmapNew (
                    const TQAEngine *engine,
                    unsigned long flags,
                    TQAImagePixelType pixelType,
                    const TQAImage *image,
                    TQABitmap **newBitmap);
```

engine	A drawing engine.
flags	A set of bit flags specifying features of the new bitmap. See "Bitmap Flags Masks" (page 1-64) for complete information
pixelType	The type of pixels in the new bitmap. See "Pixel Types" (page 1-35) for a description of the values you can pass in this parameter.
image	A pixel image to use for the new bitmap. The width and height fields of this image can have any values greater than 0.
newBitmap	On entry, the address of a pointer variable. On exit, that variable points to a new bitmap. If a new bitmap cannot be created, *newBitmap is set to the value NULL.

DESCRIPTION

The QABitmapNew function returns, through the newBitmap parameter, a pointer to a new bitmap associated with the drawing engine specified by the engine

parameter. You can draw the returned bitmap by calling the `QADrawBitmap` function.

The `flags` parameter specifies a set of bitmap features. If the `kQABitmap_Lock` bit in that parameter is set but the drawing engine cannot guarantee that the bitmap will remain locked in memory, the `QABitmapNew` function returns an error.

SPECIAL CONSIDERATIONS

`QABitmapNew` does not automatically copy the pixmap data pointed to by the `images` parameter. As a result, you should not release or reuse the storage occupied by the pixel image until you've called `QABitmapDetach`. Note, however, that `QABitmapNew` does copy all of the information contained in the `TQAImage` structure, so you can free or reuse that memory after `QABitmapNew` completes successfully.

QABitmapDetach

You can use the `QABitmapDetach` function to detach a bitmap from a drawing engine.

```
TQAError QABitmapDetach (const TQAEngine *engine, TQABitmap *bitmap);
```

`engine`	A drawing engine.
`bitmap`	A bitmap.

DESCRIPTION

The `QABitmapDetach` function causes the drawing engine specified by the `engine` parameter to copy the data associated with the bitmap specified by the `bitmap` parameter. Once the data are copied, you can reuse or dispose of the memory you originally specified in a call to `QABitmapNew`.

QABitmapBindColorTable

You can use the `QABitmapBindColorTable` function to bind a color lookup table to a bitmap.

```
TQAError QABitmapBindColorTable (
                    const TQAEngine *engine,
                    TQABitmap *bitmap,
                    TQAColorTable *colorTable);
```

`engine` A drawing engine.

`bitmap` A bitmap.

`colorTable` A color lookup table (as returned by a previous call to `QAColorTableNew`).

DESCRIPTION

The `QABitmapBindColorTable` function binds the color lookup table specified by the `colorTable` parameter to the bitmap specified by the `bitmap` parameter. Before you can draw any bitmap whose pixel type is either `kQAPixel_CL4` or `kQAPixel_CL8`, you must bind a color lookup table to it. In addition, the type of the specified color lookup table must match that of the pixel type of the bitmap to which it is bound. For example, a color lookup table of type `kQAColorTable_CL8_RGB32` can be bound only to a bitmap whose pixel type is `kQAPixel_CL8`.

QABitmapDelete

You can use the `QABitmapDelete` function to delete a bitmap.

```
void QABitmapDelete (const TQAEngine *engine, TQABitmap *bitmap);
```

`engine` A drawing engine.

`bitmap` A bitmap.

DESCRIPTION

The `QABitmapDelete` function deletes the bitmap specified by the `bitmap` parameter from the drawing engine specified by the `engine` parameter.

Managing Drawing Engines

QuickDraw 3D RAVE provides routines that you can use to manage drawing engines. For example, you can use these routines to find a drawing engine for a particular device.

QADeviceGetFirstEngine

You can use the `QADeviceGetFirstEngine` function to get the first drawing engine that can draw to a particular device.

```
TQAEngine *QADeviceGetFirstEngine (const TQADevice *device);
```

`device` A device, or the value `NULL`.

DESCRIPTION

The `QADeviceGetFirstEngine` function returns, as its function result, the first drawing engine that is capable of drawing into the device specified by the `device` parameter. The first engine is defined to be the first engine that satisfies one of these criteria:

1. The drawing engine selected by the user (for example, using the RAVE control panel).
2. The drawing engine that is tightly coupled to the specified device (that is, that can render only to that device).
3. The drawing engine that accelerates more features than any other drawing engine.

If you pass the value `NULL` in the `device` parameter, `QADeviceGetFirstEngine` returns a drawing engine without regard for its ability to drive any particular device. You can use this technique to find all available engines.

QADeviceGetNextEngine

You can use the `QADeviceGetNextEngine` function to get the next drawing engine that can draw to a particular device.

```
TQAEngine *QADeviceGetNextEngine (
                    const TQADevice *device,
                    const TQAEngine *currentEngine);
```

`device` A device, or the value `NULL`.

`currentEngine` A drawing engine.

DESCRIPTION

The `QADeviceGetNextEngine` function returns, as its function result, the drawing engine that supports the device specified by the `device` parameter that follows the engine specified by the `currentEngine` parameter. The value you pass in the `currentEngine` parameter should have been obtained from a previous call to `QADeviceGetFirstEngine` or `QADeviceGetNextEngine`.

If you pass the value `NULL` in the `device` parameter, `QADeviceGetNextEngine` returns a the next drawing engine without regard for its ability to drive any particular device. You can use this technique to find all available engines.

QAEngineCheckDevice

You can use the `QAEngineCheckDevice` function to determine whether a particular drawing engine can draw into a particular device.

```
TQAError QAEngineCheckDevice (
                    const TQAEngine *engine,
                    const TQADevice *device);
```

`engine` A drawing engine.

`device` A device.

DESCRIPTION

The `QAEngineCheckDevice` function returns, as its function result, a code that indicates whether the drawing engine specified by the `engine` parameter can draw into the device specified by the `device` parameter (`kQANoErr`) or not (`kQAError`).

QAEngineGestalt

You can use the `QAEngineGestalt` function to get information about a drawing engine.

```
TQAError QAEngineGestalt (
                    const TQAEngine *engine,
                    TQAGestaltSelector selector,
                    void *response);
```

`engine` A drawing engine.

`selector` A selector that determines what kind of information is to be returned about the specified drawing engine. See “Gestalt Selectors” (page 1-57) for complete information about the available selectors and the information they return.

`response` A pointer to a buffer into which the returned information is copied. Your application is responsible for allocating this buffer. The size and meaning of the data copied to the buffer depend on the selector you pass in the `selector` parameter.

DESCRIPTION

The `QAEngineGestalt` function returns, in the `response` parameter, a buffer of information about features of the type specified by the `selector` parameter associated with the drawing engine specified by the `engine` parameter.

SEE ALSO

See “Finding a Drawing Engine” (page 1-16) for code illustrating how to call `QAEngineGestalt`.

QAEngineEnable

You can use the `QAEngineEnable` function to enable a drawing engine.

```
TQAError QAEngineEnable (long vendorID, long engineID);
```

`vendorID` A vendor ID.

`engineID` A drawing engine ID.

DESCRIPTION

The `QAEngineEnable` function enables the drawing engine specified by the `vendorID` and `engineID` parameters.

QAEngineDisable

You can use the `QAEngineDisable` function to disable a drawing engine.

```
TQAError QAEngineDisable (long vendorID, long engineID);
```

`vendorID` A vendor ID.

`engineID` An engine ID.

DESCRIPTION

The `QAEngineDisable` function disables the drawing engine specified by the `vendorID` and `engineID` parameters.

Manipulating Draw Contexts

QuickDraw 3D RAVE provides routines that you can use to manipulate draw contexts. For example, you can use the `QASetInt` routine to set an integer-valued state variable associated with a draw context.

IMPORTANT

These functions are currently implemented as C language macros that call the methods of the drawing engine. Your application should use these macros for all draw context manipulation. ▲

See the section "Application-Defined Routines," beginning on page 1-115 for complete information on the draw context methods invoked by these macros.

Note

There is one macro for each method whose address is stored in a draw context structure (of type `TQADrawContext`). ◆

QAGetFloat

You can use the `QAGetFloat` function to get a floating-point value of a draw context state variable.

```
#define QAGetFloat(drawContext,tag) \
                (drawContext)->getFloat (drawContext,tag)
```

`drawContext`	A draw context.
`tag`	A state variable tag.

DESCRIPTION

The `QAGetFloat` function returns, as its function result, the floating-point value of the draw context state variable specified by the `drawContext` and `tag` parameters. If the specified tag is not recognized or supported by that draw context, `QAGetFloat` returns the value 0.

QASetFloat

You can use the `QASetFloat` function to set a floating-point value for a draw context state variable.

```
#define QASetFloat(drawContext,tag,newValue) \
                (drawContext)->setFloat (drawContext,tag,newValue)
```

drawContext	A draw context.
tag	A state variable tag.
newValue	The new value of the specified state variable.

DESCRIPTION

The QASetFloat function sets the value of the draw context state variable specified by the drawContext and tag parameters to the floating-point value specified by the newValue parameter.

QAGetInt

You can use the QAGetInt function to get a long integer value of a draw context state variable.

```
#define QAGetInt(drawContext,tag) \
                (drawContext)->getInt (drawContext,tag)
```

drawContext	A draw context.
tag	A state variable tag.

DESCRIPTION

The QAGetInt function returns, as its function result, the long integer value of the draw context state variable specified by the drawContext and tag parameters. If the specified tag is not recognized or supported by that draw context, QAGetInt returns the value 0.

QASetInt

You can use the `QASetInt` function to set a long integer value for a draw context state variable.

```
#define QASetInt(drawContext,tag,newValue) \
                (drawContext)->setInt (drawContext,tag,newValue)
```

`drawContext`	A draw context.
`tag`	A state variable tag.
`newValue`	The new value of the specified state variable.

DESCRIPTION

The `QASetInt` function sets the value of the draw context state variable specified by the `drawContext` and `tag` parameters to the long integer value specified by the `newValue` parameter.

QAGetPtr

You can use the `QAGetPtr` function to get a pointer value of a draw context state variable.

```
#define QAGetPtr(drawContext,tag) \
                (drawContext)->getPtr (drawContext,tag)
```

`drawContext`	A draw context.
`tag`	A state variable tag.

DESCRIPTION

The `QAGetPtr` function returns, as its function result, the pointer value of the draw context state variable specified by the `drawContext` and `tag` parameters. If the specified tag is not recognized or supported by that draw context, `QAGetPtr` returns the value 0.

QASetPtr

You can use the `QASetPtr` function to set a pointer value for a draw context state variable.

```
#define QASetPtr(drawContext,tag,newValue) \
                (drawContext)->setPtr (drawContext,tag,newValue)
```

`drawContext`	A draw context.
`tag`	A state variable tag.
`newValue`	The new value of the specified state variable.

DESCRIPTION

The `QASetPtr` function sets the value of the draw context state variable specified by the `drawContext` and `tag` parameters to the pointer value specified by the `newValue` parameter.

QADrawPoint

You can use the `QADrawPoint` function to draw a point.

```
#define QADrawPoint(drawContext,v) \
                (drawContext)->drawPoint (drawContext,v)
```

`drawContext`	A draw context.
`v`	A Gouraud vertex.

DESCRIPTION

The `QADrawPoint` function draws the single point specified by the `v` parameter to the draw context specified by the `drawContext` parameter. The size of the point is determined by the `kQATag_Width` state variable of the draw context.

QADrawLine

You can use the `QADrawLine` function to draw a line between two points.

```
#define QADrawLine(drawContext,v0,v1) \
                (drawContext)->drawLine (drawContext,v0,v1)
```

`drawContext`	A draw context.
`v0`	A Gouraud vertex.
`v1`	A Gouraud vertex.

DESCRIPTION

The `QADrawLine` function draws the line specified by the `v0` and `v1` parameters to the draw context specified by the `drawContext` parameter. The size of the line is determined by the `kQATag_Width` state variable of the draw context. If the specified vertices have different colors, the line color is interpolated smoothly between the two vertex colors.

QADrawTriGouraud

You can use the `QADrawTriGouraud` function to draw Gouraud-shaded triangles.

```
#define QADrawTriGouraud(drawContext,v0,v1,v2,flags) \
        (drawContext)->drawTriGouraud (drawContext,v0,v1,v2,flags)
```

`drawContext`	A draw context.
`v0`	A Gouraud vertex.
`v1`	A Gouraud vertex.
`v2`	A Gouraud vertex.
`flags`	A set of triangle flags. See “Triangle Flags Masks,” beginning on page 1-63 for a complete description of the available flags.

DESCRIPTION

The `QADrawTriGouraud` function draws the Gouraud-shaded triangle determined by the three points specified by the `v0`, `v1`, and `v2` parameters into the draw context specified by the `drawContext` parameter. Features of the triangle are determined by the `flags` parameter. Currently, this parameter is used to specify an orientation for the triangle.

QADrawTriTexture

You can use the `QADrawTriTexture` function to draw texture-mapped triangles.

```
#define QADrawTriTexture(drawContext,v0,v1,v2,flags) \
        (drawContext)->drawTriTexture (drawContext,v0,v1,v2,flags)
```

`drawContext`	A draw context.
`v0`	A texture vertex.
`v1`	A texture vertex.
`v2`	A texture vertex.
`flags`	A set of triangle flags. See “Triangle Flags Masks,” beginning on page 1-63 for a complete description of the available flags.

DESCRIPTION

The `QADrawTriTexture` function draws the texture-mapped triangle determined by the three points specified by the `v0`, `v1`, and `v2` parameters into the draw context specified by the `drawContext` parameter. The texture used for the mapping is determined by the value of the `kQATag_Texture` state variable. Features of the triangle are determined by the `flags` parameter. Currently, this parameter is used to specify an orientation for the triangle.

SPECIAL CONSIDERATIONS

The `QADrawTriTexture` function is optional and must be supported only by drawing engines that support texture mapping.

QASubmitVerticesGouraud

You can use the `QASubmitVerticesGouraud` function to submit Gouraud vertices.

```
#define QASubmitVerticesGouraud(drawContext,nVertices,vertices) \
    (drawContext)->submitVerticesGouraud(drawContext,nVertices,vertices)
```

`drawContext`	A draw context.
`nVertices`	The number of Gouraud vertices pointed to by the `vertices` parameter.
`vertices`	A pointer to an array of Gouraud vertices.

DESCRIPTION

The `QASubmitVerticesGouraud` function submits the list of vertices pointed to by the `vertices` parameter to the draw context specified by the `drawContext` parameter. The vertices define a triangle mesh. Note, however, that `QASubmitVerticesGouraud` does not draw the specified mesh, but simply defines the mesh for a subsequent call to `QADrawTriMeshGouraud`.

Your application is responsible for managing the memory occupied by the Gouraud vertices. `QASubmitVerticesGouraud` does not copy the vertex data pointed to by the `vertices` parameter. Accordingly, you must not dispose of or reuse that memory until you've finished drawing the triangle mesh defined by `QASubmitVerticesGouraud`.

SPECIAL CONSIDERATIONS

If a drawing engine does not support triangle meshes, QuickDraw 3D RAVE decomposes a triangle mesh into individual triangles. As a result, you can always use the `QASubmitVerticesGouraud` function to submit a triangle mesh.

QASubmitVerticesTexture

You can use the `QASubmitVerticesTexture` function to submit texture vertices.

```
#define QASubmitVerticesTexture(drawContext,nVertices,vertices) \
    (drawContext)->submitVerticesTexture(drawContext,nVertices,vertices)
```

drawContext	A draw context.
nVertices	The number of texture vertices pointed to by the vertices parameter.
vertices	A pointer to an array of texture vertices.

DESCRIPTION

The QASubmitVerticesTexture function submits the list of vertices pointed to by the vertices parameter to the draw context specified by the drawContext parameter. The vertices define a triangle mesh. Note, however, that QASubmitVerticesTexture does not draw the specified mesh, but simply defines the mesh for a subsequent call to QADrawTriMeshTexture.

Your application is responsible for managing the memory occupied by the texture vertices. QASubmitVerticesTexture does not copy the vertex data pointed to by the vertices parameter. Accordingly, you must not dispose of or reuse that memory until you've finished drawing the triangle mesh defined by QASubmitVerticesTexture.

SPECIAL CONSIDERATIONS

The QASubmitVerticesTexture function is optional and must be supported only by drawing engines that support texture mapping.

If a drawing engine does not support triangle meshes, QuickDraw 3D RAVE decomposes a triangle mesh into individual triangles.

QADrawTriMeshGouraud

You can use the QADrawTriMeshGouraud function to draw a triangle mesh with Gouraud shading.

```
#define QADrawTriMeshGouraud(drawContext,nTriangle,triangles) \
    (drawContext)->drawTriMeshGouraud (drawContext,nTriangle,triangles)
```

drawContext	A draw context.
nTriangle	The number of indexed triangles pointed to by the triangles parameter.
triangles	A pointer to an array of indexed triangles. See “Indexed Triangle Structure” (page 1-80) for a description of indexed triangles.

DESCRIPTION

The QADrawTriMeshGouraud function draws, with Gouraud shading, the triangle mesh specified by the triangles parameter into the draw context specified by the drawContext parameter. Each triangle in the mesh is defined by a TQAIndexedTriangle data structure, which contains three indices into the array of Gouraud vertices previously submitted to the draw context by a call to the QASubmitVerticesGouraud function.

SPECIAL CONSIDERATIONS

QADrawTriMeshGouraud operates only on a triangle mesh previously submitted using the QASubmitVerticesGouraud function. Use QADrawTriMeshTexture to draw a triangle mesh submitted using the QASubmitVerticesTexture function.

QADrawTriMeshTexture

You can use the QADrawTriMeshTexture function to draw a texture-mapped triangle mesh.

```
#define QADrawTriMeshTexture(drawContext,nTriangle,triangles) \
    (drawContext)->drawTriMeshTexture (drawContext,nTriangle,triangles)
```

drawContext	A draw context.
nTriangle	The number of indexed triangles pointed to by the triangles parameter.
triangles	A pointer to an array of indexed triangles. See “Indexed Triangle Structure” (page 1-80) for a description of indexed triangles.

DESCRIPTION

The `QADrawTriMeshTexture` function draws the texture-mapped triangle mesh specified by the `triangles` parameter into the draw context specified by the `drawContext` parameter. Each triangle in the mesh is defined by a `TQAIndexedTriangle` data structure, which contains three indices into the array of texture vertices previously submitted to the draw context by a call to the `QASubmitVerticesTexture` function.

SPECIAL CONSIDERATIONS

`QADrawTriMeshTexture` operates only on a triangle mesh previously submitted using the `QASubmitVerticesTexture` function. Use `QADrawTriMeshGouraud` to draw a triangle mesh submitted using the `QASubmitVerticesGouraud` function.

The `QADrawTriMeshTexture` function is optional and must be supported only by drawing engines that support texture mapping.

QADrawVGouraud

You can use the `QADrawVGouraud` function to draw Gouraud-shaded objects defined by vertices.

```
#define QADrawVGouraud(drawContext,nVertices,vertexMode,vertices,flags) \
 (drawContext)->drawVGouraud(drawContext,nVertices,vertexMode,vertices,flags)
```

`drawContext`	A draw context.
`nVertices`	The number of vertices contained in the `vertices` array.
`vertexMode`	A vertex mode. See “Vertex Modes” (page 1-56) for a description of the available vertex modes.
`vertices`	An array of Gouraud vertices.
`flags`	An array of triangle flags, or the value `NULL`. See “Triangle Flags Masks” (page 1-63) for a description of the available triangle flags. This parameter is valid only if the `vertexMode` parameter contains the value `kQAVertexMode_Tri`, `kQAVertexMode_Strip`, or `kQAVertexMode_Fan`.

DESCRIPTION

The `QADrawVGouraud` function draws the vertices in the array specified by the `vertices` parameter into the draw context specified by the `drawContext` parameter, according to the vertex modes flag specified by the `vertexMode` parameter. For instance, if the value of the `vertexMode` parameter is `kQAVertexMode_Polyline`, then the vertices in that array are interpreted as defining a polyline (a set of connected line segments). Gouraud shading is applied to whatever objects are drawn.

SPECIAL CONSIDERATIONS

The `QADrawVGouraud` function is optional and must be supported only by drawing engines that do not want calls to `QADrawVGouraud` to be replaced by calls to the `QADrawPoint`, `QADrawLine`, or `QADrawTriGouraud` functions.

QADrawVTexture

You can use the `QADrawVTexture` function to draw texture-mapped objects defined by vertices.

```
#define QADrawVTexture(drawContext,nVertices,vertexMode,vertices,flags) \
 (drawContext)->drawVTexture(drawContext,nVertices,vertexMode,vertices,flags)
```

`drawContext`	A draw context.
`nVertices`	The number of vertices contained in the `vertices` array.
`vertexMode`	A vertex mode. See “Vertex Modes” (page 1-56) for a description of the available vertex modes.
`vertices`	An array of texture vertices.
`flags`	An array of triangle flags, or the value `NULL`. See “Triangle Flags Masks” (page 1-63) for a description of the available triangle flags. This parameter is valid only if the `vertexMode` parameter contains the value `kQAVertexMode_Tri`, `kQAVertexMode_Strip`, or `kQAVertexMode_Fan`.

DESCRIPTION

The `QADrawVTexture` function draws the vertices in the array specified by the `vertices` parameter into the draw context specified by the `drawContext` parameter, according to the vertex modes flag specified by the `vertexMode` parameter. For instance, if the value of the `vertexMode` parameter is `kQAVertexMode_Polyline`, then the vertices in that array are interpreted as defining a polyline (a set of connected line segments). Texture mapping (using the texture determined by the value of the `kQATag_Texture` state variable) is applied to whatever objects are drawn.

IMPORTANT

The vertex modes `kQAVertexMode_Point` and `kQAVertexMode_Line` are supported only by drawing engines that support the `kQAOptional_OpenGL` feature. All other drawing engines should ignore requests to texture map points or lines. ▲

SPECIAL CONSIDERATIONS

The `QADrawVTexture` function is optional and must be supported only by drawing engines that support texture mapping and do not want calls to `QADrawVTexture` to be replaced by calls to the `QADrawPoint`, `QADrawLine`, or `QADrawTriTexture` methods.

QADrawBitmap

You can use the `QADrawBitmap` function to draw bitmaps into a draw context.

```
#define QADrawBitmap(drawContext,v,bitmap) \
                    (drawContext)->drawBitmap (drawContext,v,bitmap)
```

`drawContext`	A draw context.
`v`	A Gouraud vertex.
`bitmap`	A pointer to a bitmap (returned by a previous call to `QABitmapNew`).

DESCRIPTION

The QADrawBitmap function draws the bitmap specified by the bitmap parameter into the draw context specified by the drawContext parameter, with the upper-left corner of the bitmap located at the point specified by the v parameter. The v parameter can contain negative values in its x or y fields, so you can position upper-left corner of the bitmap outside the draw context rectangle. This allows you to move the bitmap smoothly off any edge of the draw context.

QARenderStart

You can use the QARenderStart function to initialize a draw context before an engine performs any rendering into that context.

```
#define QARenderStart(drawContext,dirtyRect,initialContext) \
(drawContext)->renderStart (drawContext,dirtyRect,initialContext)
```

`drawContext` A draw context.

`dirtyRect` The minimum area of the specified draw context to clear, or the value NULL.

`initialContext` A previously cached draw context, or the value NULL.

DESCRIPTION

The QARenderStart function performs any operations necessary to initialize the draw context specified by the drawContext parameter. This includes clearing the z buffer and the color buffers of the draw context. If the value of the initialContext parameter is NULL, then QARenderStart clears the z buffer to 1.0 and sets the color buffers to the values of the kQATag_ColorBG_a, kQATag_ColorBG_r, kQATag_ColorBG_g, and kQATag_ColorBG_b draw context state variables. If, however, the value of the initialContext parameter is not NULL, then QARenderStart uses the previously cached draw context specified by that parameter to initialize the draw context specified by the drawContext parameter.

The dirtyRect parameter indicates the minimum area of the specified draw context to clear on initialization. If the value of the dirtyRect parameter is NULL, the entire draw context is cleared. If the value of the dirtyRect parameter is not

NULL, it indicates the rectangle in the draw context to clear. Some drawing engines may exhibit improved performance when an area that is smaller than the entire draw context rectangle is passed. However, the interpretation of the dirtyRect parameter is dependent on the drawing engine, which may choose to initialize the entire draw context. As a result, you should not use this parameter as a means to avoid clearing all of a draw context or to perform incremental rendering. Instead, you should use the initialContext parameter to achieve such effects.

SPECIAL CONSIDERATIONS

You should call QARenderStart before performing any rendering operations in the specified draw context, and you should call either QARenderEnd to signal the end of rendering operations or QARenderAbort to cancel rendering operations. However, when a drawing engine is performing OpenGL rendering, the QARenderStart function operates just like the OpenGL function glClear. In OpenGL mode, it is not necessary that a call to QARenderStart always be balanced by a matching call to QARenderEnd, and drawing commands may occur at any time.

SEE ALSO

See “Using a Draw Context as a Cache” (page 1-20) for information on creating a draw context cache (that is, a draw context you can use as the initial context specified in the initialContext parameter).

QARenderEnd

You can use the QARenderEnd function to signal the end of any rendering into a draw context.

```
#define QARenderEnd(drawContext,modifiedRect) \
                (drawContext)->renderEnd (drawContext,modifiedRect)
```

drawContext A draw context.

modifiedRect The minimum area of the back buffer of the specified draw context to display, or the value NULL.

DESCRIPTION

The `QARenderEnd` function performs any operations necessary to display an image rendered into the draw context specified by the `drawContext` parameter. If the draw context is double buffered, `QARenderEnd` displays the back buffer. If the draw context is single buffered, `QARenderEnd` calls `QAFlush`.

The `modifiedRect` parameter indicates the minimum area of the back buffer of the specified draw context that should be displayed. If the value of the `modifiedRect` parameter is `NULL`, the entire back buffer is displayed. If the value of the `modifiedRect` parameter is not `NULL`, it indicates the rectangle in the back buffer to display. Some drawing engines may exhibit improved performance when an area that is smaller than the entire draw context rectangle is passed (to avoid unnecessary pixel copying). However, the interpretation of the `modifiedRect` parameter is dependent on the drawing engine, which may choose to draw the entire back buffer.

The `QARenderEnd` function returns a result code (of type `TQAError`) indicating whether any errors have occurred since the previous call to `QARenderStart`. If all rendering commands completed successfully, the value `kQANoErr` is returned. If any other value is returned, you should assume that the rendered image is incorrect.

SPECIAL CONSIDERATIONS

You should call `QARenderStart` before performing any rendering operations in the specified draw context, and you should call either `QARenderEnd` to signal the end of rendering operations or `QARenderAbort` to cancel rendering operations. Once you have called `QARenderEnd`, you should not submit any drawing requests until you have called `QARenderStart` again.

QARenderAbort

You can use the `QARenderAbort` function to cancel any asynchronous drawing requests for a draw context.

```
#define QARenderAbort(drawContext) \
                        (drawContext)->renderAbort (drawContext)
```

`drawContext` A draw context.

DESCRIPTION

The `QARenderAbort` function immediately stops the draw context specified by the `drawContext` parameter from processing any asynchronous drawing commands it is currently processing and causes it to discard any queued commands.

The `QARenderAbort` function returns a result code (of type `TQAError`) indicating whether any errors have occurred since the previous call to `QARenderStart`. If all rendering commands completed successfully, the value `kQANoErr` is returned. If any other value is returned, you should assume that the rendered image is incorrect.

SPECIAL CONSIDERATIONS

You should call either `QARenderEnd` or `QARenderAbort`, but not both.

QAFlush

You can use the `QAFlush` function to flush a draw context.

```
#define QAFlush(drawContext) (drawContext)->flush (drawContext)
```

`drawContext` A draw context.

DESCRIPTION

The `QAFlush` function causes the drawing engine associated with the draw context specified by the `drawContext` parameter to begin rendering all drawing commands that are queued in a buffer awaiting processing. QuickDraw 3D RAVE allows a drawing engine to buffer as many drawing commands as desired. Accordingly, the successful completion of a drawing command (such as `QADrawPoint`) does not guarantee that the specified object is visible on the screen. You can call `QAFlush` to have a drawing engine start processing queued commands. Note, however, that `QAFlush` is not a blocking call—that is, the successful completion of `QAFlush` does not guarantee that all buffered commands have been processed. Calling `QAFlush` guarantees only that all queued commands will eventually be processed.

Typically, you should occasionally call `QAFlush` to update the screen image during a lengthy set of rendering operations in a single-buffered draw context. `QAFlush` has no visible effect when called on a double-buffered draw context, but it does initiate rendering to the back buffer.

The `TQAFlush` function returns a result code (of type `TQAError`) indicating whether any errors have occurred since the previous call to `QARenderStart`. If all rendering commands completed successfully, the value `kQANoErr` is returned. If any other value is returned, you should assume that the rendered image is incorrect.

SPECIAL CONSIDERATIONS

The `QARenderEnd` function automatically calls `QAFlush`.

SEE ALSO

To ensure that all buffered commands have been processed, you can call `QASync` instead of `QAFlush`.

QASync

You can use the `QASync` function to synchronize a draw context.

```
#define QASync(drawContext) (drawContext)->sync (drawContext)
```

`drawContext` A draw context.

DESCRIPTION

The `QASync` function operates just like the `QAFlush` function, except that it waits until all queued drawing commands have been processed before returning. See the description of `QAFlush` (page 1-110) for complete details.

QAGetNoticeMethod

You can use the `QAGetNoticeMethod` function to get the notice method of a draw context.

```
#define QAGetNoticeMethod(drawContext, method, completionCallBack, refCon) \
    (drawContext)->getNoticeMethod (drawContext, method, completionCallBack, refCon)
```

`drawContext` A draw context.

`method` A method selector. See "Notice Method Selectors" (page 1-68) for a description of the available method selectors.

`completionCallBack` On exit, a pointer to the current draw context notice method of the specified type.

`refCon` On exit, the reference constant of the specified notice method.

DESCRIPTION

The `QAGetNoticeMethod` function returns, in the `completionCallBack` parameter, a pointer to the current notice method of the draw context specified by the `drawContext` parameter that has the type specified by the `method` parameter. `QAGetNoticeMethod` also returns, in the `refCon` parameter, the reference constant associated with that notice method.

SEE ALSO

Use `QASetNoticeMethod` (next) to set the notice method for a draw context.

QASetNoticeMethod

You can use the `QASetNoticeMethod` function to set the notice method of a draw context.

```
#define QASetNoticeMethod(drawContext, method, completionCallBack, refCon) \
    (drawContext)->setNoticeMethod (drawContext, method, completionCallBack, refCon)
```

`drawContext` A draw context.

`method` A method selector. See "Notice Method Selectors" (page 1-68) for a description of the available method selectors.

`completionCallBack`
A pointer to the desired draw context notice method of the specified type. See "Notice Methods" (page 1-147) for information about notice methods.

`refCon` A reference constant for the specified notice method. This value is passed unchanged to the notice method when it is called.

DESCRIPTION

The `QASetNoticeMethod` function sets the notice method of type `method` of the draw context specified by the `drawContext` parameter to the function pointed to by the `completionCallBack` parameter. `QASetNoticeMethod` also sets the reference constant of that method to the value specified by the `refCon` parameter.

Registering a Custom Drawing Engine

QuickDraw 3D RAVE provides functions that you can use to register a custom drawing engine and its drawing methods.

QARegisterEngine

You can use the `QARegisterEngine` function to register a custom drawing engine with QuickDraw 3D RAVE.

```
TQAError QARegisterEngine (TQAEngineGetMethod engineGetMethod);
```

`engineGetMethod`
The method retrieval method of your drawing engine. See "Method Reporting Methods" (page 1-146) for a complete description of this method.

DESCRIPTION

The `QARegisterEngine` function registers your custom drawing engine with QuickDraw 3D RAVE. You should call this function at startup time (usually from the initialization routine in the shared library containing the code for your drawing engine). QuickDraw 3D RAVE uses the method specified by the `engineGetMethod` parameter to retrieve function pointers for the non-drawing methods defined in your drawing engine.

SPECIAL CONSIDERATIONS

You should call `QARegisterEngine` only to register a custom drawing engine. Applications using QuickDraw 3D RAVE to draw into one or more draw contexts do not need to use this function.

QARegisterDrawMethod

You can use the `QARegisterDrawMethod` function to register a public draw context method with QuickDraw 3D RAVE.

```
TQAError QARegisterDrawMethod (
                    TQADrawContext *drawContext,
                    TQADrawMethodTag methodTag,
                    TQADrawMethod method);
```

`drawContext` A draw context.

`methodTag` A selector that determines which draw context method is to be registered for the specified draw context. See “Public Draw Context Method Selectors” (page 1-67) for complete information about the available method selectors.

`method` A pointer to the draw context method of the specified type.

DESCRIPTION

The `QARegisterDrawMethod` function changes the method pointer of the draw context specified by the `drawContext` parameter that has the type specified by the `methodTag` parameter to the method specified by the `method` parameter. You

should call `QARegisterDrawMethod` instead of directly changing the fields of a draw context structure.

Application-Defined Routines

This section describes the routines you might need to define to add a new drawing engine to the QuickDraw 3D Acceleration Layer.

Note
See "Writing a Drawing Engine," beginning on page 1-23 for step-by-step details on writing a drawing engine. ◆

This section also describes the notice method you might need to define to have your application receive notices when certain events occur. See "Notice Methods" (page 1-147) for details.

Public Draw Context Methods

To write a drawing engine, you need to implement a number of drawing methods, pointers to which are contained in a draw context structure (of type `TQADrawContext`). These functions are called whenever an application uses one of the drawing macros described earlier (in "Manipulating Draw Contexts," beginning on page 1-94). For example, when an application uses the `QADrawPoint` macro to draw a point in a draw context linked to your drawing engine, your engine's `TQADrawPoint` method is called.

A draw context structure is passed as the first parameter to all these draw context methods. This allows you to retrieve your draw context's private data, which is pointed to by the first field of that structure.

IMPORTANT
Most of the draw context methods declare the draw context structure passed to them as `const`, in which case you should not alter any fields of that structure. Only three methods are allowed to change fields of the draw context structure: `TQASetFloat`, `TQASetInt`, and `TQASetPtr`. Failure to heed the `const` declaration may cause any code calling your engine (including QuickDraw 3D) to fail. ▲

Pointers to your drawing engine's public draw context methods are assigned to the fields of a draw context structure by your `TQADrawPrivateNew` method. See page 1-136 for details.

TQAGetFloat

A drawing engine must define a method to get a floating-point value of a draw context state variable.

```
typedef float (*TQAGetFloat) (
                    const TQADrawContext *drawContext,
                    TQATagFloat tag);
```

`drawContext` A draw context.

`tag` A state variable tag.

DESCRIPTION

Your `TQAGetFloat` function should return, as its function result, the floating-point value of the draw context state variable specified by the `drawContext` and `tag` parameters. If you do not recognize or support the specified tag, your `TQAGetFloat` function should return the value 0.

TQASetFloat

A drawing engine must define a method to set a floating-point value for a draw context state variable.

```
typedef void (*TQASetFloat) (
                    TQADrawContext *drawContext,
                    TQATagFloat tag,
                    float newValue);
```

`drawContext` A draw context.

`tag` A state variable tag.

`newValue` The new value of the specified state variable.

DESCRIPTION

Your `TQASetFloat` function should set the value of the draw context state variable specified by the `drawContext` and `tag` parameters to the floating-point value specified by the `newValue` parameter.

Your drawing engine must accept all possible values for the `tag` parameter. If you encounter a value in the `tag` parameter that you cannot recognize, you should do nothing. Similarly, you should do nothing if the `tag` parameter specifies a state variable for optional features your drawing engine does not support.

SPECIAL CONSIDERATIONS

If your `TQASetFloat` function needs to change one or more of the function pointers in the specified draw context, it must call the `QARegisterDrawMethod` function to do so. It should not directly change the fields of a draw context.

TQAGetInt

A drawing engine must define a method to get a long integer value of a draw context state variable.

```
typedef unsigned long (*TQAGetInt) (
                    const TQADrawContext *drawContext,
                    TQATagInt tag);
```

`drawContext` A draw context.

`tag` A state variable tag.

DESCRIPTION

Your `TQAGetInt` function should return, as its function result, the long integer value of the draw context state variable specified by the `drawContext` and `tag` parameters. If you do not recognize or support the specified tag, your `TQAGetInt` function should return the value 0.

TQASetInt

A drawing engine must define a method to set a long integer value for a draw context state variable.

```
typedef void (*TQASetInt) (
                    TQADrawContext *drawContext,
                    TQATagInt tag,
                    unsigned long newValue);
```

`drawContext` A draw context.

`tag` A state variable tag.

`newValue` The new value of the specified state variable.

DESCRIPTION

Your `TQASetInt` function should set the value of the draw context state variable specified by the `drawContext` and `tag` parameters to the long integer value specified by the `newValue` parameter.

Your drawing engine must accept all possible values for the `tag` parameter. If you encounter a value in the `tag` parameter that you cannot recognize, you should do nothing. Similarly, you should do nothing if the `tag` parameter specifies a state variable for optional features your drawing engine does not support.

SPECIAL CONSIDERATIONS

If your `TQASetInt` function needs to change one or more of the function pointers in the specified draw context, it must call the `QARegisterDrawMethod` function to do so. It should not directly change the fields of a draw context.

TQAGetPtr

A drawing engine must define a method to get a pointer value of a draw context state variable.

```
typedef void *(*TQAGetPtr) (
                    const TQADrawContext *drawContext,
                    TQATagPtr tag);
```

`drawContext` A draw context.

`tag` A state variable tag.

DESCRIPTION

Your `TQAGetPtr` function should return, as its function result, the pointer value of the draw context state variable specified by the `drawContext` and `tag` parameters. If you do not recognize or support the specified tag, your `TQAGetPtr` function should return the value 0.

TQASetPtr

A drawing engine must define a method to set a pointer value for a draw context state variable.

```
typedef void (*TQASetPtr) (
                    TQADrawContext *drawContext,
                    TQATagPtr tag,
                    const void *newValue);
```

`drawContext` A draw context.

`tag` A state variable tag.

`newValue` The new value of the specified state variable.

DESCRIPTION

Your `TQASetPtr` function should set the value of the draw context state variable specified by the `drawContext` and `tag` parameters to the pointer value specified by the `newValue` parameter.

Your drawing engine must accept all possible values for the `tag` parameter. If you encounter a value in the `tag` parameter that you cannot recognize, you should do nothing. Similarly, you should do nothing if the `tag` parameter specifies a state variable for optional features your drawing engine does not support.

SPECIAL CONSIDERATIONS

If your `TQASetPtr` function needs to change one or more of the function pointers in the specified draw context, it must call the `QARegisterDrawMethod` function to do so. It should not directly change the fields of a draw context.

TQADrawPoint

A drawing engine must define a method to draw a point.

```
typedef void (*TQADrawPoint) (
                    const TQADrawContext *drawContext,
                    const TQAVGouraud *v);
```

`drawContext` A draw context.

`v` A Gouraud vertex.

DESCRIPTION

Your `TQADrawPoint` function should draw the single point specified by the `v` parameter to the draw context specified by the `drawContext` parameter. The size of the point is determined by the `kQATag_Width` state variable of the draw context.

TQADrawLine

A drawing engine must define a method to draw a line between two points.

```
typedef void (*TQADrawLine) (
                    const TQADrawContext *drawContext,
                    const TQAVGouraud *v0,
                    const TQAVGouraud *v1);
```

`drawContext` A draw context.

`v0` A Gouraud vertex.

`v1` A Gouraud vertex.

DESCRIPTION

Your `TQADrawLine` function should draw the line specified by the `v0` and `v1` parameters to the draw context specified by the `drawContext` parameter. The size of the line is determined by the `kQATag_Width` state variable of the draw context. If the specified vertices have different colors, the line color is interpolated smoothly between the two vertex colors.

TQADrawTriGouraud

A drawing engine must define a method to draw Gouraud-shaded triangles.

```
typedef void (*TQADrawTriGouraud) (
                    const TQADrawContext *drawContext,
                    const TQAVGouraud *v0,
                    const TQAVGouraud *v1,
                    const TQAVGouraud *v2,
                    unsigned long flags);
```

`drawContext` A draw context.

`v0` A Gouraud vertex.

`v1` A Gouraud vertex.

`v2` A Gouraud vertex.

`flags` A set of triangle flags. See "Triangle Flags Masks," beginning on page 1-63 for a complete description of the available flags.

DESCRIPTION

Your `TQADrawTriGouraud` function should draw the Gouraud-shaded triangle determined by the three points specified by the `v0`, `v1`, and `v2` parameters into the draw context specified by the `drawContext` parameter. Features of the triangle are determined by the `flags` parameter. Currently, this parameter is used to specify an orientation for the triangle.

TQADrawTriTexture

A drawing engine may define a method to draw texture-mapped triangles. This method is optional and must be supported only by drawing engines that support texture mapping.

```
typedef void (*TQADrawTriTexture) (
                    const TQADrawContext *drawContext,
                    const TQAVTexture *v0,
                    const TQAVTexture *v1,
                    const TQAVTexture *v2,
                    unsigned long flags);
```

`drawContext` A draw context.

`v0` A texture vertex.

`v1` A texture vertex.

`v2` A texture vertex.

`flags` A set of triangle flags. See "Triangle Flags Masks," beginning on page 1-63 for a complete description of the available flags.

DESCRIPTION

Your `TQADrawTriTexture` function should draw the texture-mapped triangle determined by the three points specified by the `v0`, `v1`, and `v2` parameters into the draw context specified by the `drawContext` parameter. The texture used for

the mapping is determined by the value of the `kQATag_Texture` state variable. Features of the triangle are determined by the `flags` parameter. Currently, this parameter is used to specify an orientation for the triangle.

TQASubmitVerticesGouraud

A drawing engine may define a method to submit Gouraud vertices.

```
typedef void (*TQASubmitVerticesGouraud) (
                    const TQADrawContext *drawContext,
                    unsigned long nVertices,
                    const TQAVGouraud *vertices);
```

`drawContext` A draw context.

`nVertices` The number of Gouraud vertices pointed to by the `vertices` parameter.

`vertices` A pointer to an array of Gouraud vertices.

DESCRIPTION

Your `TQASubmitVerticesGouraud` function should prepare to render a Gouraud-shaded triangular mesh in the draw context specified by the `drawContext` parameter using the vertices pointed to by the `vertices` parameter. The actual triangulation and drawing of the mesh does not occur until an application calls the `QADrawTriMeshGouraud` function.

The calling application is responsible for managing the memory occupied by the Gouraud vertices. Your `TQASubmitVerticesGouraud` function should not copy the vertex data pointed to by the `vertices` parameter.

SPECIAL CONSIDERATIONS

The `TQASubmitVerticesGouraud` method is optional. If your drawing engine does not support triangle meshes, QuickDraw 3D RAVE decomposes a triangle mesh into individual triangles when the user calls the `QASubmitVerticesGouraud` function to submit a triangle mesh.

There is no QuickDraw 3D RAVE function that an application can use to unsubmit a triangle mesh. Your drawing engine must manage memory in some appropriate manner.

TQASubmitVerticesTexture

A drawing engine may define a method to submit texture vertices. This method is optional and must be supported only by drawing engines that support texture mapping.

```
typedef void (*TQASubmitVerticesTexture) (
                    const TQADrawContext *drawContext,
                    unsigned long nVertices,
                    const TQAVTexture *vertices);
```

`drawContext` A draw context.

`nVertices` The number of texture vertices pointed to by the `vertices` parameter.

`vertices` A pointer to an array of texture vertices.

DESCRIPTION

Your `TQASubmitVerticesTexture` function should prepare to render a texture-mapped triangular mesh in the draw context specified by the `drawContext` parameter using the vertices pointed to by the `vertices` parameter. The actual triangulation and drawing of the mesh does not occur until an application calls the `QADrawTriMeshTexture` function.

The calling application is responsible for managing the memory occupied by the texture vertices. Your `TQASubmitVerticesTexture` function should not copy the vertex data pointed to by the `vertices` parameter.

SPECIAL CONSIDERATIONS

The `TQASubmitVerticesTexture` method is optional. If your drawing engine does not support triangle meshes, QuickDraw 3D RAVE decomposes a triangle mesh into individual triangles when the user calls the `QASubmitVerticesTexture` function to submit a triangle mesh.

There is no QuickDraw 3D RAVE function that an application can use to unsubmit a triangle mesh. Your drawing engine must manage memory in some appropriate manner.

TQADrawTriMeshGouraud

A drawing engine may define a method to draw a triangle mesh with Gouraud shading.

```
typedef void (*TQADrawTriMeshGouraud) (
                    const TQADrawContext *drawContext,
                    unsigned long nTriangles,
                    const TQAIndexedTriangle *triangles);
```

`drawContext`	A draw context.
`nTriangle`	The number of indexed triangles pointed to by the `triangles` parameter.
`triangles`	A pointer to an array of indexed triangles. See “Indexed Triangle Structure” (page 1-80) for a description of indexed triangles.

DESCRIPTION

Your `TQADrawTriMeshGouraud` function should draw, with Gouraud shading, the triangle mesh specified by the `triangles` parameter into the draw context specified by the `drawContext` parameter. Each triangle in the mesh is defined by a `TQAIndexedTriangle` data structure, which contains three indices into the array of Gouraud vertices previously submitted to the draw context by a call to the `QASubmitVerticesGouraud` function.

SPECIAL CONSIDERATIONS

The `TQADrawTriMeshGouraud` method is optional. If your drawing engine does not support triangle meshes, QuickDraw 3D RAVE decomposes a triangle mesh into individual triangles when the user calls the `QASubmitVerticesGouraud` function to submit a triangle mesh.

TQADrawTriMeshTexture

A drawing engine may define a method to draw a texture-mapped triangle mesh. This method is optional and must be supported only by drawing engines that support texture mapping.

```
typedef void (*TQADrawTriMeshTexture) (
                    const TQADrawContext *drawContext,
                    unsigned long nTriangles,
                    const TQAIndexedTriangle *triangles);
```

`drawContext` A draw context.

`nTriangle` The number of indexed triangles pointed to by the `triangles` parameter.

`triangles` A pointer to an array of indexed triangles. See “Indexed Triangle Structure” (page 1-80) for a description of indexed triangles.

DESCRIPTION

Your `TQADrawTriMeshTexture` function should draw the texture-mapped triangle mesh specified by the `triangles` parameter into the draw context specified by the `drawContext` parameter. Each triangle in the mesh is defined by a `TQAIndexedTriangle` data structure, which contains three indices into the array of texture vertices previously submitted to the draw context by a call to the `QASubmitVerticesTexture` function.

SPECIAL CONSIDERATIONS

The `TQADrawTriMeshTexture` method is optional. If your drawing engine does not support triangle meshes, QuickDraw 3D RAVE decomposes a triangle mesh into individual triangles when the user calls the `QASubmitVerticesTexture` function to submit a triangle mesh.

TQADrawVGouraud

A drawing engine may define a method to draw Gouraud-shaded objects defined by vertices. This method is optional and must be supported only by drawing engines that do not want calls to `QADrawVGouraud` to be replaced by calls to the `QADrawPoint`, `QADrawLine`, or `QADrawTriGouraud` methods.

```
typedef void (*TQADrawVGouraud) (
                    const TQADrawContext *drawContext,
                    unsigned long nVertices,
                    TQAVertexMode vertexMode,
                    const TQAVGouraud vertices[],
                    const unsigned long flags[]);
```

`drawContext` A draw context.

`nVertices` The number of vertices contained in the `vertices` array.

`vertexMode` A vertex mode. See “Vertex Modes” (page 1-56) for a description of the available vertex modes.

`vertices` An array of Gouraud vertices.

`flags` An array of triangle flags, or the value `NULL`. See “Triangle Flags Masks” (page 1-63) for a description of the available triangle flags. This parameter is valid only if the `vertexMode` parameter contains the value `kQAVertexMode_Tri`, `kQAVertexMode_Strip`, or `kQAVertexMode_Fan`.

DESCRIPTION

Your `TQADrawVGouraud` function should draw the vertices in the array specified by the `vertices` parameter into the draw context specified by the `drawContext` parameter, according to the vertex modes flag specified by the `vertexMode` parameter. For instance, if the value of the `vertexMode` parameter is `kQAVertexMode_Polyline`, then the vertices in that array are interpreted as defining a polyline (a set of connected line segments). Gouraud shading should be applied to whatever objects are drawn.

TQADrawVTexture

A drawing engine may define a method to draw texture-mapped objects defined by vertices. This method is optional and must be supported only by drawing engines that support texture mapping and do not want calls to `QADrawVTexture` to be replaced by calls to the `QADrawPoint`, `QADrawLine`, or `QADrawTriTexture` methods.

```
typedef void (*TQADrawVTexture) (
                    const TQADrawContext *drawContext,
                    unsigned long nVertices,
                    TQAVertexMode vertexMode,
                    const TQAVTexture vertices[],
                    const unsigned long flags[]);
```

`drawContext` A draw context.

`nVertices` The number of vertices contained in the `vertices` array.

`vertexMode` A vertex mode. See “Vertex Modes” (page 1-56) for a description of the available vertex modes.

`vertices` An array of texture vertices.

`flags` An array of triangle flags, or the value `NULL`. See “Triangle Flags Masks” (page 1-63) for a description of the available triangle flags. This parameter is valid only if the `vertexMode` parameter contains the value `kQAVertexMode_Tri`, `kQAVertexMode_Strip`, or `kQAVertexMode_Fan`.

DESCRIPTION

Your `TQADrawVTexture` function should draw the vertices in the array specified by the `vertices` parameter into the draw context specified by the `drawContext` parameter, according to the vertex modes flag specified by the `vertexMode` parameter. For instance, if the value of the `vertexMode` parameter is `kQAVertexMode_Polyline`, then the vertices in that array are interpreted as defining a polyline (a set of connected line segments). Texture mapping (using the texture determined by the value of the `kQATag_Texture` state variable) should be applied to whatever objects are drawn.

IMPORTANT

The vertex modes `kQAVertexMode_Point` and `kQAVertexMode_Line` are supported only by drawing engines that support the `kQAOptional_OpenGL` feature. All other drawing engines should ignore requests to texture map points or lines. ▲

TQADrawBitmap

A drawing engine must define a method to draw bitmaps into a draw context.

```
typedef void (*TQADrawBitmap) (
                    const TQADrawContext *drawContext,
                    const TQAVGouraud *v,
                    TQABitmap *bitmap);
```

`drawContext` A draw context.

`v` A Gouraud vertex.

`bitmap` A pointer to a bitmap (returned by a previous call to `QABitmapNew`).

DESCRIPTION

Your `TQADrawBitmap` function should draw the bitmap specified by the `bitmap` parameter into the draw context specified by the `drawContext` parameter, with the upper-left corner of the bitmap located at the point specified by the `v` parameter. The `v` parameter can contain negative values in its `x` or `y` fields, so you can position upper-left corner of the bitmap outside the draw context rectangle. This allows you to move the bitmap smoothly off any edge of the draw context.

TQARenderStart

A drawing engine must define a method to initialize a draw context before the engine performs any rendering into that context.

```
typedef void (*TQARenderStart) (
                    const TQADrawContext *drawContext,
                    const TQARect *dirtyRect,
                    const TQADrawContext *initialContext);
```

`drawContext` A draw context.

`dirtyRect` The minimum area of the specified draw context to clear, or the value `NULL`.

`initialContext`
A previously cached draw context, or the value `NULL`.

DESCRIPTION

Your `TQARenderStart` function should perform any operations necessary to initialize the draw context specified by the `drawContext` parameter. This includes clearing the z buffer and the color buffers of the draw context. If the value of the `initialContext` parameter is `NULL`, then your `TQARenderStart` function should clear the z buffer to 1.0 and set the color buffers to the values of the `kQATag_ColorBG_a`, `kQATag_ColorBG_r`, `kQATag_ColorBG_g`, and `kQATag_ColorBG_b` draw context state variables. If, however, the value of the `initialContext` parameter is not `NULL`, then your `TQARenderStart` function should use the previously cached draw context specified by that parameter to initialize the draw context specified by the `drawContext` parameter.

The `dirtyRect` parameter indicates the minimum area of the specified draw context that should be cleared on initialization. If the value of the `dirtyRect` parameter is `NULL`, the entire draw context is cleared. If the value of the `dirtyRect` parameter is not `NULL`, it indicates the rectangle in the draw context to clear. Some drawing engines may exhibit improved performance when an area that is smaller than the entire draw context rectangle is passed. However, the interpretation of the `dirtyRect` parameter is dependent on the drawing engine, which may choose to initialize the entire draw context. As a result, code calling your `QARenderStart` function should not use this parameter as a means to avoid clearing all of a draw context or to perform incremental rendering. Instead, that code should use the `initialContext` parameter to achieve such effects.

SPECIAL CONSIDERATIONS

Applications should call `QARenderStart` before performing any rendering operations in the specified draw context, and they should call either `QARenderEnd` to signal the end of rendering operations or `QARenderAbort` to cancel rendering operations. However, when a drawing engine is performing OpenGL rendering, the `QARenderStart` function operates just like the OpenGL function `glClear`. In OpenGL mode, it is not necessary that a call to `QARenderStart` always be balanced by a matching call to `QARenderEnd`, and drawing commands may occur at any time.

TQARenderEnd

A drawing engine must define a method to signal the end of any rendering into a draw context.

```
typedef TQAError (*TQARenderEnd) (
                    const TQADrawContext *drawContext,
                    const TQARect *modifiedRect);
```

`drawContext` A draw context.

`modifiedRect` The minimum area of the back buffer of the specified draw context to display, or the value `NULL`.

DESCRIPTION

Your `TQARenderEnd` function should perform any operations necessary to display an image rendered into the draw context specified by the `drawContext` parameter. If the draw context is double buffered, your function should display the back buffer. If the draw context is single buffered, your function should call `QAFlush`. In either case, your drawing engine should unlock any frame buffers or other memory that is locked, remove any cursor shields, and so forth.

The `modifiedRect` parameter indicates the minimum area of the back buffer of the specified draw context that should be displayed. If the value of the `modifiedRect` parameter is `NULL`, the entire back buffer is displayed. If the value of the `modifiedRect` parameter is not `NULL`, it indicates the rectangle in the back buffer to display. Some drawing engines may exhibit improved performance when an area that is smaller than the entire draw context rectangle is passed (to

avoid unnecessary pixel copying). However, the interpretation of the `modifiedRect` parameter is dependent on the drawing engine, which may choose to draw the entire back buffer.

Your `TQARenderEnd` function should return a result code (of type `TQAError`) indicating whether any errors have occurred since the previous call to your `TQARenderStart` function. If all rendering commands completed successfully, you should return the value `kQANoErr`. If you return any other value, the code that called `QARenderEnd` should assume that the rendered image is incorrect.

SPECIAL CONSIDERATIONS

Applications should call `QARenderStart` before performing any rendering operations in the specified draw context, and they should call either `QARenderEnd` to signal the end of rendering operations or `QARenderAbort` to cancel rendering operations. Once an application has called `QARenderEnd`, it should not submit any drawing requests until it has called `QARenderStart` again.

TQARenderAbort

A drawing engine must define a method to cancel any asynchronous drawing requests for a draw context.

```
typedef TQAError (*TQARenderAbort) (
                    const TQADrawContext *drawContext);
```

`drawContext` A draw context.

DESCRIPTION

Your `TQARenderAbort` function should immediately stop processing any asynchronous drawing command it is currently processing and it should discard any queued commands associated with the draw context specified by the `drawContext` parameter.

Your `TQARenderAbort` function should return a result code (of type `TQAError`) indicating whether any errors have occurred since the previous call to your `TQARenderStart` function. If all rendering commands completed successfully,

you should return the value `kQANoErr`. If you return any other value, the code that called `QARenderEnd` should assume that the rendered image is incorrect.

TQAFlush

A drawing engine must define a method to flush a draw context.

```
typedef TQAError (*TQAFlush) (const TQADrawContext *drawContext);
```

`drawContext` A draw context.

DESCRIPTION

Your `TQAFlush` function should cause your drawing engine to begin rendering all drawing commands that are queued in a buffer awaiting processing for the draw context specified by the `drawContext` parameter. QuickDraw 3D RAVE allows a drawing engine to buffer as many drawing commands as desired. Accordingly, the successful completion of a drawing command (such as `QADrawPoint`) does not guarantee that the specified object is visible on the screen. An application can call `QAFlush` to have your drawing engine start processing queued commands. Note, however, that `QAFlush` is not a blocking call—that is, the successful completion of `QAFlush` does not guarantee that all buffered commands have been processed. Calling `QAFlush` guarantees only that all queued commands will eventually be processed.

Typically, applications should occasionally call `QAFlush` to update the screen image during a lengthy set of rendering operations in a single-buffered draw context. `QAFlush` has no visible effect when called on a double-buffered draw context, but it does initiate rendering to the back buffer.

Your `TQAFlush` function should return a result code (of type `TQAError`) indicating whether any errors have occurred since the previous call to your `TQARenderStart` function. If all rendering commands completed successfully, you should return the value `kQANoErr`. If you return any other value, the code that called `QAFlush` should assume that the rendered image is incorrect.

TQASync

A drawing engine must define a method to synchronize a draw context.

```
typedef TQAError (*TQASync) (const TQADrawContext *drawContext);
```

`drawContext` A draw context.

DESCRIPTION

Your `TQASync` function should operate just like your `TQAFlush` function, except that it should wait until all queued drawing commands have been processed before returning. See the description of `TQAFlush` (page 1-133) for complete details.

TQAGetNoticeMethod

A drawing engine must define a method to return the notice method of a draw context.

```
typedef TQAError (*TQAGetNoticeMethod) (
                    const TQADrawContext *drawContext,
                    TQAMethodSelector method,
                    TQANoticeMethod *completionCallBack,
                    void **refCon);
```

`drawContext` A draw context.

`method` A method selector. See “Notice Method Selectors” (page 1-68) for a description of the available method selectors.

`completionCallBack`
On exit, a pointer to the current draw context notice method of the specified type.

`refCon` On exit, the reference constant of the specified notice method.

DESCRIPTION

Your `TQAGetNoticeMethod` function should return, in the `completionCallBack` parameter, a pointer to the current notice method of the draw context specified by the `drawContext` parameter that has the type specified by the `method` parameter. `TQAGetNoticeMethod` should also return, in the `refCon` parameter, the reference constant associated with that notice method.

TQASetNoticeMethod

A drawing engine must define a method to set the notice method of a draw context.

```
typedef TQAError (*TQASetNoticeMethod) (
                    const TQADrawContext *drawContext,
                    TQAMethodSelector method,
                    TQANoticeMethod completionCallBack,
                    void *refCon);
```

`drawContext` A draw context.

`method` A method selector. See "Notice Method Selectors" (page 1-68) for a description of the available method selectors.

`completionCallBack`
A pointer to the desired draw context notice method of the specified type. See "Notice Methods" (page 1-147) for information about notice methods.

`refCon` A reference constant for the specified notice method. This value is passed unchanged to the notice method when it is called.

DESCRIPTION

Your `TQASetNoticeMethod` function should set the notice method of type `method` of the draw context specified by the `drawContext` parameter to the function pointed to by the `completionCallBack` parameter. `TQASetNoticeMethod` should also set the reference constant of that method to the value specified by the `refCon` parameter.

Private Draw Context Methods

To write a drawing engine, you need to implement several private methods for managing draw contexts.

Pointers to your drawing engine's private draw context methods are returned to QuickDraw 3D RAVE by your `TQAEngineGetMethod` method. See page 1-147 for details.

TQADrawPrivateNew

A drawing engine must define a method to create its own private data and initialize a new draw context.

```
typedef TQAError (*TQADrawPrivateNew) (
                    TQADrawContext *newDrawContext,
                    const TQADevice *device,
                    const TQARect *rect,
                    const TQAClip *clip,
                    unsigned long flags);
```

`newDrawContext`
The draw context to initialize. On entry, all the fields of this structure have the value `NULL`.

`device` A device.

`rect` The rectangular region (specified in device coordinates) of the specified device that can be drawn into by the drawing engine associated with the new draw context.

`clip` The two-dimensional clipping region for the new draw context, or `NULL` if no clipping is desired. This parameter must be set to `NULL` for devices of type `kQADeviceMemory`.

`flags` A set of bit flags specifying features of the new draw context. See "Draw Context Flags Masks" (page 1-65) for complete information.

DESCRIPTION

Your `TQADrawPrivateNew` function is called whenever an application calls `QADrawContextNew` to create a new draw context associated with your drawing engine. Your function should perform any initialization required for the new draw context. In particular, it should return a pointer to the draw context's private data in the `drawPrivate` field of the draw context structure pointed to by the `newDrawContext` parameter. In addition, your `TQADrawPrivateNew` function should set any other fields of that draw context structure to point to public draw context methods defined by the drawing engine.

Because it is the responsibility of your `TQADrawPrivateNew` function to initialize the fields of a draw context structure, you can load different methods depending on the features of the device or draw context specified by the `device` and `flags` parameters. For instance, you might load one line drawing function for a device that displays 16 bits per pixel and a different line drawing function for a device that displays 32 bits per pixel. This technique allows you to avoid testing the display depth each time you draw a line.

SEE ALSO

See Listing 1-8 (page 1-26) for a sample `TQADrawPrivateNew` function.

TQADrawPrivateDelete

A drawing engine must define a method to delete its private data.

```
typedef void (*TQADrawPrivateDelete) (TQADrawPrivate *drawPrivate);
```

`drawPrivate` The draw context's private data.

DESCRIPTION

Your `TQADrawPrivateDelete` function is called whenever an application calls `QADrawContextDelete`. Your function should release any memory or other resources that were allocated by your `TQADrawPrivateNew` function.

TQAEngineCheckDevice

A drawing engine must define a method to indicate whether the engine can draw to a particular device.

```
typedef TQAError (*TQAEngineCheckDevice) (const TQADevice *device);
```

`device` A device.

DESCRIPTION

Your `TQAEngineCheckDevice` function should return, as its function result, a code that indicates whether your drawing engine can draw into the device specified by the `device` parameter (`kQANoErr`) or not (`kQAError`).

TQAEngineGestalt

A drawing engine must define a method to return information about its capabilities.

```
typedef TQAError (*TQAEngineGestalt) (
                    TQAGestaltSelector selector,
                    void *response);
```

`selector` A selector that determines what kind of information is to be returned about your drawing engine. See “Gestalt Selectors” (page 1-57) for complete information about the available selectors and the information you should return.

`response` A pointer to a buffer into which the returned information is to be copied. The calling application is responsible for allocating this buffer. The size and meaning of the data to be copied depends on the selector passed in the `selector` parameter.

DESCRIPTION

Your `TQAEngineGestalt` function is called whenever an application calls `QAEngineGestalt`. Your function should return, in the buffer pointed to by the

`response` parameter, information about features of the type specified by the `selector` parameter.

Color Lookup Table Methods

To write a drawing engine, you might need to implement several private methods for creating and disposing of color lookup tables. Pointers to your drawing engine's color lookup table methods are returned to QuickDraw 3D RAVE by your `TQAEngineGetMethod` method. See page 1-147 for details.

TQAColorTableNew

A drawing engine may define a method to create a new color lookup table. This method is optional and must be supported only by drawing engines that support color lookup tables.

```
typedef TQAError (*TQAColorTableNew) (
                    TQAColorTableType pixelType,
                    void *pixelData,
                    long transparentIndex,
                    TQAColorTable **newTable);
```

`pixelType` The type of the new color lookup table. See "Color Lookup Table Types" (page 1-37) for information on the available color lookup table types.

`pixelData` A pointer to the color lookup table entries.

`transparentIndexFlag`
A long integer, interpreted as a Boolean value, that indicates whether the color lookup table entry at index 0 is completely transparent (`TRUE`) or not (`FALSE`).

`newTable` On entry, the address of a pointer variable. On exit, set that variable to point to a new color lookup table. If a new color lookup table cannot be created, set `*newTable` to the value `NULL`.

DESCRIPTION

Your `TQAColorTableNew` function is called whenever an application calls `QAColorTableNew`. Your function should return, in the buffer pointed to by the `newTable` parameter, a pointer to a new color lookup table of the type specified by the `pixelType` parameter. The color table data is passed to your function in the `pixelData` parameter. Your method should copy that data so that the caller can dispose of the memory it occupies.

IMPORTANT
Currently, QuickDraw 3D RAVE supports only 32-bit RGB color lookup table entries. Your drawing engine might reduce the size of individual color lookup table entries to fit into its on-board memory. ▲

SPECIAL CONSIDERATIONS

Not all drawing engines need to support color lookup tables, but QuickDraw 3D RAVE does not provide color lookup table emulation for engines that do not support them.

TQAColorTableDelete

A drawing engine may define a method to dispose of color lookup table. This method is optional and must be supported only by drawing engines that support color lookup tables.

```
typedef void (*TQAColorTableDelete) (TQAColorTable *colorTable);
```

`colorTable` A color lookup table.

DESCRIPTION

Your `TQAColorTableDelete` function is called whenever an application calls `QAColorTableDelete`. Your function should delete the color lookup table specified by the `colorTable` parameter. Any memory and other resources associated with that color lookup table should be released.

Texture and Bitmap Methods

To write a drawing engine, you need to implement several private methods for managing bitmaps. If your engine supports texture mapping, you also need to implement several private methods for managing textures.

Pointers to your drawing engine's texture and bitmap methods are returned to QuickDraw 3D RAVE by your `TQAEngineGetMethod` method. See page 1-147 for details.

TQATextureNew

A drawing engine may define a method to create a new texture map. This method is optional and must be supported only by drawing engines that support texture mapping.

```
typedef TQAError (*TQATextureNew) (
                    unsigned long flags,
                    TQAImagePixelType pixelType,
                    const TQAImage images[],
                    TQATexture **newTexture);
```

`flags` A set of bit flags specifying features of the new texture map. See "Texture Flags Masks" (page 1-64) for complete information.

`pixelType` The type of pixels in the new texture map. See "Pixel Types" (page 1-35) for a description of the values you can pass in this parameter.

`images` An array of pixel images to use for the new texture map. The values in the `width` and `height` fields of these structures must be an even power of 2.

`newTexture` On entry, the address of a pointer variable. On exit, that variable points to a new texture map. If a new texture map cannot be created, `*newTexture` is set to the value `NULL`.

DESCRIPTION

Your `TQATextureNew` function is called whenever an application calls `QATextureNew`. Your function should perform any tasks required to use the

texture in texture-mapping operations. This might involve loading the texture into memory on the device associated with your drawing engine. If so, your `TQATextureNew` function should not return until the texture has been completely loaded.

The `flags` parameter specifies a set of texture map features. If the `kQATexture_Lock` bit in that parameter is set but your drawing engine cannot guarantee that the texture will remain locked in memory, your `TQATextureNew` function should return an error.

If the `kQATexture_Mipmap` bit of the `flags` parameter is clear, the `images` parameter points to a single pixel image that defines the texture map. If the `kQATexture_Mipmap` bit is set, the `images` parameter points to an array of pixel images of varying pixel depths. The first element in the array must be the mipmap page having the highest resolution, with a width and height that are even powers of 2. Each subsequent pixel image in the array should have a width and height that are half those of the previous image (with a minimum width and height of 1).

TQATextureDetach

A drawing engine may define a method to detach a texture map. This method is optional and must be supported only by drawing engines that support texture mapping.

```
typedef TQAError (*TQATextureDetach) (TQATexture *texture);
```

`texture` A texture map.

DESCRIPTION

Your `TQATextureDetach` function is called whenever an application calls `QATextureDetach`. Your function should, if necessary, load the texture specified by the `texture` parameter into memory on the device associated with your drawing engine (so that the caller can release the memory occupied by the texture). Your `TQATextureDetach` function should not return until the texture has been completely loaded.

TQATextureBindColorTable

A drawing engine may define a method to bind a color lookup table to a texture map.

```
typedef TQAError (*TQATextureBindColorTable) (
                    TQATexture *texture,
                    TQAColorTable *colorTable);
```

`texture` A texture map.

`colorTable` A color lookup table (as returned by a previous call to `QAColorTableNew`).

DESCRIPTION

Your `TQATextureBindColorTable` function is called whenever an application calls `QATextureBindColorTable`. Your function should bind the color lookup table specified by the `colorTable` parameter to the texture map specified by the `texture` parameter. Note that the type of the specified color lookup table must match that of the pixel type of the texture map to which it is bound. For example, a color lookup table of type `kQAColorTable_CL8_RGB32` can be bound only to a texture map whose pixel type is `kQAPixel_CL8`.

TQATextureDelete

A drawing engine may define a method to delete a texture map. This method is optional and must be supported only by drawing engines that support texture mapping.

```
typedef void (*TQATextureDelete) (TQATexture *texture);
```

`texture` A texture map.

DESCRIPTION

Your `TQATextureDelete` function is called whenever an application calls `QATextureDelete`. Your function should delete the texture map specified by the `texture` parameter.

TQABitmapNew

A drawing engine must define a method to create a new bitmap.

```
typedef TQAError (*TQABitmapNew) (
                    unsigned long flags,
                    TQAImagePixelType pixelType,
                    const TQAImage *image,
                    TQABitmap **newBitmap);
```

`flags` A set of bit flags specifying features of the new bitmap. See "Bitmap Flags Masks" (page 1-64) for complete information

`pixelType` The type of pixels in the new bitmap. See "Pixel Types" (page 1-35) for a description of the values you can pass in this parameter.

`image` A pixel image to use for the new bitmap. The `width` and `height` fields of this image can have any values greater than 0.

`newBitmap` On entry, the address of a pointer variable. On exit, that variable points to a new bitmap. If a new bitmap cannot be created, `*newBitmap` is set to the value `NULL`.

DESCRIPTION

Your `TQABitmapNew` function is called whenever an application calls `QABitmapNew`. Your function should perform any tasks required to draw the bitmap in the draw context associated with your drawing engine. This might involve loading the bitmap into memory on the device associated with your drawing engine. If so, your `TQABitmapNew` function should not return until the bitmap has been completely loaded.

The `flags` parameter specifies a set of bitmap features. If the `kQABitmap_Lock` bit in that parameter is set but your drawing engine cannot guarantee that the

bitmap will remain locked in memory, your `TQABitmapNew` function should return an error.

TQABitmapDetach

A drawing engine must define a method to detach a bitmap from a drawing engine.

```
typedef TQAError (*TQABitmapDetach) (TQABitmap *bitmap);
```

`bitmap` A bitmap.

DESCRIPTION

Your `TQABitmapDetach` function is called whenever an application calls `QABitmapDetach`. Your function should, if necessary, load the bitmap specified by the `bitmap` parameter into memory on the device associated with your drawing engine (so that the caller can release the memory occupied by the bitmap). Your `TQABitmapDetach` function should not return until the bitmap has been completely loaded.

TQABitmapBindColorTable

A drawing engine may define a method to bind a color lookup table to a bitmap.

```
typedef TQAError (*TQABitmapBindColorTable) (
                    TQABitmap *bitmap,
                    TQAColorTable *colorTable);
```

`bitmap` A bitmap.

`colorTable` A color lookup table (as returned by a previous call to `QAColorTableNew`).

DESCRIPTION

Your `TQABitmapBindColorTable` function is called whenever an application calls `QABitmapBindColorTable`. Your function should bind the color lookup table specified by the `colorTable` parameter to the bitmap specified by the `bitmap` parameter. Note that the type of the specified color lookup table must match that of the pixel type of the bitmap to which it is bound. For example, a color lookup table of type `kQAColorTable_CL8_RGB32` can be bound only to a bitmap whose pixel type is `kQAPixel_CL8`.

TQABitmapDelete

A drawing engine must define a method to delete a bitmap.

```
typedef void (*TQABitmapDelete) (TQABitmap *bitmap);
```

`bitmap` A bitmap.

DESCRIPTION

Your `TQABitmapDelete` function is called whenever an application calls `QABitmapDelete`. Your function should delete the bitmap specified by the `bitmap` parameter.

Method Reporting Methods

To write a drawing engine, you need to implement a method for reporting some of your engine's methods to QuickDraw 3D RAVE.

A pointer to your drawing engine's method reporting method is passed as a parameter to the `QARegisterEngine` function. See page 1-113 for details.

TQAEngineGetMethod

A drawing engine must define a method to return pointers to some of its methods.

```
typedef TQAError (*TQAEngineGetMethod) (
                    TQAEngineMethodTag methodTag,
                    TQAEngineMethod *method);
```

`methodTag` A selector that determines which method is to be returned about your drawing engine. See "Drawing Engine Method Selectors" (page 1-65) for complete information about the available method selectors.

`method` On exit, a pointer to your drawing engine's method of the specified type.

DESCRIPTION

Your `TQAEngineGetMethod` function is called by QuickDraw 3D RAVE to retrieve the addresses of some of your engine's methods. Your function should return, in the `method` parameter, a pointer to the drawing engine method whose type is specified by the `methodTag` parameter.

Notice Methods

Your application can define a notice method that is called at specific times (for example, when the renderer is finished rendering an image). A pointer to your notice method is passed as a parameter to the `QASetNoticeMethod` function.

TQANoticeMethod

An application can define a method to respond asynchronously to certain events associated with the operation of QuickDraw 3D RAVE.

```
typedef void (*TQANoticeMethod) (
                    TQADrawContext *drawContext,
                    void *refCon);
```

`drawContext`	A draw context.
`refCon`	The reference constant associated with the notice method.

DESCRIPTION

Your `TQANoticeMethod` function is called by QuickDraw 3D RAVE at the times specified when an application installed the notice method using the `QASetNoticeMethod` function. For example, if the value of the `method` parameter passed to `QASetNoticeMethod` was `kQAMethod_RenderCompletion`, then the associated notice method is called whenever the renderer finishes rendering an image in the draw context specified by the `drawContext` parameter. The `refCon` parameter is an application-defined reference constant; this is simply the value of the `refCon` parameter that was passed to `QASetNoticeMethod`.

Note
You can install one notice method for each defined notice selector. See page 1-68 for a description of the available notice selectors. ◆

Summary of QuickDraw 3D RAVE

C Summary

Constants

Platform Values

```
#define kQAMacOS                                1
#define kQAGeneric                              2
#define kQAWin32                                3
```

Version Values

```
typedef enum TQAVersion {
    kQAVersion_Prerelease               = 0,
    kQAVersion_1_0                      = 1,
    kQAVersion_1_0_5                    = 2,
    kQAVersion_1_1                      = 3
} TQAVersion;
```

Pixel Types

```
typedef enum TQAImagePixelType {
    kQAPixel_Alpha1                     = 0,
    kQAPixel_RGB16                      = 1,
    kQAPixel_ARGB16                     = 2,
    kQAPixel_RGB32                      = 3,
    kQAPixel_ARGB32                     = 4,
    kQAPixel_CL4                        = 5,
    kQAPixel_CL8                        = 6
} TQAImagePixelType;
```

Color Lookup Table Types

```
typedef enum TQAColorTableType {
    kQAColorTable_CL8_RGB32             = 0,
    kQAColorTable_CL4_RGB32             = 1
} TQAColorTableType;
```

Device Types

```
typedef enum TQADeviceType {
    kQADeviceMemory                     = 0,
    kQADeviceGDevice                    = 1,
    kQADeviceWin32DC                    = 2,
    kQADeviceDDSurface                  = 3
} TQADeviceType;
```

Clip Types

```
typedef enum TQAClipType {
    kQAClipRgn                          = 0,
    kQAClipWin32Rgn                     = 1
} TQAClipType;
```

Tags for State Variables

```
typedef enum TQATagInt {
    kQATag_ZFunction                    = 0,        /*required variables*/
    kQATag_Antialias                    = 8,        /*optional variables*/
    kQATag_Blend                        = 9,
    kQATag_PerspectiveZ                 = 10,
    kQATag_TextureFilter                = 11,
    kQATag_TextureOp                    = 12,
    kQATag_CSGTag                       = 14,
    kQATag_CSGEquation                  = 15,
    kQATagGL_DrawBuffer                 = 100,      /*OpenGL variables*/
    kQATagGL_TextureWrapU               = 101,
    kQATagGL_TextureWrapV               = 102,
    kQATagGL_TextureMagFilter           = 103,
    kQATagGL_TextureMinFilter           = 104,
    kQATagGL_ScissorXMin                = 105,
    kQATagGL_ScissorYMin                = 106,
```

```
    kQATagGL_ScissorXMax                = 107,
    kQATagGL_ScissorYMax                = 108,
    kQATagGL_BlendSrc                   = 109,
    kQATagGL_BlendDst                   = 110,
    kQATagGL_LinePattern                = 111,
    kQATagGL_AreaPattern0               = 117,
    kQATagGL_AreaPattern31              = 148,
    kQATag_EngineSpecific_Minimum       = 1000
} TQATagInt;

typedef enum TQATagFloat {
    kQATag_ColorBG_a                    = 1,        /*required variables*/
    kQATag_ColorBG_r                    = 2,
    kQATag_ColorBG_g                    = 3,
    kQATag_ColorBG_b                    = 4,
    kQATag_Width                        = 5,
    kQATag_ZMinOffset                   = 6,
    kQATag_ZMinScale                    = 7,
    kQATagGL_DepthBG                    = 112,      /*OpenGL variables*/
    kQATagGL_TextureBorder_a            = 113,
    kQATagGL_TextureBorder_r            = 114,
    kQATagGL_TextureBorder_g            = 115,
    kQATagGL_TextureBorder_b            = 116
} TQATagFloat;

typedef enum TQATagPtr {
    kQATag_Texture                      = 13
} TQATagPtr;
```

Z Sorting Function Selectors

```
/*values for kQATag_ZFunction*/
#define kQAZFunction_None                       0
#define kQAZFunction_LT                         1
#define kQAZFunction_EQ                         2
#define kQAZFunction_LE                         3
#define kQAZFunction_GT                         4
#define kQAZFunction_NE                         5
#define kQAZFunction_GE                         6
#define kQAZFunction_True                       7
```

Antialiasing Selectors

```
/*values for kQATag_Antialias*/
#define kQAAntiAlias_Off                        0
#define kQAAntiAlias_Fast                       1
#define kQAAntiAlias_Mid                        2
#define kQAAntiAlias_Best                       3
```

Blending Operations

```
/*values for kQATag_Blend*/
#define kQABlend_PreMultiply                    0
#define kQABlend_Interpolate                    1
#define kQABlend_OpenGL                         2
```

Z Perspective Selectors

```
/*values for kQATag_PerspectiveZ*/
#define kQAPerspectiveZ_Off                     0
#define kQAPerspectiveZ_On                      1
```

Texture Filter Selectors

```
/*values for kQATag_TextureFilter*/
#define kQATextureFilter_Fast                   0
#define kQATextureFilter_Mid                    1
#define kQATextureFilter_Best                   2
```

Texture Operations

```
/*masks for kQATag_TextureOp*/
#define kQATextureOp_None                       0
#define kQATextureOp_Modulate                   (1 << 0)
#define kQATextureOp_Highlight                  (1 << 1)
#define kQATextureOp_Decal                      (1 << 2)
#define kQATextureOp_Shrink                     (1 << 3)
```

CSG IDs

```
/*values for kQATag_CSGTag*/
#define kQACSGTag_None                          0xffffffffUL
#define kQACSGTag_0                             0
#define kQACSGTag_1                             1
#define kQACSGTag_2                             2
#define kQACSGTag_3                             3
#define kQACSGTag_4                             4
```

Texture Wrapping Values

```
/*values for kQATagGL_TextureWrapU and kQATagGL_TextureWrapV*/
#define kQAGL_Repeat                            0
#define kQAGL_Clamp                             1
```

Source Blending Values

```
/*values for kQATagGL_BlendSrc*/
#define kQAGL_SourceBlend_XXX                   0
```

Destination Blending Values

```
/*values for kQATagGL_BlendDst*/
#define kQAGL_DestBlend_XXX                     0
```

Buffer Drawing Operations

```
/*masks for kQATagGL_DrawBuffer*/
#define kQAGL_DrawBuffer_None                   0
#define kQAGL_DrawBuffer_FrontLeft              (1 << 0)
#define kQAGL_DrawBuffer_FrontRight             (1 << 1)
#define kQAGL_DrawBuffer_BackLeft               (1 << 2)
#define kQAGL_DrawBuffer_BackRight              (1 << 3)
#define kQAGL_DrawBuffer_Front                  \
                        (kQAGL_DrawBuffer_FrontLeft | kQAGL_DrawBuffer_FrontRight)
#define kQAGL_DrawBuffer_Back                   \
                        (kQAGL_DrawBuffer_BackLeft | kQAGL_DrawBuffer_BackRight)
```

Line and Point Widths

```
/*values for kQATag_Width*/
#define kQAMaxWidth                                 128.0
```

Vertex Modes

```
typedef enum TQAVertexMode {
    kQAVertexMode_Point                 = 0,
    kQAVertexMode_Line                  = 1,
    kQAVertexMode_Polyline              = 2,
    kQAVertexMode_Tri                   = 3,
    kQAVertexMode_Strip                 = 4,
    kQAVertexMode_Fan                   = 5
} TQAVertexMode;
```

Gestalt Selectors

```
typedef enum TQAGestaltSelector {
    kQAGestalt_OptionalFeatures         = 0,
    kQAGestalt_FastFeatures             = 1,
    kQAGestalt_VendorID                 = 2,
    kQAGestalt_EngineID                 = 3,
    kQAGestalt_Revision                 = 4,
    kQAGestalt_ASCIINameLength          = 5,
    kQAGestalt_ASCIIName                = 6,
    kQAGestalt_AvailableTexMem          = 7
} TQAGestaltSelector;
```

Gestalt Optional Features Response Masks

```
#define kQAOptional_None                            0
#define kQAOptional_DeepZ                           (1 << 0)
#define kQAOptional_Texture                         (1 << 1)
#define kQAOptional_TextureHQ                       (1 << 2)
#define kQAOptional_TextureColor                    (1 << 3)
#define kQAOptional_Blend                           (1 << 4)
#define kQAOptional_BlendAlpha                      (1 << 5)
#define kQAOptional_Antialias                       (1 << 6)
#define kQAOptional_ZSorted                         (1 << 7)
#define kQAOptional_PerspectiveZ                    (1 << 8)
```

```
#define kQAOptional_OpenGL                      (1 << 9)
#define kQAOptional_NoClear                     (1 << 10)
#define kQAOptional_CSG                         (1 << 11)
#define kQAOptional_BoundToDevice               (1 << 12)
#define kQAOptional_CL4                         (1 << 13)
#define kQAOptional_CL8                         (1 << 14)
```

Gestalt Fast Features Response Masks

```
#define kQAFast_None                            0
#define kQAFast_Line                            (1 << 0)
#define kQAFast_Gouraud                         (1 << 1)
#define kQAFast_Texture                         (1 << 2)
#define kQAFast_TextureHQ                       (1 << 3)
#define kQAFast_Blend                           (1 << 4)
#define kQAFast_Antialiasing                    (1 << 5)
#define kQAFast_ZSorted                         (1 << 6)
#define kQAFast_CL4                             (1 << 7)
#define kQAFast_CL8                             (1 << 8)
```

Vendor and Engine IDs

```
#define kQAVendor_BestChoice                    (-1)
#define kQAVendor_Apple                         0
#define kQAVendor_ATI                           1
#define kQAVendor_Radius                        2
#define kQAVendor_Mentor                        3
#define kQAVendor_Matrox                        4
#define kQAVendor_Yarc                          5

#define kQAEngine_AppleSW                       0
#define kQAEngine_AppleHW                       (-1)
#define kQAEngine_AppleHW2                      1
```

Triangle Flags Masks

```
#define kQATriFlags_None                        0
#define kQATriFlags_Backfacing                  (1 << 0)
```

Texture Flags Masks

```
#define kQATexture_None                         0
#define kQATexture_Lock                         (1 << 0)
#define kQATexture_Mipmap                       (1 << 1)
#define kQATexture_NoCompression                (1 << 2)
#define kQATexture_HighCompression              (1 << 3)
```

Bitmap Flags Masks

```
#define kQABitmap_None                          0
#define kQABitmap_Lock                          (1 << 1)
#define kQABitmap_NoCompression                 (1 << 2)
#define kQABitmap_HighCompression               (1 << 3)
```

Draw Context Flags Masks

```
#define kQAContext_None                         0
#define kQAContext_NoZBuffer                    (1 << 0)
#define kQAContext_DeepZ                        (1 << 1)
#define kQAContext_DoubleBuffer                 (1 << 2)
#define kQAContext_Cache                        (1 << 3)
```

Drawing Engine Method Selectors

```
typedef enum TQAEngineMethodTag {
    kQADrawPrivateNew                           = 0,
    kQADrawPrivateDelete                        = 1,
    kQAEngineCheckDevice                        = 2,
    kQAEngineGestalt                            = 3,
    kQATextureNew                               = 4,
    kQATextureDetach                            = 5,
    kQATextureDelete                            = 6,
    kQABitmapNew                                = 7,
    kQABitmapDetach                             = 8,
    kQABitmapDelete                             = 9,
    kQAColorTableNew                            = 10,
    kQAColorTableDelete                         = 11,
    kQATextureBindColorTable                    = 12,
    kQABitmapBindColorTable                     = 13
} TQAEngineMethodTag;
```

Public Draw Context Method Selectors

```
typedef enum TQADrawMethodTag {
    kQASetFloat                                 = 0,
    kQASetInt                                   = 1,
    kQASetPtr                                   = 2,
    kQAGetFloat                                 = 3,
    kQAGetInt                                   = 4,
    kQAGetPtr                                   = 5,
    kQADrawPoint                                = 6,
    kQADrawLine                                 = 7,
    kQADrawTriGouraud                           = 8,
    kQADrawTriTexture                           = 9,
    kQADrawVGouraud                             = 10,
    kQADrawVTexture                             = 11,
    kQADrawBitmap                               = 12,
    kQARenderStart                              = 13,
    kQARenderEnd                                = 14,
    kQARenderAbort                              = 15,
    kQAFlush                                    = 16,
    kQASync                                     = 17,
    kQASubmitVerticesGouraud                    = 18,
    kQASubmitVerticesTexture                    = 19,
    kQADrawTriMeshGouraud                       = 20,
    kQADrawTriMeshTexture                       = 21,
    kQASetNoticeMethod                          = 22,
    kQAGetNoticeMethod                          = 23
} TQADrawMethodTag;
```

Notice Method Selectors

```
typedef enum TQAMethodSelector {
    kQAMethod_RenderCompletion                  = 0,
    kQAMethod_DisplayModeChanged                = 1
} TQAMethodSelector;
```

Data Types

Basic Data Types

```
typedef struct TQAEngine                    TQAEngine;

typedef struct TQATexture                   TQATexture;

typedef struct TQABitmap                    TQABitmap;

typedef struct TQAColorTable                TQAColorTable;

typedef struct TQADrawPrivate               TQADrawPrivate;
```

Memory Device Structure

```
typedef struct TQADeviceMemory {
    long                            rowBytes;
    TQAImagePixelType               pixelType;
    long                            width;
    long                            height;
    void                            *baseAddr;
} TQADeviceMemory;
```

Rectangle Structure

```
typedef struct TQARect {
    long                            left;
    long                            right;
    long                            top;
    long                            bottom;
} TQARect;
```

Macintosh Device and Clip Structures

```
typedef union TQAPlatformDevice {
    TQADeviceMemory                 memoryDevice;
    GDHandle                        gDevice;
} TQAPlatformDevice;
```

```
typedef union TQAPlatformClip {
    RgnHandle                           clipRgn;
} TQAPlatformClip;
```

Windows Device and Clip Structures

```
typedef union TQAPlatformDevice {
    TQADeviceMemory                     memoryDevice;
    HDC                                 hdc;
    struct {
        LPDIRECTDRAW                    lpDirectDraw;
        LPDIRECTDRAWSURFACE             lpDirectDrawSurface;
    };
} TQAPlatformDevice;
```

```
typedef union TQAPlatformClip {
    HRGN                                clipRgn;
} TQAPlatformClip
```

Generic Device and Clip Structures

```
typedef union TQAPlatformDevice {
    TQADeviceMemory                     memoryDevice;
} TQAPlatformDevice;
```

```
typedef union TQAPlatformClip {
    void                                *region;
} TQAPlatformClip;
```

Device Structure

```
typedef struct TQADevice {
    TQADeviceType                       deviceType;
    TQAPlatformDevice                   device;
} TQADevice;
```

Clip Data Structure

```
typedef struct TQAClip {
    TQAClipType                         clipType;
    TQAPlatformClip                     clip;
} TQAClip;
```

Image Structure

```
struct TQAImage {
    long                                width;
    long                                height;
    long                                rowBytes;
    void                                *pixmap;
};
typedef struct TQAImage TQAImage;
```

Vertex Structures

```
typedef struct TQAVGouraud {
    float                               x;
    float                               y;
    float                               z;
    float                               invW;
    float                               r;
    float                               g;
    float                               b;
    float                               a;
} TQAVGouraud;

typedef struct TQAVTexture {
    float                               x;
    float                               y;
    float                               z;
    float                               invW;
    float                               r;
    float                               g;
    float                               b;
    float                               a;
    float                               uOverW;
    float                               vOverW;
```

```
    float                               kd_r;
    float                               kd_g;
    float                               kd_b;
    float                               ks_r;
    float                               ks_g;
    float                               ks_b;
} TQAVTexture;
```

Draw Context Structure

```
struct TQADrawContext {
    TQADrawPrivate                      *drawPrivate;
    const TQAVersion                    version;
    TQASetFloat                         setFloat;
    TQASetInt                           setInt;
    TQASetPtr                           setPtr;
    TQAGetFloat                         getFloat;
    TQAGetInt                           getInt;
    TQAGetPtr                           getPtr;
    TQADrawPoint                        drawPoint;
    TQADrawLine                         drawLine;
    TQADrawTriGouraud                   drawTriGouraud;
    TQADrawTriTexture                   drawTriTexture;
    TQADrawVGouraud                     drawVGouraud;
    TQADrawVTexture                     drawVTexture;
    TQADrawBitmap                       drawBitmap;
    TQARenderStart                      renderStart;
    TQARenderEnd                        renderEnd;
    TQARenderAbort                      renderAbort;
    TQAFlush                            flush;
    TQASync                             sync;
    TQASubmitVerticesGouraud            submitVerticesGouraud;
    TQASubmitVerticesTexture            submitVerticesTexture;
    TQADrawTriMeshGouraud               drawTriMeshGouraud;
    TQADrawTriMeshTexture               drawTriMeshTexture;
    TQASetNoticeMethod                  setNoticeMethod;
    TQAGetNoticeMethod                  getNoticeMethod;
};
typedef struct TQADrawContext TQADrawContext;
```

Drawing Engine Methods Union

```
typedef union TQAEngineMethod {
    TQADrawPrivateNew                   drawPrivateNew;
    TQADrawPrivateDelete                drawPrivateDelete;
    TQAEngineCheckDevice                engineCheckDevice;
    TQAEngineGestalt                    engineGestalt;
    TQATextureNew                       textureNew;
    TQATextureDetach                    textureDetach;
    TQATextureDelete                    textureDelete;
    TQABitmapNew                        bitmapNew;
    TQABitmapDetach                     bitmapDetach;
    TQABitmapDelete                     bitmapDelete;
    TQAColorTableNew                    colorTableNew;
    TQAColorTableDelete                 colorTableDelete;
    TQATextureBindColorTable            textureBindColorTable;
    TQABitmapBindColorTable             bitmapBindColorTable;
} TQAEngineMethod;
```

Public Draw Context Methods Union

```
typedef union TQADrawMethod {
    TQASetFloat                         setFloat;
    TQASetInt                           setInt;
    TQASetPtr                           setPtr;
    TQAGetFloat                         getFloat;
    TQAGetInt                           getInt;
    TQAGetPtr                           getPtr;
    TQADrawPoint                        drawPoint;
    TQADrawLine                         drawLine;
    TQADrawTriGouraud                   drawTriGouraud;
    TQADrawTriTexture                   drawTriTexture;
    TQADrawVGouraud                     drawVGouraud;
    TQADrawVTexture                     drawVTexture;
    TQADrawBitmap                       drawBitmap;
    TQARenderStart                      renderStart;
    TQARenderEnd                        renderEnd;
    TQARenderAbort                      renderAbort;
    TQAFlush                            flush;
    TQASync                             sync;
    TQASubmitVerticesGouraud            submitVerticesGouraud;
```

```
    TQASubmitVerticesTexture            submitVerticesTexture;
    TQADrawTriMeshGouraud               drawTriMeshGouraud;
    TQADrawTriMeshTexture               drawTriMeshTexture;
    TQASetNoticeMethod                  setNoticeMethod;
    TQAGetNoticeMethod                  getNoticeMethod;
} TQADrawMethod;
```

Indexed Triangle Structure

```
typedef struct TQAIndexedTriangle {
    unsigned long                       triangleFlags;
    unsigned long                       vertices[3];
} TQAIndexedTriangle;
```

QuickDraw 3D RAVE Routines

Creating and Deleting Draw Contexts

```
TQAError QADrawContextNew         (const TQADevice *device,
                                   const TQARect *rect,
                                   const TQAClip *clip,
                                   const TQAEngine *engine,
                                   unsigned long flags,
                                   TQADrawContext **newDrawContext);

void QADrawContextDelete          (TQADrawContext *drawContext);
```

Creating and Deleting Color Lookup Tables

```
TQAError QAColorTableNew          (const TQAEngine *engine,
                                   TQAColorTableType tableType,
                                   void *pixelData,
                                   long transparentIndexFlag,
                                   TQAColorTable **newTable);

void QAColorTableDelete           (const TQAEngine *engine, TQAColorTable *colorTable);
```

Manipulating Textures and Bitmaps

```
TQAError QATextureNew             (const TQAEngine *engine,
                                   unsigned long flags,
                                   TQAImagePixelType pixelType,
                                   const TQAImage images[],
                                   TQATexture **newTexture);

TQAError QATextureDetach          (const TQAEngine *engine,
                                   TQATexture *texture);

TQAError QATextureBindColorTable  (const TQAEngine *engine,
                                   TQATexture *texture,
                                   TQAColorTable *colorTable);

void QATextureDelete              (const TQAEngine *engine, TQATexture *texture);

TQAError QABitmapNew              (const TQAEngine *engine,
                                   unsigned long flags,
                                   TQAImagePixelType pixelType,
                                   const TQAImage *image,
                                   TQABitmap **newBitmap);

TQAError QABitmapDetach           (const TQAEngine *engine, TQABitmap *bitmap);

TQAError QABitmapBindColorTable   (const TQAEngine *engine,
                                   TQABitmap *bitmap,
                                   TQAColorTable *colorTable);

void QABitmapDelete               (const TQAEngine *engine, TQABitmap *bitmap);
```

Managing Drawing Engines

```
TQAEngine *QADeviceGetFirstEngine (const TQADevice *device);

TQAEngine *QADeviceGetNextEngine  (const TQADevice *device,
                                   const TQAEngine *currentEngine);

TQAError QAEngineCheckDevice      (const TQAEngine *engine, const TQADevice *device);

TQAError QAEngineGestalt          (const TQAEngine *engine,
                                   TQAGestaltSelector selector,
                                   void *response);

TQAError QAEngineEnable           (long vendorID, long engineID);

TQAError QAEngineDisable          (long vendorID, long engineID);
```

Manipulating Draw Contexts

```
#define QAGetFloat(drawContext,tag) \
                            (drawContext)->getFloat (drawContext,tag)

#define QASetFloat(drawContext,tag,newValue) \
                            (drawContext)->setFloat (drawContext,tag,newValue)

#define QAGetInt(drawContext,tag) \
                            (drawContext)->getInt (drawContext,tag)

#define QASetInt(drawContext,tag,newValue) \
                            (drawContext)->setInt (drawContext,tag,newValue)

#define QAGetPtr(drawContext,tag) \
                            (drawContext)->getPtr (drawContext,tag)

#define QASetPtr(drawContext,tag,newValue) \
                            (drawContext)->setPtr (drawContext,tag,newValue)

#define QADrawPoint(drawContext,v) \
                            (drawContext)->drawPoint (drawContext,v)

#define QADrawLine(drawContext,v0,v1) \
                            (drawContext)->drawLine (drawContext,v0,v1)

#define QADrawTriGouraud(drawContext,v0,v1,v2,flags) \
                    (drawContext)->drawTriGouraud (drawContext,v0,v1,v2,flags)

#define QADrawTriTexture(drawContext,v0,v1,v2,flags) \
                    (drawContext)->drawTriTexture (drawContext,v0,v1,v2,flags)

#define QASubmitVerticesGouraud(drawContext,nVertices,vertices) \
                (drawContext)->submitVerticesGouraud(drawContext,nVertices,vertices)

#define QASubmitVerticesTexture(drawContext,nVertices,vertices) \
                (drawContext)->submitVerticesTexture(drawContext,nVertices,vertices)

#define QADrawTriMeshGouraud(drawContext,nTriangle,triangles) \
                (drawContext)->drawTriMeshGouraud (drawContext,nTriangle,triangles)

#define QADrawTriMeshTexture(drawContext,nTriangle,triangles) \
                (drawContext)->drawTriMeshTexture (drawContext,nTriangle,triangles)

#define QADrawVGouraud(drawContext,nVertices,vertexMode,vertices,flags) \
(drawContext)->drawVGouraud (drawContext,nVertices,vertexMode,vertices,flags)
```

```
#define QADrawVTexture(drawContext,nVertices,vertexMode,vertices,flags) \
(drawContext)->drawVTexture (drawContext,nVertices,vertexMode,vertices,flags)

#define QADrawBitmap(drawContext,v,bitmap) \
                            (drawContext)->drawBitmap (drawContext,v,bitmap)

#define QARenderStart(drawContext,dirtyRect,initialContext) \
            (drawContext)->renderStart (drawContext,dirtyRect,initialContext)

#define QARenderEnd(drawContext,modifiedRect) \
                            (drawContext)->renderEnd (drawContext,modifiedRect)

#define QARenderAbort(drawContext) (drawContext)->renderAbort (drawContext)

#define QAFlush(drawContext) (drawContext)->flush (drawContext)

#define QASync(drawContext) (drawContext)->sync (drawContext)

#define QAGetNoticeMethod(drawContext, method, completionCallBack, refCon) \
    (drawContext)->getNoticeMethod (drawContext, method, completionCallBack, refCon)

#define QASetNoticeMethod(drawContext, method, completionCallBack, refCon) \
    (drawContext)->setNoticeMethod (drawContext, method, completionCallBack, refCon)
```

Registering a Custom Drawing Engine

```
TQAError QARegisterEngine         (TQAEngineGetMethod engineGetMethod);

TQAError QARegisterDrawMethod     (TQADrawContext *drawContext,
                                   TQADrawMethodTag methodTag,
                                   TQADrawMethod method);
```

Application-Defined Routines

Public Draw Context Methods

```
typedef float (*TQAGetFloat)      (const TQADrawContext *drawContext, TQATagFloat tag);

typedef void (*TQASetFloat)       (TQADrawContext *drawContext,
                                   TQATagFloat tag,
                                   float newValue);

typedef unsigned long (*TQAGetInt)(const TQADrawContext *drawContext, TQATagInt tag);
```

```
typedef void (*TQASetInt)         (TQADrawContext *drawContext,
                                   TQATagInt tag,
                                   unsigned long newValue);

typedef void *(*TQAGetPtr)        (const TQADrawContext *drawContext, TQATagPtr tag);

typedef void (*TQASetPtr)         (TQADrawContext *drawContext,
                                   TQATagPtr tag,
                                   const void *newValue);

typedef void (*TQADrawPoint)      (const TQADrawContext *drawContext,
                                   const TQAVGouraud *v);

typedef void (*TQADrawLine)       (const TQADrawContext *drawContext,
                                   const TQAVGouraud *v0,
                                   const TQAVGouraud *v1);

typedef void (*TQADrawTriGouraud) (const TQADrawContext *drawContext,
                                   const TQAVGouraud *v0,
                                   const TQAVGouraud *v1,
                                   const TQAVGouraud *v2,
                                   unsigned long flags);

typedef void (*TQADrawTriTexture) (const TQADrawContext *drawContext,
                                   const TQAVTexture *v0,
                                   const TQAVTexture *v1,
                                   const TQAVTexture *v2,
                                   unsigned long flags);

typedef void (*TQASubmitVerticesGouraud) (
                                   const TQADrawContext *drawContext,
                                   unsigned long nVertices,
                                   const TQAVGouraud *vertices);

typedef void (*TQASubmitVerticesTexture) (
                                   const TQADrawContext *drawContext,
                                   unsigned long nVertices,
                                   const TQAVTexture *vertices);

typedef void (*TQADrawTriMeshGouraud) (
                                   const TQADrawContext *drawContext,
                                   unsigned long nTriangles,
                                   const TQAIndexedTriangle *triangles);
```

```
typedef void (*TQADrawTriMeshTexture) (
                                    const TQADrawContext *drawContext,
                                    unsigned long nTriangles,
                                    const TQAIndexedTriangle *triangles);

typedef void (*TQADrawVGouraud)   (const TQADrawContext *drawContext,
                                    unsigned long nVertices,
                                    TQAVertexMode vertexMode,
                                    const TQAVGouraud vertices[],
                                    const unsigned long flags[]);

typedef void (*TQADrawVTexture)   (const TQADrawContext *drawContext,
                                    unsigned long nVertices,
                                    TQAVertexMode vertexMode,
                                    const TQAVTexture vertices[],
                                    const unsigned long flags[]);

typedef void (*TQADrawBitmap)     (const TQADrawContext *drawContext,
                                    const TQAVGouraud *v,
                                    TQABitmap *bitmap);

typedef void (*TQARenderStart)    (const TQADrawContext *drawContext,
                                    const TQARect *dirtyRect,
                                    const TQADrawContext *initialContext);

typedef TQAError (*TQARenderEnd)  (const TQADrawContext *drawContext,
                                    const TQARect *modifiedRect);

typedef TQAError (*TQARenderAbort)(const TQADrawContext *drawContext);

typedef TQAError (*TQAFlush)      (const TQADrawContext *drawContext);

typedef TQAError (*TQASync)       (const TQADrawContext *drawContext);

typedef TQAError (*TQAGetNoticeMethod) (
                                    const TQADrawContext *drawContext,
                                    TQAMethodSelector method,
                                    TQANoticeMethod *completionCallBack,
                                    void **refCon);

typedef TQAError (*TQASetNoticeMethod) (
                                    const TQADrawContext *drawContext,
                                    TQAMethodSelector method,
                                    TQANoticeMethod completionCallBack,
                                    void *refCon);
```

Private Draw Context Methods

```
typedef TQAError (*TQADrawPrivateNew) (
                                TQADrawContext *newDrawContext,
                                const TQADevice *device,
                                const TQARect *rect,
                                const TQAClip *clip,
                                unsigned long flags);

typedef void (*TQADrawPrivateDelete) (
                                TQADrawPrivate *drawPrivate);

typedef TQAError (*TQAEngineCheckDevice) (
                                const TQADevice *device);

typedef TQAError (*TQAEngineGestalt) (
                                TQAGestaltSelector selector, void *response);
```

Color Lookup Table Methods

```
typedef TQAError (*TQAColorTableNew)(
                                TQAColorTableType pixelType,
                                void *pixelData,
                                long transparentIndex,
                                TQAColorTable **newTable);

typedef void (*TQAColorTableDelete) (
                                TQAColorTable *colorTable);
```

Texture and Bitmap Methods

```
typedef TQAError (*TQATextureNew) (unsigned long flags,
                                TQAImagePixelType pixelType,
                                const TQAImage images[],
                                TQATexture **newTexture);

typedef TQAError (*TQATextureDetach) (
                                TQATexture *texture);

typedef TQAError (*TQATextureBindColorTable) (
                                TQATexture *texture,
                                TQAColorTable *colorTable);

typedef void (*TQATextureDelete)  (TQATexture *texture);
```

```
typedef TQAError (*TQABitmapNew)  (unsigned long flags,
                                   TQAImagePixelType pixelType,
                                   const TQAImage *image,
                                   TQABitmap **newBitmap);

typedef TQAError (*TQABitmapDetach) (
                                   TQABitmap *bitmap);

typedef TQAError (*TQABitmapBindColorTable) (
                                   TQABitmap *bitmap,
                                   TQAColorTable *colorTable);

typedef void (*TQABitmapDelete)   (TQABitmap *bitmap);
```

Method Reporting Methods

```
typedef TQAError (*TQAEngineGetMethod) (
                                   TQAEngineMethodTag methodTag,
                                   TQAEngineMethod *method);
```

Notice Methods

```
typedef void (*TQANoticeMethod)   (TQADrawContext *drawContext, void *refCon);
```

Result Codes

kQANoErr	0	No error
kQAError	1	Generic error code
kQAOutOfMemory	2	Insufficient memory for requested operation
kQANotSupported	3	Requested feature is not supported
kQAOutOfDate	4	A newer drawing engine was registered
kQAParamErr	5	Invalid parameter
kQAGestaltUnknown	6	Requested Gestalt type isn't available
kQADisplayModeUnsupported	7	Engine cannot render to the display in its current mode

Bibliography

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.

Glassner, A.S. ed., *Graphics Gems,* Harcourt Brace Jovanovich, Boston, 1990 and following.

Hearn, Donald, and M. Pauline Baker, *Computer Graphics,* second edition, Prentice-Hall, Englewood Cliffs, NJ, 1986.

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

3D Three-dimensional.

3D driver See **drawing engine.**

accelerator See **graphics accelerator.**

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.

antialiasing The smoothing of jagged edges on a displayed shape by modifying the transparencies of individual pixels along the shape's edge. Compare **aliasing.**

antialiasing mode A draw context state variable that determines the kind of antialiasing applied to a drawing context.

API See **application programming interface.**

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 stipple pattern One of 32 patterns that specifies which bits in an area are to be drawn and which are masked out. Compare **line stipple pattern.**

background color The color that is used when a buffer is cleared by `QARenderStart`.

bitmap A two-dimensional array of values, each of which represents the state of one pixel**.** Defined by the `TQABitmap` data type. See also **pixmap.**

blend matte A pixel map that defines the blending of the pixels in a source and destination image. The alpha channel value of a pixel in the matte governs the relative intensity of a source pixel when copied to the destination image.

cache See **draw context cache.**

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.**

clipping region A region to which an application can limit drawing. The initial clipping region of a graphics port is an arbitrarily large rectangle: one that covers the entire QuickDraw coordinate plane. An application can set the clipping region to any arbitrary region, to aid in drawing inside the graphics port.

CLUT See **color lookup table.**

color lookup table (CLUT) A data structure that maps an index to an actual color.

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.**

constructive solid geometry (CSG) A way of modeling solid objects constructed from the union, intersection, or difference of other solid objects.

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 ID A number, attached to an object as an attribute, that identifies an object for CSG operations.

deep z buffering Z buffering with a resolution of greater than or equal to 24 bits per pixel.

device See **hardware device, virtual device.**

device structure A data structure that contains basic information about a virtual device. Defined by the `TQADevice` data type.

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.**

double buffering A technique that uses two buffers to store images.

draw context A structure that maintains state information and other data associated with a drawing engine and device. A draw context is defined by the `TQADrawContext` data structure.

draw context cache A draw context whose `QAContext_Cache` flag is set.

draw context method See **private draw context method, public draw context method.**

draw context state variable See **state variable.**

draw context structure A data structure that contains basic information about a draw context. Defined by the `TQADrawContext` data type.

drawing engine Any software component that supports the low-level rasterization operations required for interactive 3D rendering.

flat shading See **constant shading.**

`GDevice` **data structure** A data structure of type `GDevice` that holds information about the physical characteristics of a video device or offscreen graphics world.

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.**

Gouraud vertex A vertex used for drawing Gouraud-shaded triangles, and also for drawing points and lines. A Gouraud vertex is defined by the `TQAVGouraud` data structure. Compare **texture vertex.**

graphics accelerator Any hardware device used to accelerate rendering.

graphics device A virtual device that represents a video device (such as a plug-in video card or built-in video interface) that controls a screen, or an offscreen graphics world (which allows your application to build complex images off the screen before displaying them). For a video device or an offscreen graphics world, a graphics device is defined by the `GDHandle` data type. Compare **memory device.**

hardware device A physical part of a Macintosh computer, or a piece of external equipment, that can exchange information with applications or with the Macintosh Operating System. Compare **virtual device.**

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.**

high-quality texture mapping Texture mapping that uses trilinear interpolation or an equivalent algorithm.

image The two-dimensional product of rendering.

image structure A data structure that defines a bitmap or a pixmap. Defined by the `TQAImage` data type.

indexed triangle A data structure that defines three vertices and a set of triangle flags. Defined by the `TQAIndexedTriangle` data type.

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**.

Lambertian reflection See **diffuse reflection.**

Lambert illumination A method of calculating the illumination of a point on a surface based on diffuse reflection. Compare **Phong illumination.**

line stipple pattern A pattern that specifies which bits in a line are to be drawn and which are masked out. Compare **area stipple pattern.**

matte See **blend matte.**

memory device A virtual device that represents an area of memory. A memory device is defined by a structure of type `TQAMemoryDevice`. Compare **graphics device.**

memory device structure A data structure that contains basic information about a memory device. Defined by the `TQAMemoryDevice` data type.

method selector A value that indicates of which method a drawing engine should return the address.

mipmapping A method of storing texture maps in an array of pixel images of varying pixel depths. The first element in the array must be the mipmap page having the highest resolution, with a width and height that are even powers of 2. Each subsequent pixel image in the array should have a width and height that are half those of the previous image.

notice method An application-defined method that is called asynchronously at certain specified times during the operations of QuickDraw 3D RAVE.

OpenGL A graphics library developed by Silicon Graphics that you can use to create, configure, render, and interact with models of three-dimensional objects.

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.**

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.**

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 `TQAPixmap` data type. See also **bitmap.**

plug-in See **drawing engine.**

private draw context method A draw context method that is called only by QuickDraw 3D RAVE. Compare **public draw context method.**

public draw context method A method to which a pointer is contained in one of the method fields of a draw context structure. Compare **private draw context method.**

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 Acceleration Layer The hardware abstraction layer that is comprised of QuickDraw 3D RAVE and all registered drawing engines.

QuickDraw 3D Renderer Acceleration Virtual Engine (RAVE) The part of the Macintosh system software that controls 3D drawing engines.

rasterization The process of determining values for the pixels in a rendered image. Also called *scan conversion*.

RAVE See **QuickDraw 3D Renderer Acceleration Virtual Engine.**

rectangle structure A data structure that defines the area into which a drawing engine is to draw. Defined by the `TQARect` data type.

render To create an image (on the screen or some other medium) of a model.

renderer The software or hardware that renders an image of a model. See also **interactive renderer, wireframe renderer.**

rendering The process of creating an image (on the screen or some other medium) of a model. See also **rasterization.**

scan conversion See **rasterization.**

scene cache See **draw context cache.**

scissor box A rectangle that determines which pixels can be modified by drawing commands.

smooth shading See **Gouraud shading, Phong shading**.

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.

state variable A variable associated with a draw context that maintains state information about the draw context. See also **state value, tag.**

state value The value of a state variable.

tag A unique identifier associated with a state variable.

texture magnification function A function that is called when a pixel being textured maps to an area that is less than or equal to one texture element. See also **texture minifying function.**

texture mapping A technique wherein a predefined image (the texture) is mapped onto the surface of an object in a model. See also **high-quality texture mapping.**

texture mapping filter mode A state variable that determines how a drawing engine performs texture mapping filtering.

texture mapping operation A state variable that determines how a drawing engine performs texture mapping.

texture minifying function A function that is called when a pixel being textured maps to an area that is greater than one texture element. See also **texture magnification function.**

texture mode A value that determines certain features of a texture map.

texture vertex A vertex used for drawing triangles to which a texture is to be mapped. A texture vertex is defined by the `TQAVTexture` data structure. Compare **Gouraud vertex.**

transparency The ability of an object to allow light to pass through it.

transparency blending function A function that determines the kind of transparency blending applied to a drawing context when combining new pixels (the "source" pixels) with the pixels already in a frame buffer (the "destination" pixels).

triangle mode A value that determines how a drawing engine draws a triangle.

triangle mesh A mesh composed entirely of triangles.

trimesh See **triangle mesh.**

***uv* clamping** See **clamp.**

***uv* wrapping** See **wrap.**

vertex A dimensionless position in three-dimensional space at which two or more lines (for instance, edges) intersect. Defined by the `TQAVGouraud` and `TQAVTexture` data types. See also **Gouraud vertex, texture vertex.**

vertex mode A value that determines how a drawing engine interprets and draws an array of vertices.

virtual device A representation of a hardware device that is the destination for a drawing engine. A virtual device is defined by a structure of type `TQADevice`. Compare **hardware device.** See also **graphics device, memory device.**

visual line determination See **hidden line removal**.

visual surface determination See **hidden surface removal**.

wireframe renderer A renderer that creates line drawings of models. See also **interactive renderer**.

wrap For a shader effect, to replicate the entire effect across the mapped area. Compare **clamp.**

z perspective control A state variable that determines whether the `z` or the `invW` field of a vertex (of type `TQAVGouraud` or `TQAVTexture`) is to be used for hidden surface removal.

z sorting function A function that determines which surfaces are to be removed during hidden surface removal.

z test mode See **z sorting function.**

Index

Numerals

3D drivers. *See* drawing engines

A

alpha channels 1-36, 1-44, 1-52, 1-60, 1-75, 1-76
  using for blend mattes 1-22
  using for transparency 1-21 to 1-22
antialiasing modes 1-22 to 1-23, 1-40, 1-48
area stipple patterns 1-43

B

background colors 1-13, 1-18, 1-44 to 1-45
back-to-front rendering 1-60
bitmap flags 1-64
bitmaps
  methods for 1-144 to 1-146
  routines for 1-88 to 1-91
blend mattes 1-22

C

caches. *See* draw context caches
clamp 1-53
clip data structures 1-72
clip types 1-38
CLUT. *See* color lookup tables
color look tables
  methods for 1-139 to 1-140
color lookup tables
  binding to a bitmap 1-90, 1-145
  binding to a texture map 1-87, 1-143
  pixel types for 1-37
  routines for creating and deleting 1-82 to 1-84
  types of 1-37
constructive solid geometry 1-8, 1-41, 1-53 to 1-54, 1-61
CSG. *See* constructive solid geometry
CSG equations 1-41
CSG IDs 1-41, 1-53

D

devices. *See* hardware devices, virtual devices
device structures 1-14, 1-72
device types 1-38
double buffering 1-8
draw context caches 1-20 to 1-21
draw context flags 1-65
draw context methods. *See* private draw context methods, public draw context methods
draw contexts 1-12 to 1-13
  creating 1-17 to 1-18
  drawing in 1-19 to 1-20
  repositioning 1-18
  routines for creating and deleting 1-81 to 1-82
  routines for manipulating 1-94 to 1-113
  setting state variables of 1-18, 1-19
draw context state variables. *See* state variables
draw context structures 1-77 to 1-80
drawing engine methods
  selectors for 1-66
drawing engines 1-10 to 1-12
  defined 1-8
  finding 1-16 to 1-17
  optional features 1-11
  registering 1-113
  required features 1-10
  routines for 1-91 to 1-94

writing 1-23 to 1-34

E

engine IDs 1-63

F

fast features, selectors for 1-61

G

Gouraud vertices 1-19, 1-73
graphics devices
  defined 1-13

H

hidden surface removal 1-8

I

image structures 1-73
indexed triangle structures 1-80
interactive renderer
  optional features 1-11

L

line stipple patterns 1-43

M

mattes. *See* blend mattes
memory devices
  defined 1-13
memory device structures 1-14, 1-69
method reporting methods 1-146 to 1-147
method selectors 1-66
mipmapping 1-64, 1-86, 1-142

N

notice methods 1-147 to 1-148
  methods for getting and setting 1-134 to 1-135
  routines for getting and setting 1-112 to 1-113
  selectors for 1-68
  writing 1-147 to 1-148

O

OpenGL 1-8, 1-30 to 1-34
OpenGL buffer drawing modes 1-55
OpenGL texture wrapping modes 1-54
optional features, selectors for 1-59

P

pixel types 1-35
plug-ins. *See* drawing engines
private draw context methods 1-136 to 1-139
  writing 1-26 to 1-27
public draw context methods 1-115 to 1-135
  registering 1-114
  selectors for 1-67
  writing 1-24 to 1-26

Q

QABitmapBindColorTable function 1-90
QABitmapDelete function 1-90
QABitmapDetach function 1-89
QABitmapNew function 1-88
QADeviceGetFirstEngine function 1-16, 1-91
QADeviceGetNextEngine function 1-16, 1-92
QADrawBitmap function 1-106
QADrawContextDelete function 1-82
QADrawContextNew function 1-81
QADrawLine function 1-99
QADrawPoint function 1-98
QADrawTriGouraud function 1-99
QADrawTriMeshGouraud function 1-102
QADrawTriMeshTexture function 1-103
QADrawTriTexture function 1-100
QADrawVGouraud function 1-104
QADrawVTexture function 1-105
QAEngineCheckDevice function 1-92
QAEngineDisable function 1-94
QAEngineEnable function 1-94
QAEngineGestalt function 1-93
  selectors for 1-57
QAFlush function 1-110
QAGetFloat function 1-95
QAGetInt function 1-96
QAGetNoticeMethod function 1-112
QAGetPtr function 1-97
QARegisterDrawMethod function 1-114
QARegisterEngine function 1-113
QARenderAbort function 1-109
QARenderEnd function 1-108
QARenderStart function 1-107
QASetFloat function 1-95
QASetInt function 1-97
QASetNoticeMethod function 1-112
QASetPtr function 1-98
QASubmitVerticesGouraud function 1-101
QASubmitVerticesTexture function 1-101
QASync function 1-111
QATextureDelete function 1-87
QATextureDetach function 1-86
QATextureNew function 1-85
QuickDraw 3D Acceleration Layer 1-9
QuickDraw 3D RAVE 1-7 to 1-170
  application-defined routines in 1-115 to 1-148
  constants for 1-34 to 1-68
  data structures for 1-69 to 1-81
  defined 1-8
  naming conventions 1-34
  result codes 1-170
  routines in 1-81 to 1-115
  sample code for 1-13 to 1-23
  version of 1-35

R

RAVE. *See* QuickDraw 3D Renderer Acceleration Virtual Engine
RAVE control panel 1-91
rectangle structures 1-70
result codes 1-170

S

sample routines
  MyDrawPoint 1-25
  MyDrawPrivateDelete 1-27
  MyDrawPrivateNew 1-26
  MyEngineGestalt 1-28
  MyEngineGetMethod 1-29
  MyFindPreferredEngine 1-16
  MySetBackgroundToBlack 1-18
scene caches. *See* draw context caches
scissor boxes 1-43
state variables
  defined 1-12
  setting 1-18 to 1-19
  tags for 1-38 to 1-47

T

tags
  defined 1-12

texture border colors 1-45 to 1-46
texture flags 1-64
texture magnification functions 1-42
texture mapping filter modes 1-51
texture mapping operations 1-41, 1-51
  supporting in OpenGL 1-33 to 1-34
texture minifying functions 1-42
texture modes 1-64
textures
  compressing 1-64
  determining memory available for 1-58
  methods for 1-141 to 1-144
  routines for 1-85 to 1-88
texture vertices 1-20, 1-75
TQABitmapDelete method 1-146
TQABitmapDetach method 1-145
TQABitmapNew method 1-144
TQAClip data type 1-73
TQADevice data type 1-72
TQADeviceMemory data type 1-69
TQADrawBitmap method 1-129
TQADrawContext data type 1-78
TQADrawLine method 1-121
TQADrawPoint method 1-25, 1-120
TQADrawPrivateDelete method 1-27, 1-137
TQADrawPrivateNew method 1-26, 1-136
TQADrawTriGouraud method 1-121
TQADrawTriMeshGouraud function 1-125
TQADrawTriMeshTexture function 1-126
TQADrawTriTexture method 1-122
TQADrawVGouraud method 1-127
TQADrawVTexture method 1-128
TQAEngineCheckDevice method 1-138
TQAEngineGestalt method 1-27 to 1-28, 1-138
TQAEngineGetMethod method 1-27, 1-28, 1-29 to 1-30, 1-147
TQAFlush method 1-133
TQAGetFloat method 1-116
TQAGetInt method 1-117
TQAGetNoticeMethod function 1-134
TQAGetPtr method 1-119
TQAImage data type 1-73
TQAIndexedTriangle data structure 1-80
TQAPlatformClip data type 1-70, 1-72
TQAPlatformDevice data type 1-70, 1-72
TQARect data type 1-70
TQARenderAbort method 1-132
TQARenderEnd method 1-131
TQARenderStart method 1-130
TQASetFloat method 1-116
TQASetInt method 1-118
TQASetNoticeMethod function 1-135
TQASetPtr method 1-119
TQASubmitVerticesGouraud function 1-123
TQASubmitVerticesTexture function 1-124
TQASync method 1-134
TQATextureDelete method 1-143
TQATextureDetach method 1-142
TQATextureNew method 1-139, 1-140, 1-141
TQAVGouraud data type 1-74
TQAVTexture data type 1-75
transparency blending functions 1-40, 1-49
  destination blending values 1-55
  source blending values 1-54
  supporting in OpenGL 1-30 to 1-33
triangle flags 1-63
triangle meshes
  drawing 1-102 to 1-104, 1-125 to 1-126
  introduced 1-11
  submitting vertices for 1-101 to 1-102, 1-123 to 1-125
triangle modes 1-63
trimeshes. *See* triangle meshes

U

uv clamping. *See* clamp
uv wrapping. *See* wrap

V

vendor IDs 1-62
version, of QuickDraw 3D RAVE 1-35
vertex modes 1-56
vertices. *See* Gouraud vertices, texture vertices
virtual devices

defined 1-13
specifying 1-14 to 1-16

Z

z perspective controls 1-40, 1-50
z sorting functions 1-40, 1-47
z test modes. *See* z sorting functions

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 IINTX 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 Adobe Letter Gothic.

WRITER
Tim Monroe

ILLUSTRATOR
Sandee Karr

PROJECT MANAGER
Patricia Eastman

Special thanks to Mike Kelley and Brent Pease.