QuickDraw™ 3D Renderer Acceleration Virtual Engine

RAVE

A 3D Graphics Hardware Abstraction Layer

Engineering Reference Specification

Version 1.0.5
March 28, 1996

Brent Pease
Technical Lead, QD3D RAVE, IMG
ALink: PEASE
email: brent@apple.com
Mike Kelley
Manager, Dynamic Media, , IMG
ALink: KELLEY.M
email: mwk@apple.com

This page is intentionally
almost blank

Contents

Chapter 1: Organization of this Document

This document is an Engineering Requirements Specification (ERS) for the Power Macintosh low level 3D driver, called RAVE. Version 0.2 of this document was a draft version distributed to third parties for review. Version 0.9 was a major revision which incorporated third party feedback.

The 1.0 release of RAVE described here matches implementation used by the QuickDraw™ 3D 1.0 interactive renderer. All portions of this document can be considered final, except for the extended features and modes used for OpenGL compatibility. These are still under review.

The 1.0.2 release is identical to 1.0, except that `QARenderEnd()`, `QAFlush()`, `QASync()`, and `QARenderAbort()` now return an error code.

◊ The 1.0.5 release changed the name from Tinsel Town to RAVE and adds new functionality which includes the TriMesh and color lookup tables. The TriMesh is described in Chapter 6 and color lookup tables are also described in chapter 6 under the section "Using color lookup tables."

This document can be divided into three sections:

Chapters 1 - 3 provide an introduction to RAVE. This material is useful for any developer considering using RAVE, or for QuickDraw 3D developers who would like more information on the mechanism used to plug drawing engines into the QuickDraw 3D interactive renderer.

Chapters 4 - 8 provide a specification of the RAVE application programming interface (API). These are the calls used by an application to draw an image with RAVE. This material is important for application or middle-ware developers who need to know how to use RAVE as a drawing engine, and for developers who are planning to add their own RAVE drawing engine.

Chapters 9 - 10 describe how to add a new RAVE drawing engine. This section is primarily of use by hardware or software vendors who wish to add their own custom 3D drawing engine to RAVE.

This document assumes the reader is familiar with low-level 3D rendering algorithms. For those who would like additional information, the following reference (usually referred to as "Foley and van Dam") will be useful:

Computer Graphics, Principles and Practice, 2nd edition. Foley, van Dam, Feiner, Hughes. Addison-Wesly Publishing Company. Chapters 3, 15, 16, 18, and 19.

◊ Throughout this document, important paragraphs are marked by a ◊ symbol.

Code examples, and names which are taken from code or include files, are shown in Courier typeface, e.g. `QADrawTriGouraud()`.

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

Chapter 2: Background

What is a low level 3D driver?

A low level 3D driver is a software layer designed to support the low-level rasterization operations required for interactive 3D rendering. In many respects a 3D driver interface is very similar to a 2D drawing API. There are, however, several key differences:

— 3D drawing requires a Z (depth) value, which is used to perform hidden surface removal.

— Support for double-buffered (or back buffered) image display is necessary. Double buffering conceals the flashing caused by re-drawing the image. High performance double buffering can also used to avoid the tearing artifacts often caused by updating a window at high speed.

— Special 3D rasterization modes such as texture mapping are supported.

What is not included?

There is no absolute definition of what should be included in a low level 3D driver. However, this document assumes that none of the following are directly supported:

— No transformation, shading or clipping.

— No I/O mechanisms (i.e. the driver is a drawing mechanism only).

— No high level primitives such as curved surfaces.

Who are the users?

A low level 3D driver is used for four purposes:

— It provides a hardware abstraction layer (HAL) that allows system software (e.g. QuickDraw 3D) to utilize a wide variety of hardware without code changes.

— It provides third party hardware vendors with a means to ship 3D acceleration hardware that can plug-and-play with a variety of 3D applications.

— It provides a highly optimized means for a third party middle-ware vendor (e.g. a vendor of a game development framework) to access hardware and Apple's optimized software rasterizers.

— It provides a highly optimized means for specialized application vendors (e.g. games and entertainment applications) to access hardware acceleration through a very flexible and lightweight mechanism.

Note that in the first three cases, the 3D driver layer operates as a System Programming Interface (SPI). However, the fourth bullet implies use by application developers, i.e. as an Application Programming Interface (API).

Chapter 3: Functional Overview

Plug-and-play for 3D drawing engines

RAVE is designed to provide independent software and hardware vendors (ISVs and IHVs) with a simple and efficient means of adding low level 3D rasterizers to the Macintosh. Typically these low level 3D rasterizers are designed to accelerate 3D rendering for interactive use.

3D rasterizers plugged into RAVE are referred to as *drawing engines*. When a drawing engine is registered with the RAVE manager, it becomes available for use by all applications running on the system. QuickDraw 3D's interactive renderer uses RAVE, so registering a drawing engine with RAVE automatically makes it available for use by all QuickDraw 3D-based applications as well.

For an IHV, RAVE is used primarily as a means of making the 3D rasterization features of the vendor's hardware available for use by Macintosh applications. By writing a RAVE plug-in, an IHV's product can immediately plug-and-play with 3D software on the Macintosh.

For most ISVs, RAVE will not be used directly; instead, it will be an enabling technology that provides a flexible plug-and-play environment for 3D acceleration. However, some ISVs will use RAVE directly:

— Vendors of middle-ware (e.g. a third party game development framework) should use RAVE as their interactive 3D drawing method.

— Vendors of games or entertainment applications who prefer an extremely low level, but very lightweight, drawing library may use RAVE in place of a higher level API such as QuickDraw 3D.

◊ *Warning: Because RAVE provides no support for Apple's 3D metafile, 3D user interface, plug-in shaders, or windows that cross devices, it is not recommended for use by general applications.*

Designed for speed

RAVE is intended for interactive 3D rendering. To provide maximum performance, RAVE has been designed to provide the minimum possible overhead between the application and the drawing engine. Two key design features were made to meet this goal:

— Calls from an application to RAVE do not require a context change.

— Calls to a RAVE drawing engine do not pass through an intermediate manager layer — the application calls directly into the selected drawing engine's code.

◊ Because of these features, calling a drawing engine through RAVE provides the same level of performance as linking the engine directly with the application.

Minimum feature set

RAVE does not require that all drawing engines provide the same features. The minimum feature set of a RAVE drawing engine is:

— Hidden surface removal (usually Z buffering with a minimum of 16 bits/pixel)

— Points and lines of programmable width

— Gouraud shaded triangles

— Bitmaps of 1, 16 or 32 bits/pixel

— Double buffering

Optional features

More advanced RAVE drawing engines may support any or all of the following features:

— High precision hidden surface removal (24 bits or more)

— Perspective corrected hidden surface removal

— Texture mapping, fast and/or high quality

— Transparency blending, RGB or ARGB

— Antialiasing, fast and/or high quality

— Z sorted rendering of non-opaque objects

— OpenGL support, which includes a collection of features such as scissoring, multiple blending modes, area and line stipple patterns, etc.

Apple-supplied drawing engine

Apple ships RAVE with a software-only drawing engine which is highly optimized for the Power Macintosh. This drawing engine is guaranteed to be able to draw to any device. This basic drawing engine provides the following features (in addition to the minimum feature set):

— Z buffering with 16 or 32 bits of precision

— Direct rendering at 16 or 32 bit/pixel (fewer than 16 bits/pixel is supported with lower performance)

— Perspective-corrected texture mapping

Multiple device support

RAVE does not require that a drawing engine be capable of drawing to all devices in the system. Instead, when the application wishes to choose a drawing engine, it must specify to which device the drawing will be performed. Each drawing engine is queried by RAVE to determine if it can support the indicated device; if not, the drawing engine will not be offered to the application.

This means that a drawing engine can specialize itself for the device(s) for which it is most suitable. For example, a drawing engine that uses a frame buffer's built-in 3D acceleration hardware may have no effective means of rendering to a different device. Rather than forcing that drawing engine to implement an inefficient solution, RAVE allows the drawing engine to work with only its native device.

◊ *Warning: This means that RAVE does **not** provide automatic support for windows that cross multiple devices. It is the application's responsibility to recognize these cases, and construct multiple RAVE drawing contexts (potentially with different drawing engines) as necessary to draw the entire window.*

QuickDraw 3D High-Level API

RAVE and the high level QuickDraw 3D API are designed to work together as a team. The QuickDraw 3D interactive renderer uses the available RAVE drawing engines to accelerate interactive rendering for all applications that use QuickDraw 3D.

The high level QuickDraw 3D API provides much greater functionality than RAVE, and is the recommended API for general purpose 3D programming. Some examples of features included in QuickDraw 3D are:

- Cut and paste of 3D data
- Automatic support of multiple devices
- Powerful, high level datatypes such as NURBS and mesh
- A highly optimized interactive renderer
- A plug-in mechanism for high quality renderers (RAVE supports only interactive rendering)
- A plug-in shader mechanism
- User interface guidelines and tool kits

OpenGL support

Although the minimum feature set of a RAVE drawing engine doesn't provide all the features necessary for OpenGL, a drawing engine may *optionally* choose to add support for these features. A drawing engine that provides these optional features can be used to accelerate OpenGL compliant rendering, as well as to accelerate QuickDraw 3D.

API naming conventions

All functions, datatypes and constants declared by the RAVE.h include file follow naming conventions to avoid conflicts with application code:

— All function names begin with the prefix `QA`, e.g. `QADrawTriGouraud()`.

— All data type names begin with the prefix `TQA`, e.g. `TQADrawContext`.

— All constant names begin with the prefix `kQA`, e.g. `kQAAntiAlias_Fast`.

Chapter 4: The Drawing Context

◊ All data structures and prototypes in this document are defined in the RAVE.h include file.

All RAVE drawing is performed into a drawing context, referred to as a `TQADrawContext`. More than one `TQADrawContext` can exist simultaneously; each one maintains its own state information, and is unaffected by calls which reference any other `TQADrawContext`. All drawing calls take a `TQADrawContext` as their first parameter.

The `TQADrawContext` data structure is shown below:

```
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;
};
```

The `drawPrivate` field points to the private data maintained by the drawing engine associated with this context. The `version` field is a constant field initialized by the RAVE manager; it indicates the RAVE manager version. The remaining fields (`setFloat`, `setInt` etc.) are function pointers to the methods of the drawing engine.

None of the `TQADrawContext` fields are directly referenced by the application. Instead, the application uses the macros defined in RAVE.h to call the methods. For example, the application code shown below sets the background color of a `TQADrawContext` to opaque black (the `QASetFloat()` function is described in more detail in Chapter 5: State Variables):

```
#include "RAVE.h"
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);
```

During compilation, macro expansion causes the `QASetFloat()` call to be replaced with the code shown below. Because this code directly calls the drawing engine's `setFloat` method, no intermediate manager layer is necessary, providing the highest possible performance.

```
(drawContext)->setFloat (drawContext, 1, 1.0);
```

◊ It is recommended that all calls to drawing engine methods be performed with the macros defined in RAVE.h, as directly referencing the `TQADrawContext` method fields may complicate ports to future versions of RAVE.

◊ The `TQADrawContext` methods are not static — some calls (e.g. `QASetInt()`) can cause the methods to change. This is discussed in more detail in Chapter 9: Adding a New Drawing Engine. This issue does not affect applications that always use the macros defined in RAVE.h to access the drawing methods.

The TQADevice

When a `TQADrawContext` is created, it requires information about where drawing should be performed. This information is provided by a `TQADevice`, which is passed as a parameter to `QADrawContextNew()` (discussed later in this chapter). A `TQADevice` represents any one of a variety of different device types into which drawing can occur. On the Macintosh, a `TQADevice` can represent either a `GDevice`, or a region of memory.

On the Macintosh, the `TQADevice` structure (and its supporting datatypes) are:

```
typedef enum TQADeviceType
{
   kQADeviceMemory    = 0,
   kQADeviceGDevice   = 1
} TQADeviceType;

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

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

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

Using TQADevice to draw to memory

The following example code initializes a TQADevice for drawing to memory:

```
TQADevice  myDevice;
long       targetMemory [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 = targetMemory;
```

Drawing to memory occurs in the native pixel format of the platform. Note that not all drawing engines support drawing to memory (see Choosing a drawing engine, later in this chapter).

Using TQADevice to draw to a GDevice

The following example code initializes a TQADevice for drawing to a GDevice:

```
TQADevice  myDevice;
GDHandle   gDeviceHandle;
...
myDevice.deviceType = kQADeviceGDevice;
myDevice.device.gDevice = gDeviceHandle;
```

Drawing across multiple GDevices

An individual `TQADrawContext` can render to only a single `TQADevice`, and therefore to only a single `GDevice`. Because a Macintosh window can cross multiple `GDevices`, it is the application's responsibility to determine which `GDevices` the window touches, and to create a separate `TQADrawContext` for each one.

QAEngineGestalt

To assist the application in choosing a drawing engine, `QAEngineGestalt()` provides information about the functionality of a drawing engine:

```
TQAError QAEngineGestalt (
   const TQAEngine    *engine,   /* Engine being queried */
   TQAGestaltSelector selector,  /* Gestalt parameter being requested */
   void               *response);/* (Out) Buffer that gets response */
```

`engine` selects which drawing engine is being queried.

`selector` is an enumerated constant which indicates which gestalt value is being requested. It can be any one of the following values:

`kQAGestalt_OptionalFeatures`: Returns a mask of one or more `kQAOptional_xxx` flags (described below). `response` should point to an `unsigned long`.

`kQAGestalt_FastFeatures` Returns: a mask of one or more `kQAFast_xxx` flags (described below). `response` should point to an `unsigned long`.

`kQAGestalt_VendorID`: Returns the vendor ID of this engine. `response` should point to a `long`.

`kQAGestalt_EngineID`: Returns the engine ID of this engine. `response` should point to a `long`.

`kQAGestalt_Revision`: Returns the revision number of this engine (larger values are more recent). `response` should point to a `long`.

`kQAGestalt_ASCIINameLength`: Returns the `strlen()` of the .`kQAGestalt_ASCIIName` (described next). `response` should point to a `long`.

`kQAGestalt_ASCIIName`: Copies the ascii name of this drawing engine into `response`. `response` should point to a C string, whose length is determined by `kQAGestalt_ASCIINameLength` (described above).

`response` a pointer to where the retrieved information should be stored. The type and size of this data is dependent on the value of `selector`.

Gestalt Optional Features

The `kQAGestalt_OptionalFeatures` gestalt response is a bit mask for which any combination of the flags shown below can be ORed together. Each flag indicates whether the named feature is supported by the drawing engine. Note that supported features are not necessary accelerated (e.g. the drawing may perform the feature in software).

`kQAOptional_DeepZ`: Deep Z buffering (i.e. Z buffer resolution > = 24 bits/pixel).

`kQAOptional_Texture`: Texture mapping.

`kQAOptional_TextureHQ`: High quality texture mapping (tri-linear interpolation or equivalent).

`kQAOptional_TextureColor`: Full color texture modulation and highlight.

`kQAOptional_Blend`: Transparency blending.

`kQAOptional_BlendAlpha`: Transparency blending that outputs an alpha channel.

`kQAOptional_Antialias`: Antialiasing.

`kQAOptional_ZSorted`: Z sorted rendering (e.g. for transparency).

`kQAOptional_PerspectiveZ`: Perspective corrected hidden surface removal.

`kQAOptional_OpenGL`: Extended OpenGL feature set.

`kQAOptional_NoClear`: This engine doesn't clear the buffer before drawing, so double-buffering may not be required in some applications.

Gestalt Fast Features

The `kQAGestalt_FastFeatures` gestalt value is a bit mask for which any combination of the flags shown below can be ORed together. Each flags indicates whether the named feature is accelerated by the drawing engine. Unfortunately, it is difficult to define exactly what accelerated means — we consider these flags to mean that the named feature is performed substantially faster than it would be in software with a fast CPU.

`kQAFast_Line:` Line drawing.

`kQAFast_Gouraud:` Gouraud shading.

`kQAFast_Texture:` Texture mapping.

`kQAFast_TextureHQ:` High quality texture mapping.

`kQAFast_Blend:` Transparency blending.

`kQAFast_Antialiasing:` Antialiasing.

`kQAFast_ZSorted:` Z sorted rendering.

Choosing a drawing engine

Not all drawing engines can be used with all `TQADevices`. For example, some drawing engines may not support `kQADeviceMemory`. Others may only support a particular `GDevice`. Therefore, once the application has initialized the target `TQADevice`, the application must scan through the available drawing engines to choose one which is capable of drawing to the target `TQADevice`. If more than one drawing engine can draw to the `TQADevice`, the application must choose between the available engines.

The application can search through the available drawing engines with the `QADeviceGetFirstEngine()` and `QADeviceGetNextEngine()` functions:

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

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

◊ By default, `QADeviceGetFirstEngine()` returns the *preferred drawing engine* for the indicated device — in most cases, this is the best engine to choose for high performance rendering. The following heuristic is used to choose the preferred engine:

First choice: The user's preference (e.g. set by the monitors control panel).

Second choice: The drawing engine which is tightly coupled to the indicated device (i.e. it can render only to that device).

Third choice: The drawing engine which accelerates the most features.

If the default choice of the preferred engine isn't appropriate, the application can search for an engine with the desired features. For example, assume an application requires a drawing engine that accelerates texture mapping. This could be coded as:

```
TQAEngine *findPreferredEngine (TQADevice *device)
{
   TQAEngine       *engine;
   unsigned long   fast;

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

Here, QADeviceGetFirstEngine() and QADeviceGetNextEngine() are used to loop through the drawing engines which can target device, and QAEngineGestalt() is used to get information about each engine's specific features.

◊ If a NULL TQADevice pointer is passed to QADeviceGetFirstEngine() or QADeviceGetNextEngine(), the available drawing engines are returned without any device checking. This is useful when the application needs to query information about all the available engines regardless of which devices are supported.

Creating a TQADrawContext

Once a TQADevice has been initialized, and a drawing engine (i.e. TQAEngine) has been chosen, a TQADrawContext can be created. This is performed with the QADrawContextNew() function:

```
TQAError QADrawContextNew (
   const TQADevice  *device,/* Target device */
   const TQARect    *rect,  /* Target rectangle (device coordinates) */
   const TQAClip    *clip,  /* 2D clip region */
   const TQAEngine  *engine,/* Drawing engine to use */
   unsigned long    flags,  /* Mask of kQAContext_xxx */
   TQADrawContext   **newDrawContext); /* (Out) New TQADrawContext */
```

device is a pointer to the target TQADevice.

rect is the rectangular region of the target TQADevice to which this TQADrawContext will draw. rect is in device coordinates.

clip is the 2D clipping region for rect, or NULL (indicating no clipping). This 2D clipping region will be applied to any pixels before they are drawn to the TQADevice. Clipping is not supported when device is of type kQADeviceMemory (i.e. clip must be NULL).

engine specifies the drawing engine, as discussed in Choosing a drawing engine, earlier in this chapter.

The flags parameter is a bit mask for which any combination of the following flags can be ORed together:

kQAContext_NoZBuffer: The TQADrawContext should not be Z buffered.

kQAContext_DeepZ: Z should have at least 24 bits of precision.

kQAContext_DoubleBuffer: The TQADrawContext should be double buffered.

kQAContext_Cache: This draw context will be used to create a scene cache. See Chapter 8: Buffering and Synchronization for more discussion.

newDrawContext is a pointer to the TQADrawContext pointer which is initialized by this call. If an error occurs, *newDrawContext is set to NULL.

For example, to create a double buffered display with a Z buffer:

```
TQADrawContext  *drawContext;
if (QADrawContextNew (&myDevice, &myRect, &myClip,
      engine, kQAContext_DoubleBuffer, &drawContext) != kQANoErr)
{
   /* Error! Could not create TQADrawContext */
}
```

Deleting a TQADrawContext

`QADrawContextNew()` allocates memory and other resources which must eventually be freed. The application performs this by calling `QADrawContextDelete()` when it is finished with the `TQADrawContext`:

```
void QADrawContextDelete (
   TQADrawContext     *drawContext);  /* Drawing context to delete */
```

For example, to delete the drawing context created in the previous example:

```
QADrawContextDelete (drawContext);
```

Re-positioning a TQADrawContext

There is no call to re-position a `TQADrawContext` once it has been created. Instead, the old context must be deleted, and a new `TQADrawContext` created for the new window position.

2D clipping a TQADrawContext

The drawing engine determines the 2D clipping for a `TQADrawContext` from the `clip` parameter passed to `QADrawContextNew()`. There is no call to change the clip region of a `TQADrawContext` after it has been created; instead, the old `TQADrawContext` must be deleted, and a new one created.

The `clip` parameter is of type `TQAClip`. Like `TQADevice`, `TQAClip` supports different types of clipping information depending on the platform. On the Macintosh, `TQAClip` provides a `clipRgn`, as shown below:

```
TQAClip    myClip;
RgnHandle  clipRgn;
...
myClip.clipType = kQAClipRgn;
myClip.clip.clipRgn = clipRgn;
```

Supporting different pixel depths

Because the drawing engine can test the pixel depth of the target `TQADevice`, an engine has complete freedom to choose which bit depths it supports — engines that don't support the pixel depth of the targeted `TQADevice` will never be returned by `QADeviceGetFirstEngine()` or `QADeviceGetNextEngine()`

It is recommended that all drawing engines support 16 and 32 bits/pixel.

Chapter 5: State Variables

A `TQADrawContext` retains a variety of state information about the current rendering modes. This information is stored as state variables. Each state variable has a unique identifier constant called a *tag*.

State variables may be either float, unsigned long or pointer. The list below (from RAVE.h) shows the tag names, and their datatypes:

```
* These variables are required by all drawing engines:
kQATag_ZFunction           (Int)   One of kQAZFunction_xxx
kQATag_ColorBG_a           (Float) Background color alpha
kQATag_ColorBG_r           (Float) Background color red
kQATag_ColorBG_g           (Float) Background color green
kQATag_ColorBG_b           (Float) Background color blue
kQATag_Width               (Float) Line and point width (pixels)
kQATag_ZMinOffset          (Float) Min Z offset to guarantee visibility
kQATag_ZMinScale           (Float) Min Z scale to guarantee visibility

* These variables are used for optional features:
kQATag_Antialias           (Int)   One of kQAAntialias_xxx
kQATag_Blend               (Int)   One of kQABlend_xxx
kQATag_PerspectiveZ        (Int)   One of kQAPerspectiveZ_xxx
kQATag_TextureFilter       (Int)   One of kQATextureFilter_xxx
kQATag_TextureOp           (Int)   Mask of kQATextureOp_xxx
kQATag_Texture             (Ptr)   Pointer to current TQATexture

* These variables are used for OpenGL support:
kQATagGL_DrawBuffer        (Int)   Mask of kQAGL_DrawBuffer_xxx
kQATagGL_TextureWrapU      (Int)   kQAGL_Clamp or kQAGL_Repeat
kQATagGL_TextureWrapV      (Int)   kQAGL_Clamp or kQAGL_Repeat
kQATagGL_TextureMagFilter  (Int)   kQAGL_Nearest or kQAGL_Linear
kQATagGL_TextureMinFilter  (Int)   kQAGL_Nearest, etc.
kQATagGL_ScissorXMin       (Int)   Minimum X value for scissor rectangle
kQATagGL_ScissorYMin       (Int)   Minimum Y value for scissor rectangle
kQATagGL_ScissorXMax       (Int)   Maximum X value for scissor rectangle
kQATagGL_ScissorYMax       (Int)   Maximum Y value for scissor rectangle
kQATagGL_BlendSrc          (Int)   Source blending operation
kQATagGL_BlendDst          (Int)   Destination blending operation
kQATagGL_LinePattern       (Int)   Line rasterization pattern
kQATagGL_AreaPattern0      (Int)   First of 32 area pattern registers
kQATagGL_AreaPattern31     (Int)   Last of 32 area pattern registers
kQATagGL_DepthBG           (Float) Background Z
kQATagGL_TextureBorder_a   (Float) Texture border color alpha
kQATagGL_TextureBorder_r   (Float) Texture border color red
kQATagGL_TextureBorder_g   (Float) Texture border color green
kQATagGL_TextureBorder_b   (Float) Texture border color blue
```

These tag values are grouped into three enumerated types: `TQATagInt`, `TQATagFloat` and `TQATagPtr`. Some variables are used only for optional features; these variables do not need to be supported by drawing engines which don't provide those optional features.

Setting a variable with QASetFloat/QASetInt/QASetPtr

State variables are set using `QASetInt()`, `QASetFloat()` or `QASetPtr()`, depending on whether the variable is `unsigned long`, `float` or a pointer:

```
void QASetFloat (
   TQADrawContext  *drawContext,      /* Draw context */
   TQATagFloat     tag,               /* Tag of variable to set */
   float           newValue);         /* New value for variable */

void QASetInt (
   TQADrawContext  *drawContext,      /* Draw context */
   TQATagInt       tag,               /* Tag of variable to set */
   unsigned long   newValue);         /* New value for variable */

void QASetPtr (
   TQADrawContext  *drawContext,      /* Draw context */
   TQATagPtr       tag,               /* Tag of variable to set */
   const void      *newValue);        /* New value for variable */
```

For example, to set the Z hidden surface test to `kQAZFunction_LT`:

```
TQADrawContext  *drawContext;
...
QASetInt (drawContext, kQATag_ZFunction, kQAZFunction_LT);
```

To allow future expansion, a drawing engine must accept `QASetFloat/Int/Ptr()` calls for tags which it doesn't recognize. These calls should be treated as no-ops. Similarly, calls to set variables used only for optional features that the drawing engine does not support should be treated as no-ops.

Reading a variable with QAGetFloat/QAGetInt/QAGetPtr

State variables are read using `QAGetFloat()`, `QAGetInt()` or `QAGetPtr()`, depending on whether the variable is `unsigned long`, `float` or a pointer:

```
float QAGetFloat (
   const TQADrawContext  *drawContext,   /* Draw context */
   TQATagFloat           tag);           /* Tag of variable to get */

unsigned long QAGetInt (
   const TQADrawContext  *drawContext,   /* Draw context */
   TQATagInt             tag);           /* Tag of variable to get */

void *QAGetPtr (
   const TQADrawContext  *drawContext,   /* Draw context */
   TQATagPtr             tag);           /* Tag of variable to get */
```

For example, to read the red component of the current background color:

```
TQADrawContext  *drawContext;
float           backgroundColor_r;
...
backgroundColor_r = QAGetFloat (drawContext, kQATag_ColorBG_r);
```

`QAGetFloat/Int/Ptr()` calls for unrecognized or unsupported tag values return 0.

Required state variables

kQATag_ColorBG_a/r/g/b

Float. Required. Default value is 0.0. Range 0.0 < = value < = 1.0

These four variables set the background color. The background color is used when clearing a buffer with `QARenderStart()`. `kQATag_ColorBG_a` sets the alpha value, `kQATag_ColorBG_r` sets the red value, `kQATag_ColorBG_g` sets the green value, `kQATag_ColorBG_b` sets the blue value.

kQATag_Width

Float. Required. Default value is 1.0. Range 0.0 < value < kQAMaxWidth

This variable sets the width of points or lines. Width is measured in pixels. kQAMaxWidth is currently defined as 128.0.

kQATag_ZFunction

Int. Required.. Default value is kQAZFunction_LT if a Z-buffered draw context, or kQAZFunction_None if not.

This sets the current Z test mode used for hidden surface removal. Draw contexts which aren't Z-buffered only support `kQAZFunction_None`. For Z-buffered contexts, `kQATag_ZFunction` can have one of the following values:

`kQAZFunction_None`: Z is neither tested nor written
`kQAZFunction_LT`: Znew < Zbuffer
`kQAZFunction_True`: Znew is always visible (and written)

The following Z test modes are also defined, but are only supported by drawing engines which support the optional OpenGL features:

`kQAZFunction_EQ`: Znew = = Zbuffer
`kQAZFunction_LE`: Znew < = Zbuffer
`kQAZFunction_GT`: Znew > Zbuffer
`kQAZFunction_NE`: Znew ! = Zbuffer
`kQAZFunction_GE:` Znew > = Zbuffer

◊ For drawing engines which support `kQAOptional_PerspectiveZ`, when `kQATag_PerspectiveZ` is set to `kQAPerspectiveZ_On`, `kQATag_ZFunction` should be interpreted so it yields the same visual result as for `kQAPerspectiveZ_Off`. For example, `kQAZFunction_LT`, which means "show the closest surface", is equivalent to a visibility function of InvWnew > InvWbuffer.

kQATag_ZMinOffset, kQATag_ZMinScale

Float. Required. READ ONLY — value is set by drawing engine.

These read-only variables are used by the drawing engine to indicate the minimum scale and offset for Z that must be performed to guarantee that a drawn object will pass a `kQAZFunction_LT` hidden surface test. For example, these variables would be used by an application that needed to draw a triangle, and then re-draw the triangle edges *slightly* closer in a different color. `kQATag_ZMinScale` indicates the value by which Z must be scaled, and `kQATag_ZMinOffset` indicates the value by which Z must be offset. The code example below shows how these values are used:

```
TQAVGouraud   myVertex;
...
/* Draw the point once */
QADrawPoint (drawContext, &myVertex);

/* Adjust Z to guarantee that the second draw will also be visible */
myVertex.z *= QAGetFloat (drawContext, kQATag_ZMinScale);
myVertex.z += QAGetFloat (drawContext, kQATag_ZMinOffset);
QADrawPoint (drawContext, &myVertex);
```

Typically, a drawing engine that uses fixed point Z will return 1.0 for scale, and a small *negative* value (e.g. -1/65536) for offset. A drawing engine that uses floating point Z will usually return 0.0 for offset, and a value slightly less than 1.0 (e.g. 0.9999) for scale.

◊ For drawing engines which support `kQAOptional_PerspectiveZ`, when `kQATag_PerspectiveZ` is set to `kQAPerspectiveZ_On`, the values returned for offset and scale are changed to the values used to scale and offset `Invw`. In this mode, scale will be >= 1.0, and offset is a small *positive* number.

Optional state variables

kQATag_Antialias

Int. Optional: Only necessary if kQAOptional_Antialias is true. Default is kQAAntiAlias_Fast.

This variable controls the current antialiasing mode. It can be set to any of:

`kQAAntiAlias_Off`: Antialiasing is forced off.

`kQAAntiAlias_Fast`: Do whatever antialiasing can be performed with *no* speed penalty (this often means antialiasing is turned off).

`kQAAntiAlias_Mid`: Turn on mid-quality antialiasing. This is the recommended setting when antialiasing at interactive speeds is desired.

`kQAAntiAlias_Best`: Turn on the highest quality antialiasing. This indicates that high quality antialiasing, even at the expense of interactive performance, is desired.

kQATag_Blend

Int. Optional: Only necessary if kQAOptional_Blend is true. Default value is kQABlend_PreMultiply.

This variable controls the current transparency blending model. It can have one of the following values (see Chapter 6: Rendering with transparency for more discussion of these modes):

`kQABlend_PreMultiply`: Use a pre-multiplied transparency blending function.

`kQABlend_Interpolate`: Use an interpolated transparency blending function.

`kQABlend_OpenGL`: Use the blending function defined by the `kQATag_BlendSrc` and `kQATag_BlendDst` state variables. This mode is only supported by drawing engines that set `kQAOptional_OpenGL` true.

kQATag_PerspectiveZ

Int. Optional: Only necessary if kQAOptional_PerspectiveZ is true. Default value is kQAPerspectiveZ_Off.

This variable controls whether the `Z` or the `InvW` field of `TQAVGouraud`/`TQAVTexture` is used for hidden surface removal. When `kQATag_PerspectiveZ` is set to `kQAPerspectiveZ_Off`, normal hidden surface removal using `Z` is performed. When set to `kQAPerspectiveZ_On`, hidden surface removal is performed with `InvW`, causing perspective-correct hidden surface removal. See `kQATag_ZFunction` for further discussion.

kQATag_Texture

Pointer. Optional: Only necessary if kQAOptional_Texture is true. Default value is NULL.

This variable holds a pointer to the current texture map. Texture map pointers are created by `TQATextureNew()`, e.g.:

```
TQATexture      *texture;
TQADrawContext  *drawContext;
TQAEngine       *engine;
if (QATextureNew (engine, kQATexture_None, kQAPixel_RGB32,
          images, &texture) != kQANoErr)
{
   ... Couldn't create texture
}
...
QASetPtr (drawContext, kQATag_Texture, texture);
```

kQATag_TextureFilter

Int. Optional: Only necessary if kQAOptional_Texture is true. Default value is kQATextureFilter_Fast.

This variable sets the current texture mapping filter mode. It can have one of the following values:

`kQATextureFilter_Fast`: The fastest texture map filtering mode available (usually means no filtering).

`kQATextureFilter_Mid`: Mid-quality texture filtering. This is the recommended setting when filtered texture mapping at interactive speeds is desired.

`kQATextureFilter_Best`: Turn on the highest quality texture filtering. This indicates that high quality, even at the expense of interactive performance, is desired.

kQATag_TextureOp

Int. Optional: Only necessary if kQAOptional_Texture is true. Default value is kQATextureOp_None.

This variable sets the current texture mapping operation. It is a bit mask for which any combination of the following flags can be ORed together:

`kQATextureOp_Modulate`: The texture map color is modulated with the interpolated `kd_r`, `kd_g` and `kd_b` values.

`kQATextureOp_Highlight`: The interpolated value of `ks_r`, `ks_g`, and `ks_b` are added to the texture map color.

`kQATextureOp_Decal`: When the texture map alpha is zero, replace the texture map color with the interpolated `r`, `g`, and `b` values.

`kQATextureOp_Shrink`: The drawing engine should tweak the incoming `u` and `v` values such that a range of 0.0 `<=` `u,v` `<=` 1.0 is guaranteed not to cause wrapping.

More detail on the texture mapping operations can be found in Chapter 6: Rendering with texture mapping.

OpenGL state variables

In general, the OpenGL state variables will correspond one-to-one with the state variables that affect rasterization in the OpenGL API itself. These are not fully specified yet, but we're working on it...

Chapter 6: Drawing

RAVE supports drawing of four types of primitives: points, lines, triangles and bitmaps. Each primitive has its own `QADraw...()` function; these functions are described in this chapter. The final sections of this chapter provide additional detail on rendering with transparency, texture mapping and antialiasing.

Points, lines and triangles are defined by vertices. RAVE uses two different vertex data types: `TQAVGouraud` for Gouraud shading, and `TQAVTexture` for texture mapping. These vertex data types are described next.

TQAVGouraud data type

The `TQAVGouraud` data structure is used to specify position, depth, color and transparency information for Gouraud shaded triangles, and for drawing points and lines.

```
typedef struct TQAVGouraud
{
   float    x;      /* X pixel coordinate, 0.0 <= x < width */
   float    y;      /* Y pixel coordinate, 0.0 <= y < height */
   float    z;      /* Z coordinate, 0.0 <= z <= 1.0 */
   float    invW;   /* 1 / w; required only for kQAPerspectiveZ_On */

   float    r;      /* Red, 0.0 <= r <= 1.0 */
   float    g;      /* Green, 0.0 <= g <= 1.0 */
   float    b;      /* Blue, 0.0 <= b <= 1.0 */
   float    a;      /* Alpha, 0.0 <= a <= 1.0, 1.0 is opaque */
} TQAVGouraud;
```

The `x` and `y` fields specify the vertex position in 2D coordinates relative to the upper left of the `rect` passed to `QADrawContextNew()`. `x` and `y` are floating point values specified in pixels.

`z` specifies the depth of the vertex. 0.0 is closest, 1.0 is farthest.

`invW` is used only by drawing engines which support `kQAOptional_PerspectiveZ`. For these engines, when the state variable `kQATag_PerspectiveZ` is set to `kQAPerspectiveZ_On`, hidden surface removal is performed with `invW` instead of with `z`. This causes hidden surface removal to be perspective-corrected. `invW` is the opposite of Z — the larger `invW` is, the closer the object. See `kQATag_ZFunction` for more discussion.

Vertex color is indicated by the `r`, `g`, and `b` fields, which represent a standard linear RGB color space. `a` represents transparency, 1.0 = opaque, 0.0 = completely transparent.

◊ By default, RAVE operates in a *pre-multiplied* transparency mode. See Rendering with transparency, later in this chapter, for more detail.

TQAVTexture data type

The `TQAVTexture` data structure is used to specify position, depth, transparency and texture mapping information for texture mapped triangles. Not all the fields are required; many are used only when the `kQATag_TextureOp` state variable is set to enable more complex texturing modes. The `kQATag_Texture` state variable specifies which texture map to use while rendering the texture mapped objects.

```
typedef struct TQAVTexture
{
   float   x;      /* X pixel coordinate, 0.0 <= x < width */
   float   y;      /* Y pixel coordinate, 0.0 <= y < height */
   float   z;      /* Z coordinate, 0.0 <= z <= 1.0 */
   float   invW;   /* 1 / w (always required) */

   /* rgb are used only when kQATextureOp_Decal is set.
      a is always required */

   float   r;      /* Red, 0.0 <= r <= 1.0 */
   float   g;      /* Green, 0.0 <= g <= 1.0 */
   float   b;      /* Blue, 0.0 <= b <= 1.0 */
   float   a;      /* Alpha, 0.0 <= a <= 1.0, 1.0 == opaque */

   /* uOverW and vOverW are required by all modes */

   float   uOverW; /* u / w */
   float   vOverW; /* v / w */

   /* kd_r/g/b are used only when kQATextureOp_Modulate is set */

   float   kd_r;   /* Scale factor for texture red, 0.0 <= kd_r */
   float   kd_g;   /* Scale factor for texture green, 0.0 <= kd_g */
   float   kd_b;   /* Scale factor for texture blue, 0.0 <= kd_b */

   /* ks_r/g/b are used only when kQATextureOp_Highlight is set */

   float   ks_r;   /* Red specular highlight, 0.0 <= ks_r <= 1.0 */
   float   ks_g;   /* Green specular highlight, 0.0 <= ks_g <= 1.0 */
   float   ks_b;   /* Blue specular highlight, 0.0 <= ks_b <= 1.0 */
} TQAVTexture;
```

The `x`, `y` and `z` fields have the same meaning as in the `TQAVGouraud` data structure.

The `uOverW`, `vOverW` and `invW` fields specify the u, v coordinates of this vertex for texture mapping. Texture coordinates are always specified in perspective corrected form, i.e. divided by the homogenous correction factor w. For non-perspective rendering modes, `invw` should always be 1.0.

When using a drawing engines that supports `kQAOptional_PerspectiveZ`, and the state variable `kQATag_PerspectiveZ` is set to `kQAPerspectiveZ_On`, hidden surface removal is performed with `invW` instead of with `z`. See the discussion for `TQAVGouraud`, earlier in this chapter.

The remaining fields are used to specify texture map color modulation and highlight information, and to specify additional color information for use when decal rendering mode is

enabled. These fields are discussed in detail in Rendering with texture mapping, later in this chapter.

QADrawPoint

`QADrawPoint()` draws a single point to `TQADrawContext`. The size of the point is specified by the `kQATag_Width` state variable.

```
void QADrawPoint (
   const TQADrawContext *drawContext,  /* Draw context */
   const TQAVGouraud     *v);          /* Vertex */
```

QADrawLine

`QADrawLine()` draws a single line to `TQADrawContext`. The width of the line is specified by the `kQATag_Width` state variable. If different colors are specified by `v0` and `v1`, the line color will be smoothly interpolated.

```
void QADrawLine (
   const TQADrawContext *drawContext,  /* Draw context */
   const TQAVGouraud     *v0,          /* Vertex 0 */
   const TQAVGouraud     *v1);         /* Vertex 1 */
```

QADrawTriGouraud

`QADrawTriGouraud()` draws a single Gouraud shaded triangle to `TQADrawContext`. `flags` can be `kQATriFlags_None` or `kQATriFlags_Backfacing`. It's suggested (but not required) that an application set `kQATriFlags_Backfacing` for backfacing triangles, as this information may assist the drawing engine in resolving ambiguous hidden surface removal situations.

```
void QADrawTriGouraud (
   const TQADrawContext *drawContext,  /* Draw context */
   const TQAVGouraud     *v0,          /* Vertex 0 */
   const TQAVGouraud     *v1,          /* Vertex 1 */
   const TQAVGouraud     *v2,          /* Vertex 2 */
   unsigned long         flags);       /* Mask of kQATriFlags_xxx */
```

QADrawTriTexture

`QADrawTriTexture()` draws a single texture mapped triangle to `TQADrawContext`. The `TQATexture` selected by the `kQATag_Texture` state variable will be used as the texture map (see Chapter 5: kQATag_Texture). `flags` has the same function as for `QADrawTriGouraud()`, above.

```
void QADrawTriTexture (
   const TQADrawContext *drawContext,  /* Draw context */
   const TQAVTexture     *v0,          /* Vertex 0 */
   const TQAVTexture     *v1,          /* Vertex 1 */
   const TQAVTexture     *v2,          /* Vertex 2 */
   unsigned long         flags);       /* Mask of kQATriFlags_xxx */
```

QADrawVGouraud

`QADrawVGouraud()` draws a variable number of `TQAVGouraud` as either points, lines, or triangles, as selected by `vertexMode`. `flags` is either `NULL`, or points to an array of `kQATriFlags_None`/`kQATriFlags_Backfacing`.

```
void QADrawVGouraud (
   const TQADrawContext  *drawContext,    /* Draw context */
   unsigned long         nVertices,       /* Number of vertices */
   TQAVertexMode         vertexMode,      /* One of kQAVertexMode_xxx */
   const TQAVGouraud     vertices[],      /* Array of vertices */
   const unsigned long   flags[]);        /* Array of per-triangle flags
                                             (or NULL) */
```

`vertexMode` can be any of the following:

```
kQAVertexMode_Point   /* Draw nVertices points */
kQAVertexMode_Line    /* Draw nVertices/2 line segments */
kQAVertexMode_Polyline/* Draw nVertices-1 connected line segments */
kQAVertexMode_Tri     /* Draw nVertices/3 triangles */
kQAVertexMode_Strip   /* Draw nVertices-2 triangles as a strip */
kQAVertexMode_Fan     /* Draw nVertices-2 triangles as a fan from v0 */
```

QADrawVTexture

`QADrawVTexture()` draws a variable number of `TQAVTexture` as either points, lines, or triangles, as selected by `vertexMode`. `flags` is either `NULL`, or points to an array of `kQATriFlags_None`/`kQATriFlags_Backfacing`. `vertexMode` has the same meaning as for `QADrawVGouraud()`, above.

```
void QADrawVTexture (
   const TQADrawContext  *drawContext,    /* Draw context */
   unsigned long         nVertices,       /* Number of vertices */
   TQAVertexMode         vertexMode,      /* One of kQAVertexMode_xxx */
   const TQAVTexture     vertices[],      /* Array of vertices */
   const unsigned long   flags[]);        /* Array of per-triangle flags
                                             (or NULL) */
```

If the drawing engine supports `kQAOptional_OpenGL`, `QADrawVTexture()` can be used to draw texture mapped points and lines. If the drawing engine does not support `kQAOptional_OpenGL`, the point and line `vertexModes` are no-ops.

TriMesh

RAVE supports a drawing type called "TriMesh." The main reason for using this data type is to save memory and processing time by sharing vertices between triangles.

◊ Note for engine developers: If a drawing engine does not support the TriMesh then the manager will decompose it into individual triangles. Therefore, the TriMesh methods only need to be supported if a drawing engine can take advantage of vertex sharing.

The process to draw a TriMesh is to first submit all the vertices that are going to be used by calling either `QASubmitVerticesGouraud()` or

QASubmitVerticesTexture(). This establishes the current state of active vertices which can be referenced by later calls to QADrawTriMeshGouraud() or QADrawTriMeshTexture(). Note that there is no unsubmit call, it is up to the engine to handle memory management in whatever way is appropriate.

Once the vertices have been submitted, triangles are drawn by calls to QADrawTriMeshGouraud() or QADrawTriMeshTexture(). The following details each API call.

QASubmitVerticesGouraud
QASubmitVerticesTexture

QASubmitVerticesGouraud() and QASubmitVerticesTexture() submit the given list of vertices to the engine. These functions will not draw anything. Note that there is a separate state for the gouraud and texture vertices so that calls to QADrawTriMeshGouraud() will use vertices submitted by QASubmitVerticesGouraud() and calls to QADrawTriMeshTexture() will use vertices submitted by QASubmitVerticesTexture().

◊ The application is resposible for allocating all memory for the vertices passed into these calls. In addition the application must make sure that this memory is valid until all drawing is completed.

```
void QASubmitVerticesGouraud (
   const TQADrawContext     *drawContext,    /* Draw context */
   unsigned long            nVertices,       /* Number of vertices */
   const TQAVGouraud        *vertices);      /* Array of vertices */

void QASubmitVerticesTexture (
   const TQADrawContext     *drawContext,    /* Draw context */
   unsigned long            nVertices,       /* Number of vertices */
   const TQAVTexture        *vertices);      /* Array of vertices */
```

QADrawTriMeshGouraud
QADrawTriMeshTexture

QADrawTriMeshGouraud() and QADrawTriMeshTexture() take a list of triangles and draws them. Each triangle is of type TQAIndexedTriangle which has a flag field and three indices that reference into the current state of vertices submitted by QASubmitVerticesGouraud() or QASubmitVerticesTexture().

```
typedef struct TQAIndexedTriangle  /* A single tri for QADrawTriMesh */
{
   unsigned long   triangleFlags;  /* Tri flags, see kQATriFlags_ */
   unsigned long   vertices[3];    /* Indices into a vertex array */
} TQAIndexedTriangle;

void QADrawTriMeshGouraud (
   const TQADrawContext    *drawContext,   /* Draw context */
   unsigned long           nTriangles,     /* Number of triangles */
   const TQAIndexedTriangle   *triangles);  /* Array of triangles */

void QADrawTriMeshTexture (
   const TQADrawContext    *drawContext,   /* Draw context */
   unsigned long           nTriangles,     /* Number of triangles */
   const TQAIndexedTriangle   *triangles);  /* Array of triangles */
```

QADrawBitmap

`QADrawBitmap()` draws a bitmap to `TQADrawContext`. The `bitmap` parameter is a `TQABitmap` pointer returned by `QABitmapNew()`; see Chapter 7: Creating Textures and Bitmaps for more information.

Unlike the other drawing calls, `QADrawBitmap()` accepts negative `x` or `y` values for the pixel position. This is so the upper-left corner of the bitmap can be positioned to the left or above the upper-left corner of the `TQADrawContext`. This allows a bitmap to be smoothly moved off any edge of the draw context rectangle. (When the bitmap boundary extends outside the draw context, it is the drawing engine's responsibility to clip it appropriately.)

```
void QADrawBitmap (
   const TQADrawContext *drawContext,
   const TQAVGouraud     *v,        /* xyz, and (if 1 bit/pixel) argb */
   TQABitmap             *bitmap);  /* Allocated by QABitmapNew() */
```

Rendering with transparency

A RAVE drawing engine that sets the `kQAOptional_Blend` flag supports two transparency blending models, *pre-multiplied* and *interpolated*. The functions for these two transparency models, assuming a back-to-front drawing order, are shown below. (`Dst` indicates the data previously in the frame buffer, `Src` indicates the new incoming data.)

Pre-multiplied

```
a = 1 – (1 – aSrc) * (1 – aDst)
r = rSrc + (1 - aSrc) * rDst
g = gSrc + (1 - aSrc) * gDst
b = bSrc + (1 - aSrc) * bDst
```

Interpolated

```
a = 1 – (1 – aSrc) * (1 – aDst)
r = aSrc * rSrc + (1 - aSrc) * rDst
g = aSrc * gSrc + (1 - aSrc) * gDst
b = aSrc * bSrc + (1 - aSrc) * bDst
```

The `kQATag_Blend` state variable selects the current transparency model; the default value is `kQABlend_PreMultiply`. The models differ only in the function for `r`, `g` and `b`; the `a` function is identical.

◊ Drawing engines that don't set the `kQAOptional_ZSorted` flag require that the transparent objects be submitted by the application in back-to-front Z order (otherwise the

blend functions shown above will not yield the correct results). Typically an application renders all opaque objects first, then submits the transparent objects in back-to-front order.

[Drawing engines that set the `kQAOptional_ZSorted` flag perform this Z sorting automatically; they are discussed later in this chapter.]

The pre-multiplied transparency model is the recommended model for rendering shaded transparent 3D primitives such as triangles. Because the pre-multiplied model does not scale `rSrc`, `gSrc` and `bSrc` by `aSrc`, it allows a transparent object to have a specular highlight amplitude greater than its alpha value. For example, a sheet of glass might pass 99% of the light behind it, indicating an alpha value of 0.01. However, that same glass could contribute a specular highlight much greater than 0.01 — 0.5 would not be uncommon. The pre-multiplied transparency model allows this object to be rendered correctly.

The interpolated transparency model (perhaps the more familiar one) is chosen by setting `kQATag_Blend` to `kQABlend_Interpolate`. This mode is less suitable for rendering shaded transparent objects, but is very effective for compositing bitmap images.

Z sorted transparency

◊ Drawing engines that set the `kQAOptional_ZSorted` flag do *not* require that transparent objects be submitted in back-to-front order. Engines of this type may either implement a rendering algorithm, such as a painter's or scanline algorithm, that naturally rasterizes objects in Z sorted order, or they may store and sort non-opaque objects themselves. Engines of this type may even correctly blend intersecting transparent objects, although this is not a required feature.

Drawing engines of this type may operate in either back-to-front or front-to-back Z order. In either case, it is the drawing engine's responsibility to implement transparency blending operations that are equivalent to the pre-multiplied and interpolated modes described in the previous section. The application's use of these modes should be unaffected, except that it is no longer the application's responsibility to submit objects in back-to-front order.

OpenGL blending modes

Drawing engines that set the `kQAOptional_OpenGL` flag support a wide variety of other blending modes, best described by official OpenGL documentation.

Rendering with texture mapping

RAVE supports a powerful texture map rendering model that allows very realistic rendering of a wide variety of texture mapped material types. RAVE's texture map rendering modes vary between extremely simple (`kQATextureOp_None`), to an advanced diffuse color mapping model (`kQATextureOp_Highlight`). These modes are described in detail in the following sections.

[If you want to skip some reading, The complete texture mapping model, later in this chapter, summarizes the entire texture mapping model in pseudo-code.]

kQATextureOp_None

Before diving into the fancy stuff, let's begin with the basics. RAVE's texture mapping mode is controlled by the `kQATag_TextureOp` state variable. By default, this variable is set to `kQATextureOp_None`. In this mode, RAVE performs the most basic texture operation possible, simply replacing the object color with the texture map color. The pseudo-code below demonstrates this mode:

```
aSrc = a * TextureLookUp(u,v).a;      /* Or opacity test */
rSrc = TextureLookUp(u,v).r;
gSrc = TextureLookUp(u,v).g;
bSrc = TextureLookUp(u,v).b;
```

`u` and `v` are the perspective corrected texture coordinates after interpolation, and `a` is the interpolated alpha value (see TQAVTexture data type, earlier in this chapter). `aSrc`, `rSrc`, `gSrc` and `bSrc` are the texture mapped object color — this `Src` color is then passed through transparency blending to generate the final pixel color .

Because there is no modulation of the texture map color, this mode creates a flat looking image with no lighting effects. This is most useful when the texture mapping engine is being used as a 2D image warping engine (e.g. for video effects). However, it doesn't create a realistic 3D rendering.

◊ In this mode, the texture map's alpha channel is used to control the transparency of the rendered object on a pixel-by-pixel basis. As shown above, the resulting pixel alpha is the product of the texture alpha and the vertex alpha (which is interpolated from the `TQAVTexture` data).

◊ To reduce cost, a drawing engine may choose to implement the alpha multiply as an opacity test, rather than a multiply, i.e.:

```
aSrc = (TextureLookUp(u,v).a == 1) ? a : 0;
```

kQATextureOp_Modulate

A more realistic texture mapped image can be obtained by modulating the texture map color with `kd_r`, `kd_g` and `kd_b` (from `TQAVTexture`). This mode is enabled by setting the `kQATextureOp_Modulate` flag in the `kQATag_TextureOp` state variable. The equations below show the effect of modulation:

```
aSrc =    a * TextureLookUp(u,v).a;      /* Or opacity test */
rSrc = kd_r * TextureLookUp(u,v).r;
gSrc = kd_g * TextureLookUp(u,v).g;
bSrc = kd_b * TextureLookUp(u,v).b;
```

Usually modulation is performed to add the effect of lighting to the texture, i.e. `kd_r`, `kd_g` and `kd_b` are the illumination brightness. Note that `kd_r`, `kd_g` and `kd_b` can have a value

greater than 1.0. This allows a more accurate rendering of scenes where the light intensity is high.

◊ To reduce cost, a drawing engine may replace the three modulation components `kd_r`, `kd_g` and `kd_b` with a single value `kd`. This replacement is transparent to the application, except that colored lights applied to texture maps will appear white. Drawing engines that use this simplification must negate the `kQAOptional_TextureColor` flag.

kQATextureOp_Highlight

Image realism can be further improved by setting the `kQATextureOp_Highlight` flag. When both `kQATextureOp_Modulate` and `kQATextureOp_Highlight` are `true`, the texture operation is:

```
aSrc =    a * TextureLookUp(u,v).a;      /* Or opacity test */
rSrc = kd_r * TextureLookUp(u,v).r + ks_r;
gSrc = kd_g * TextureLookUp(u,v).g + ks_g;
bSrc = kd_b * TextureLookUp(u,v).b + ks_b;
```

The `ks_r`, `ks_g` and `ks_b` values are used to add a specular highlight to the texture mapped object. In fact, the equations shown above bear a strong resemblance to the classic phong illumination model: `kd_r/g/b` is the diffuse light, `TextureLookUp()` is the diffuse color, and `ks_r/g/b` is the product of specular light and specular color. This mode can be described as *diffuse color mapping*.

◊ To reduce cost, a drawing engine may replace the three specular highlight components `ks_r`, `ks_g` and `ks_b` with a single value `ks`. This replacement is transparent to the application, except that the specular highlight of texture mapped objects will always be white, not colored. Drawing engines that use this simplification must negate the `kQAOptional_TextureColor` flag.

kQATextureOp_Decal

In the previous examples, the texture map alpha channel (multiplied by the vertex alpha) provides the transparency of the texture mapped object. This allows the texture map alpha to control the transparency of the object on a pixel-by-pixel basis.

◊ Setting `kQATextureOp_Decal` changes the interpretation of texture map alpha. When `kQATextureOp_Decal` is true, the texture map alpha is used to blend between the texture map color and the interpolated `r`, `g`, and `b` fields from `TQAVTexture`.

```
aT = TextureLookUp(u,v).a;
rSrc = aT * TextureLookUp(u,v).r + (1 - aT) * r;  /* Or opacity test */
gSrc = aT * TextureLookUp(u,v).g + (1 - aT) * g;
bSrc = aT * TextureLookUp(u,v).b + (1 - aT) * b;
aSrc = a;
```

◊ To reduce cost, a drawing engine may choose to implement these alpha blends as opacity tests, i.e.:

```
rSrc = (aT == 1) ? TextureLookUp(u,v).r : r;
```

The complete texture mapping model

If you find reading all this text boring, here's a much more concise description. The following pseudo-code demonstrates the complete texture mapping model. Features that can be simplified for cost reduction are noted in the right side comments.

```
/* Begin by looking up argb from the texture map */
aSrc = TextureLookUp(u,v).a;
rSrc = TextureLookUp(u,v).r;
gSrc = TextureLookUp(u,v).g;
bSrc = TextureLookUp(u,v).b;

if (stateTextureOp & kQATextureOp_Decal)
{
   rSrc = aSrc * rSrc + (1 - aSrc) * r;  /* Or opacity test */
   gSrc = aSrc * gSrc + (1 - aSrc) * g;
   bSrc = aSrc * bSrc + (1 - aSrc) * b;
   aSrc = a;
}
else
{
   aSrc = aSrc * a;                      /* Or opacity test */
}

if (stateTextureOp & kQATextureOp_Modulate)
{
   rSrc *= kd_r;                         /* Or kd replaces kd_r/g/b */
   gSrc *= kd_g;
   bSrc *= kd_b;
}

if (stateTextureOp & kQATextureOp_Highlight)
{
   rSrc += ks_r;                         /* Or ks replaces ks_r/g/b */
   gSrc += ks_g;
   bSrc += ks_b;
}
/* And proceed with transparency blending */
```

kQATextureOp_Shrink

Unlike the texture modes described previously, `kQATextureOp_Shrink` does not affect the per-pixel texture mapping algorithm. Instead, the application sets `kQATextureOp_Shrink` to avoid unwanted texture wrapping. Setting `kQATextureOp_Shrink` to true indicates that the drawing engine should guarantee that a uv range of 0.0 < = uv < = 1.0 will not cause wrapping.

In theory, a uv range of 0.0 - 1.0 should not cause wrapping anyway. However, in practice the errors that occur during uv interpolation can cause overflow or underflow of u and v, resulting in occasional one pixel texture wraps at the 0 and 1 boundaries. Setting `kQATextureOp_Shrink` indicates that these errors should be suppressed.

◊ `kQATextureOp_Shrink` is not the same as uv clamping in OpenGL. The difference is that clamping is designed to accept uv over an arbitrary range, while `kQATextureOp_Shrink` is only effective over an input uv range of 0-1. This means `kQATextureOp_Shrink` is less expensive to implement — usually it can be performed

by slightly compressing the range of u and v before interpolation begins, rather than by implementing per-pixel clamp tests. However, drawing engines that do support OpenGL-style clamping can use this feature to implement `kQATextureOp_Shrink`.

`kQATextureOp_Shrink` is typically used by applications which perform uv clamping by geometry subdivision (rather than by per-pixel clamping).

Using the texture map alpha channel for transparency

Texture maps of pixel types `kQAPixel_ARGB16` or `kQAPixel_ARGB32` include a per-pixel alpha channel. When the alpha blending mode is set to `kQABlend_PreMultiply`, this alpha channel can be used to control object transparency on a pixel-by-pixel basis.

◊ Because this transparency model assumes that diffuse color has been pre-multiplied by alpha, *every pixel of the texture map must be pre-multiplied by its associated alpha value* before the texture map is created with QATextureMapNew().

◊ This transparency mode models a transparent material (such as glass). For these types of materials the specular highlight is unaffected by the diffuse transparency of the object. In other words, setting the alpha channel of the texture to 0 will not make the object vanish — its specular highlight will still be rendered.

Using the texture map alpha channel as a matte

Setting the transparency blending model to `kQABlend_Interpolate` allows the per-pixel texture map alpha channel to be used as a matte that "cuts out" portions of the drawn geometry. With this blending mode, a per-pixel alpha value of 0 will completely eliminate the rendered object (including its specular highlight).

◊ Note that the texture pixels' diffuse colors should *not* be pre-multiplied by their associated alpha value. This multiplication will be performed by the blending operation.

◊ In this mode, the alpha channel operates as a soft-edge matte. Unfortunately, this means that per-vertex interpolated alpha cannot be used to model a transparent surface as accurately, as the specular highlight will be scaled by the alpha value. In some cases this can be corrected by increasing the brightness of the specular highlight when per-vertex alpha is used. However, for the general case the current specification doesn't provide a method for simultaneously rendering accurate transparency while using the texture alpha channel as a matte. [That's because it would require a lot more hardware! -Eds.]

Rendering with antialiasing

Drawing engines that set the `kQAOptional_Antialias` flag support antialiased rendering. The application indicates its preferred level of antialiasing with the `kQATag_Antialias` state variable (see Chapter 5); however, the interpretation of this

variable is the drawing engine's responsibility. For example, consider a drawing engine that supports antialiased line drawing with no speed penalty, but that slows down 50% when triangle antialiasing is enabled. For this engine, setting `kQATag_Antialias` to `kQAAntialias_Fast` will enable line antialiasing. However, triangle antialiasing will not be enabled until `kQATag_Antialias` is set to `kQAAntialias_Mid`.

◊ In RAVE, antialiasing operates *independently* of the transparency blending mode. This is in contrast to OpenGL, where specific blending modes must be selected when antialiasing is enabled.

Using color lookup tables

RAVE supports two pixels types defined in `TQAImagePixelType` for using color tables, `kQAPixel_CL4` and `kQAPixel_CL8`. These additional pixel types are only valid when creating a texture or bitmap. The concept of a color table has also been added to the API to support these new pixel types. `TQAColorTableType` defines the type of color tables supported by the API and `TQAColorTable` represents an actual color table.

```
typedef enum TQAImagePixelType
{
   kQAPixel_Alpha1    = 0,    /* 1 bit/pixel alpha */
   kQAPixel_RGB16     = 1,    /* 16 bit/pixel, R=14:10, G=9:5, B=4:0 */
   kQAPixel_ARGB16    = 2,    /* 16 bit/pixel, A=15, R=14:10, G=9:5,
B=4:0 */
   kQAPixel_RGB32     = 3,    /* 32 bit/pixel, R=23:16, G=15:8, B=7:0 */
   kQAPixel_ARGB32    = 4,    /* 32 bit/pixel, A=31:24, R=23:16, G=15:8,
B=7:0 */
   kQAPixel_CL4       = 5,    /* 4 bit color look up table, always big
endian, ie high 4 bits effect left pixel */
   kQAPixel_CL8       = 6        /* 8 bit color look up table */
} TQAImagePixelType;

typedef enum TQAColorTableType
{
   kQAColorTable_CL8_RGB32    = 0,     /* 256 entry, 32 bit/pixel,
R=23:16, G=15:8, B=7:0 */
   kQAColorTable_CL4_RGB32    = 1      /* 16 entry, 32 bit/pixel,
R=23:16, G=15:8, B=7:0 */
} TQAColorTableType;

typedef struct  TQAColorTable TQAColorTable;
```

Before using these pixel types you must check the gestalt flags `kQAOptional_CL8` and `kQAOptional_CL4` to see if they are supported by the current drawing engine. Additionally you may want to check `kQAFast_CL8` and `kQAFast_CL4` to see if they are accelerated. If they are not accelerated it is safe to assume that the texture or bitmap will be expanded to a pixel type that the drawing engine can render directly.

◊ Note to engine developers: Support for the color lookup table is not required, however the manager will NOT provide support if the engine does not support it. This means that if an

application tries to create a texture or bitmap that requires a color table and your engine does not support color tables the call will return the error `kQANotSupported`.

After creating a texture or bitmap of type `kQAPixel_CL4` or `kQAPixel_CL8` you must "bind" a color table created by `QAColorTableNew` to the new object by calling either `QATextureBindColorTable` or `QABitmapBindColorTable`. When the object is drawn the most recently bound color table will be used as the source color data.

Note that a color table must be bound before an object of pixel type `kQAPixel_CL4` or `kQAPixel_CL8` can be drawn. Additionally it is an error to bind a color table of a different size then the pixel type by which it was created. For example a texture created with `kQAPixel_CL4` can only be bound to a color table created with `kQACL4_RGB32`.

The only currently supported format for a color table is RGB32. The engine may color space reduce this data in order to fit in on-chip memory. When creating a color table you may optionally specify that index 0 is completely transparent by setting the `transparentIndexFlag` to true.

QAColorTableNew

```
TQAError QAColorTableNew(
   const TQAEngine    *engine,      /* Drawing engine to use */
   TQAColorTableType  tableType,    /* Depth, color space, etc. */
   void               *pixelData,   /* lookup table entries in pixelType
format */
   long               transparentIndexFlag, /* boolean, false means no
transparency, true means index 0 is transparent */
   TQAColorTable      **newTable);  /* (Out) Newly created TQAColorTable
*/
```

QAColorTableDelete

```
void QAColorTableDelete(
   const TQAEngine    *engine,         /* Drawing engine to use */
   TQAColorTable      *colorTable);      /* Previously allocated by
QAColorTableNew() */
```

QATextureBindColorTable

```
TQAError QATextureBindColorTable(
   const TQAEngine    *engine,         /* Drawing engine to use */
   TQATexture         *texture,        /* Previously allocated by
QATextureNew() */
   TQAColorTable      *colorTable);    /* Previously allocated by
QAColorTableNew() */
```

QABitmapBindColorTable

```
TQAError QABitmapBindColorTable(
   const TQAEngine    *engine,         /* Drawing engine to use */
   TQABitmap       *bitmap,         /* Previously allocated by
QABitmapNew() */
   TQAColorTable      *colorTable);      /* Previously allocated by
QAColorTableNew() */
```

Chapter 7: Creating Textures and Bitmaps

In some cases a drawing engine may need to store textures and bitmaps in special purpose memory (e.g. on an accelerator card), rather

than in general purpose system memory. To support this, RAVE provides `new` and `delete` functions for textures and bitmaps. These functions provide an opportunity for the drawing engine to copy the data into special purpose memory if necessary(or to perform any other required setup).

Although these functions allow a drawing engine to copy textures and bitmaps into special purpose memory, the API does not *require* that copying be performed. This avoids penalizing drawing engines (such as software rasterizers) which can directly use the application's texture or bitmap information in system memory.

The TQAImage

Both texture maps and bitmaps are composed of pixel images. The RAVE API describes these images with a general-purpose datatype called `TQAImage`:

```
struct TQAImage
{
   long     width;                  /* Width of pixmap */
   long     height;                 /* Height of pixmap */
   long     rowBytes;               /* Rowbytes of pixmap */
   void     *pixmap;                /* Pixmap */
};
```

A bitmap image, or non-mipmapped texture map, is described by a single `TQAImage`. A mipmapped texture is described by an array of `TQAImage`, one for each map page.

◊ For some low-cost accelerators, having `rowBytes` = width * sizeof (pixel) will improve performance.

QATextureNew

`QATextureNew()` is used to create a texture map. `QATextureNew()` sets a `TQATexture` pointer. This `TQATexture` pointer is used by the application to select a texture map during rendering (see Chapter 5: kQATag_Texture), and in subsequent calls to `QATextureDelete()` or `QATextureDetach()`.

```
TQAError QATextureNew (
   const TQAEngine    *engine,        /* Drawing engine to use */
   unsigned long      flags,          /* Mask of kQATexture_xxx flags */
   TQAImagePixelType  pixelType,      /* Depth, color space, etc. */
   const TQAImage     images[],       /* Image(s) for texture */
   TQATexture         **newTexture);  /* (Out) New TQATexture */
```

`engine` is the drawing engine with which this texture will be used.

`flags` is a bitmask for which any combination of the following flags may be ORed together:

`kQATexture_Lock`: Load this texture and do not allow it to be swapped out. Usually this is used by the application to improve performance by locking a texture which will be heavily used. For software drawing engines this is usually a no-op. Warning: If the drawing engine cannot meet this request, `QATextureNew()` will fail and return an error.

`kQATexture_Mipmap`: This is a mipmapped texture. See below for further discussion.

`pixelType` indicates the pixel format of the images. It can be any of:

`kQAPixel_RGB16`: A 16 bit/pixel map. Red is bits [14:10], green is [9:5], blue is [4:0]. There is no per-pixel alpha value, so the texture is treated as opaque (although transparency can still be applied via the triangle's vertex alpha values).

`kQAPixel_ARGB16`: Same as `kQAPixel_RGB16`, above, except that bit [15] is used as a per-pixel alpha value. Because it is one bit, alpha can be either 0 or 1. When alpha is 1, the texture is opaque; when alpha is 0, it is completely transparent.

`kQAPixel_RGB32`: A 32 bit/pixel map. Red is bits [23:16], green is [15:8], blue is [7:0]. There is no per-pixel alpha value, so the texture is treated as opaque (just like kQAPixel_RGB16).

`kQAPixel_ARGB32`: Same as `kQAPixel_RGB32`, above, except that bits [31:24] are used as an eight bit per-pixel alpha. An alpha of 255 is opaque, while an alpha of 0 is completely transparent.

`kQAPixel_CL4`: A 4 bit color lookup table indexed texture. See "Using color lookup tables" above.

`kQAPixel_CL8`: A 8 bit color lookup table indexed texture. See "Using color lookup tables" above.

`images` is an array of one or more `TQAImage` structures that point to the texture image. When `kQATexture_Mipmap` is false, `images` points to a single `TQAImage` which defines the texture map. Both the `width` and `height` of the `TQAImage` must be an even power of 2, e.g. 64, 128, 256 etc.

When `kQATexture_Mipmap` is true, `images` points to an *array* of `TQAImage`, one for each page of the texture mipmap. `images[0]` is the highest resolution page; its width and height must be an even power of 2. Each subsequent `TQAImage` should have a width and height 1/2 the value of the previous page, with the exception that width and height have a minimum value of 1. The table below gives example `TQAImage` resolutions for a 64x16 mipmapped texture:

	Width	*Height*
images[0]	64	16
images[1]	32	8
images[2]	16	4
images[3]	8	2
images[4]	4	1
images[5]	2	1
images[6]	1	1

`newTexture` is a pointer to your `TQATexture` pointer. If `QATextureNew()` returns `kQANoErr`, `newTexture` will be set to point to the new `TQATexture`.

◊ It is *not* required that `QATextureNew()` copy the pixmap data pointed to by `images`. Therefore, after calling `QATextureNew()`, the application *must not free or reuse the memory which holds the image pixmaps*. If the application needs to free or reuse the image pixmap memory, it must call `QATextureDetach()`, described later in this chapter, before doing so.

◊ Although it isn't required that the image pixmap memory be copied, `QATextureNew()` *is* required to copy all necessary information from the `TQAImage` structures themselves. Therefore the application can free or reuse this memory after the `QATextureNew()` call.

For example, to create a 128x256 non-mipmapped texture with 32 bit RGB pixels:

```
TQAEngine     *engine;
TQATexture    *texture;
TQAImage      image;
long          pixmap [256][128];
...
image.width = 128;
image.height = 256;
image.rowBytes = image.width * sizeof (long);
image.pixmap = pixmap;

if (QATextureNew (engine, kQATexture_None, kQAPixel_RGB32,
        &image, &texture) != kQANoErr)
{
   /* Error, map could not be created. */
}
/* 'image' can now be changed, but _not_ 'pixmap'! */
```

`QATextureNew()` returns `kQANotSupported` if the requested pixel type is not supported. `kQAOutOfMemory` is returned if there isn't enough memory. `kQAError` will be returned for other errors, e.g. the texture could not be locked.

QATextureDelete

`QATextureDelete()` is used to delete a `TQATexture`:

```
void QATextureDelete (
   const TQAEngine    *engine,      /* Drawing engine */
   TQATexture         *texture);    /* Created by QATextureNew() */
```

For example, to delete the texture created in the previous example:

```
QATextureDelete (engine, texture);
```

QATextureDetach

`QATextureDetach()` forces the drawing engine to copy the pixmap data which was originally provided to it by `QATextureNew()`. Once `QATextureDetach()` has been called, the pixmap data for the texture's images can be freed or reused safely.

```
TQAError QATextureDetach (
   const TQAEngine       *engine,      /* Drawing engine */
   TQATexture            *texture);    /* Created by QATextureNew() */
```

Because `QATextureDetach()` may have to allocate memory, it returns `TQAError` to indicate success or failure. If the return value is not `kQANoErr`, then the texture was not successfully detached.

QABitmapNew

`QABitmapNew()` is used to create a bitmap. `QABitmapNew()` sets a `TQABitmap` pointer. This `TQABitmap`pointer is used to render the bitmap with `QADrawBitmap()`, and in subsequent calls to `QABitmapDelete()` or `QABitmapDetach()`.

```
TQAError QABitmapNew (
   const TQAEngine    *engine,        /* Drawing engine to use */
   unsigned long      flags,          /* Mask of kQABitmap_xxx flags */
   TQAImagePixelType  pixelType,      /* Depth, color space, etc. */
   const TQAImage     *image,         /* Image */
   TQABitmap          **newBitmap);   /* (Out) New TQABitmap */
```

`engine` is the drawing engine with which this bitmap will be used.

`flags` is a bitmask. Currently, only one flag is defined:

> `kQABitmap_Lock`: Load this bitmap and do not allow it to be swapped out. Usually this is used by the application to improve performance by locking a bitmap which will be heavily used. For software drawing engines this is usually a no-op. Warning: If the drawing engine cannot meet this request, `QABitmapNew()` will fail and return an error.

`pixelType` indicates the pixel format of the images. It can be any of:

`kQAPixel_Alpha1`: A 1 bit/pixel bitmap. Bits that are 0 are fully transparent; bits that are 1 are rendered in the color passed to `QADrawBitmap()`.

`kQAPixel_RGB16`: A 16 bit/pixel map. Red is bits [14:10], green is [9:5], blue is [4:0]. There is no per-pixel alpha value, so the texture is treated as opaque (although transparency can still be applied via the triangle's vertex alpha values).

`kQAPixel_ARGB16`: Same as `kQAPixel_RGB16`, above, except that bit 15 is used as a per-pixel alpha value. Because it is one bit, alpha can be either 0 or 1. When alpha is 1, the texture is opaque; when alpha is 0, it is completely transparent.

`kQAPixel_RGB32`: A 32 bit/pixel map. Red is bits [23:16], green is [15:8], blue is [7:0]. There is no per-pixel alpha value, so the texture is treated as opaque (just like kQAPixel_RGB16).

`kQAPixel_ARGB32`: Same as `kQAPixel_RGB32`, above, except that bits 31:24 are used as an eight bit per-pixel alpha. An alpha of 255 is opaque, while an alpha of 0 is completely transparent.

`image` points to a single TQAImage which defines the bitmap. Width and height may have any value greater than 0.

`newBitmap` is a pointer to your `TQABitmap` pointer. If `QABitmapNew()` returns `kQANoErr`, `newBitmap` will be set to point to the new `TQABitmap` .

◊ It is *not* required that `QABitmapNew()` copy the pixmap data pointed to by `image`. Therefore, after calling `QABitmapNew()`, the application *must not free or reuse the memory which holds the image pixmap*. If the application needs to free or reuse the image pixmap memory, it must call `QABitmapDetach()`, described later in this chapter, before doing so.

◊ Although it isn't required that the image pixmap memory be copied, `QABitmapNew()` *is* required to copy all necessary information from the `TQAImage` structure itself. Therefore the application can free or reuse this memory after the `QABitmapNew()` call.

QABitmapDelete

`QABitmapDelete()` is used to delete a `TQABitmap`:

```
void QABitmapDelete (
   const TQAEngine    *engine,   /* Draw engine */
   TQABitmap          *bitmap);  /* Created by QABitmapNew() */
```

QABitmapDetach

`QABitmapDetach()` forces the drawing engine to copy the pixmap data which was originally provided to it by `QABitmapNew()`. Once `QABitmapDetach()` has been called, the pixmap data for the bitmap's image can be freed or reused safely.

```
TQAError QABitmapDetach (
   const TQAEngine    *engine,   /* Draw engine */
   TQABitmap          *bitmap);  /* Created by QABitmapNew() */
```

Because `QABitmapDetach()` may have to allocate memory, it returns `TQAError` to indicate success or failure. If the return value is not `kQANoErr`, then the bitmap was not successfully detached.

Chapter 8: Buffering and Synchronization

QARenderStart

`QARenderStart()` is used to initialize a `TQADrawContext` before rendering. This function must always be called before any `QADraw...()` calls are made.

```
void QARenderStart (
   const TQADrawContext *drawContext,    /* Draw context */
   const TQARect        *dirtyRect,      /* Minimum area to clear */
   const TQADrawContext *initialContext);/* Previously cached context */
```

When `initialContext` is `NULL`, `QARenderStart()` clears the `drawContext` `Z` buffer to 1.0, and the argb buffer to the values contained in the state variables `kQATag_ColorBG_a/r/g/b`. When `initialContext` is non-`NULL`, `drawContext` is initialized to the contents of `initialContext`. See Using TQADrawContext as a cache, later in this chapter, for more discussion of `initialContext`.

`dirtyRect` indicates the minimum area of the `drawContext` to initialize. With some drawing engines, setting `dirtyRect` to an area smaller than the entire draw context will improve performance by avoiding unnecessary re-initialization of the draw context. Note, however, that `dirtyRect` is only a hint — the drawing engine may choose to initialize the entire buffer anyway. Therefore, `dirtyRect` cannot be used to avoid clearing a region of the previous image, or to perform incremental rendering. Instead, effects like these should be performed with `initialContext`.

If `NULL` is passed for `dirtyRect`, the entire buffer will be initialized.

◊ When OpenGL rendering is being performed, `QARenderStart()` performs the function of `glClear()`. In this mode, `QARenderStart()` and `QARenderEnd()` are no longer required to occur in matched pairs, and `QADraw...()` commands may occur at any time.

QARenderEnd

`QARenderEnd()` signals the end of rendering to a `TQADrawContext`. For a double-buffered context, this displays the back buffer. For a single-buffered context, this causes a call to `QAFlush()`, discussed later in this chapter, and signals the drawing engine that rendering is complete. This signal is then used for releasing locks on framebuffer regions, removing cursor shields, etc.

After a call to `QARenderEnd()` has been made, no further `QADraw...()` calls can be made until `QARenderStart()` has been called again.

```
TQAError QARenderEnd (
   const TQADrawContext  *drawContext,    /* Draw context */
   const TQARect         *modifiedRect);  /* Minimum area to show */
```

`modifiedRect` indicates the minimum area of the `drawContext` back buffer to show. On some drawing engines, setting `modifiedRect` to an area smaller than the entire draw context may improve performance by avoiding unnecessary pixel copying. Note, however, that `modifiedRect` is only a hint — the drawing engine may choose to show the entire buffer anyway.

If `modifiedRect` is `NULL`, the entire back buffer is shown.

`QARenderEnd()` returns a `TQAError` value, which signals whether there have been any error since the previous call to `QARenderStart()`. If all rendering commands executed correctly, `kQANoErr` is returned. If *any* rendering call caused an error, an error code other than `kQANoErr` will be returned.

◊ If the return value does not equal `kQANoErr`, it indicates that an error occurred while rendering the frame. In this case, the application should assume that the rendered image is incorrect.

◊ Calling `QARenderEnd()` automatically causes a call to `QAFlush()`.

QARenderAbort

`QARenderAbort()` causes any asynchronous rendering operations in `drawContext` to be aborted immediately, and any queued commands to be discarded. `QARenderAbort()` replaces `QARenderEnd()` as a means of ending the render to a draw context — the application should not call both.

`QARenderAbort()` returns a `TQAError` value; see `QARenderEnd()` for a discussion of how to interpret this value.

```
TQAError QARenderAbort (
   const TQADrawContext  *drawContext);     /* Draw context */
```

QAFlush

RAVE permits a drawing engine to buffer as many drawing commands as desired. This means that, even when drawing to a single-buffered draw context, drawing an object does not guarantee that the object will become visible on the screen.

```
TQAError QAFlush (
   const TQADrawContext  *drawContext);     /* Draw context */
```

`QAFlush()` causes the drawing engine to begin rendering all buffered commands. `QAFlush()` is *not* a blocking call — calling `QAFlush()` does not guarantee that rendering of the buffered commands has completed, merely that it has begun. `QAFlush()`

does guarantee that all the buffered calls will be performed *eventually* — wait long enough, and the rendered image will be complete.

`QAFlush()` is typically used to occasionally update a long, single-buffered render, so that the user can see what progress has been made. `QAFlush()` has no visible effect on a double-buffered draw context, although it will initiate rendering to the back buffer.

`QAFlush()` returns a `TQAError` value; see `QARenderEnd()` for a discussion of how to interpret this value.

◊ Calling `QARenderEnd()` automatically causes a call to `QAFlush()`.

QASync

`QASync()` is functionally identical to `QAFlush()`, except that it is blocking — it doesn't return until all outstanding rendering commands have been completed.

```
TQAError QASync (
   const TQADrawContext  *drawContext);     /* Draw context */
```

`QASync()` should be called whenever *completion* of all rendering is necessary. For example, an application should call `QASync()` before reading the rendered image to save it to disk.

`QASync()` returns a `TQAError` value; see `QARenderEnd()` for a discussion of how to interpret this value.

Using TQADrawContext as a cache

To improve performance when a large percentage of the objects in a scene don't change from frame to frame, RAVE supports draw context *caching*. To use this feature, the application first builds a cache by creating a `TQADrawContext` with the `QAContext_Cache` flag, and then drawing the unchanging objects to that context. This cache can then be passed to `QARenderStart()` as the `initialContext` — basically this means the `QARenderStart()` will initialize the buffer to the image stored in the cache, rather than to a blank screen.

For example, consider a (rather trivial) application where two triangles, `t1` and `t2`, remain constant from frame to frame, but triangle `t3` changes every frame This could be coded as:

```
TQAVGouraud     t1[3], t2[3], t3[3];
TQADrawContext  *cache, *draw;
...
/* Create TQADrawContexts (we should be checking for errors!) */
QADrawContextNew (device, rect, NULL, engine, QAContext_Cache, &cache);
QADrawContextNew (device, rect, NULL, engine,
     QAContext_DoubleBuffer, &draw );

/* Create the cache context */
QARenderStart (cache, NULL, NULL);
QADrawTriGouraud (cache, &t1[0], &t1[1], &t1[2], kQATriFlags_None);
QADrawTriGouraud (cache, &t2[0], &t2[1], &t2[2], kQATriFlags_None);
QARenderEnd (cache, NULL);

/* Render a bunch of frames using the cache and moving tri3 only */
while (movingTriangle3)
{
   myMoveTri (t3);
   QARenderStart (draw, NULL, cache);
   QADrawTriGouraud (draw, &t3[0], &t3[1], &t3[2], kQATriFlags_None);
   QARenderEnd (draw, NULL);
}
```

A drawing engine is not required to support caching; if it doesn't, it should return `NULL` when the `QAContext_Cache` flag is passed to `QADrawContextNew()`.

◊ Cache contexts must be singled buffered, and must be created with the same `TQADevice` and `TQARect` parameters as the draw context with which they will be used.

Chapter 9: Adding a New Drawing Engine

Developing a new drawing engine and adding it to RAVE requires seven steps:

1: Write methods for the public calls in `TQADrawContext` (`setInt`, `setFloat`, `drawPoint` etc.). This methods are all prototyped in RAVE.h, e.g. the `setInt` method is a function pointer of type `TQASetInt`. Writing these methods is most of the work; fortunately, RAVE allows you to begin with a minimal feature set so you can get something running quickly.

2: Write a `TQADrawPrivateNew` and `TQADrawPrivateDelete` method for your drawing engine's private draw context data. This is where you store your state variables, and any other private data necessary for rendering a `TQADrawContext` . These methods will be called by the `QADrawContextNew()` and `QADrawContextDelete()` functions. Because these aren't public methods, their prototypes are in RAVE_system.h (which is used only for drawing engine development).

3: Write `TQATextureNew`, `TQATextureDetach`, `TQATextureDelete`, `TQABitmapNew`, `TQABitmapDetach`, and `TQABitmapDelete` methods. These prototypes are in Drive3D_system.h. These methods are called by their associated public functions in RAVE.h, e.g. `QATextureNew()` calls your `TQATextureNew` method.

4: Write a `TQAEngineGestalt` method for your engine; its functionality is the same as `QAEngineGestalt()`. You'll need to get an Apple-assigned `vendorID` number to service the `kQAGestalt_VendorID` request. The prototype is in Drive3D_system.h.

5: Write a `TQAEngineDeviceCheck` method. RAVE will call this method to determine which `TQADevices` your drawing engine supports. Yes, same place for the prototype.

6: Write a `TQAEngineGetMethod` method. The RAVE manager will call this method to retrieve your engine's methods during registration. This method is used only for the engine methods (the draw context methods are set by your `TQADrawPrivateNew` function).

7: Finally, build all your code as a shared library, and include an initialization call to `QARegisterEngine()`, to which you pass your `TQAEngineGetMethod` method. This call registers your drawing engine with the RAVE manager.

Apple's 3D drawing engine development kit includes the RAVE_system.h include file, and code examples for all of these steps. The remainder of this chapter provides more detail on each of the steps described above.

The public TQADrawContext methods

The public `TQADrawContext` structure, defined in RAVE.h, holds function pointers which point to your engine's drawing methods. These function pointers are called whenever the application uses one of the drawing macros, such as `QADrawPoint()`, defined in RAVE.h. `TQADrawContext` also has a pointer (`drawPrivate`) to your engine's private data for this `TQADrawContext`. There is also a `version` field, which is set by the RAVE manager. In future releases of RAVE, this field will be used to signal any additions to the `TQADrawContext` 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;
};
```

The `TQADrawContext` is passed as the first parameter to all of your draw context methods. This allows your functions to retrieve the `drawPrivate` pointer, to which all of your private data is attached. For most of your functions, the `TQADrawContext` pointer is passed as `const`. This indicates that your function must not alter any field of the `TQADrawContext` . Respect the `const` declaration — if you override it and change anything in the `TQADrawContext` structure, you will break many apps (including QuickDraw 3D).

Three functions receive the `TQADrawContext` without a `const` declaration: `QASetFloat()`, `QASetInt()` and `QASetPtr()`. These functions are permitted to change methods in the `TQADrawContext`. For example, this allows `TQASetInt()` to change the `drawTriTexture` method depending on the current state of the `kQATag_TextureOp` state variable.

◊ When your `TQASetInt/Float/Ptr()` methods need to change a draw context method pointer, they should call `QARegisterDrawMethod()`. This notifies the

manager that a method has been changed. Do *not* directly change the method pointer in the `TQADrawContext` structure.

To demonstrate how these methods are called, consider the following application code that draws a point to `drawContext`:

```
TQADrawContext   *drawContext;
TQAVGouraud      vertex;
...
QADrawPoint (drawContext, &vertex);
```

From this source code, the C preprocessor macro substitution generates the code shown below, which calls your engine's `drawPoint` method:

```
(drawContext)->drawPoint (drawContext, &vertex);
```

All of the method function pointers in `TQADrawContext` are defined in RAVE.h. For example, the `TQADrawPoint` function type is:

```
typedef void (*TQADrawPoint) (
   const TQADrawContext  *drawContext,    /* Draw context */
   const TQAVGouraud     *v);             /* Vertex */
```

◊ With two exceptions, your drawing engine must implement a function for all of the methods in `TQADrawContext`. The first exception is for drawing engines that don't support texture mapping. These don't need to provide `drawTriTexture` or `drawVTexture` methods. The second exception is for the `drawVGouraud` and `drawVTexture` methods. If your application does not provide these, the RAVE manager will insert generic functions that decompose these calls into multiple calls to your `drawPoint/Line/Tri` functions.

The pseudo-code below shows an example function `MyDrawPoint()`, which matches the `TQADrawPoint` template:

```
void MyDrawPoint (
   const TQADrawContext  *drawContext,    /* Draw context */
   const TQAVGouraud     *v)              /* Vertex */
{
   MyPrivateData      *myData;   /* Actual type of my private context */

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

   /* Call my Z-buffered pixel draw function with xyz and argb, and
    * also pass it the current Z function, which is stored in my
    * private draw context data structure. This isn't a complete
    * implementation! (I really should be checking kQATag_Width, for
    * example) */

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

The private TQADrawContext new and delete functions

Once you have written all your `TQADrawContext` public methods, the next step is to write a `TQADrawPrivateNew` method. This method is prototyped in Drive3D_system.h:

```
typedef TQAError (*TQADrawPrivateNew) (
   TQADrawContext  *newDrawContext,/* Draw context to initialize */
   const TQADevice *device,/* Target device */
   const TQARect   *rect,  /* Target rectangle (device coordinates) */
   const TQAClip   *clip,  /* 2D clip region (or NULL) */
   unsigned long   flags); /* Mask of kQAContext_xxx */
```

This method will be called by the RAVE manager when the application creates a new drawing context with `QADrawContextNew()`. The pseudo-code below shows an example function `MyDrawPrivateNew()`, which matches the `TQADrawPrivateNew` template:

```
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 drawContext->drawPrivate. */

   myData = MyDataNew (...);
   drawContext->drawPrivate = (TQADrawPrivate *) myData;
   if ( ! myData)
   {
      return (kQAOutOfMemory);
   }

   /* Set the method pointers of drawContext to point to my
    * draw methods. */

   newDrawContext->setFloat = MySetFloat;
   newDrawContext->setInt = MySetInt;
   ...
   return (kQANoErr);
}
```

RAVE initializes all the fields of the `TQADrawContext` to `NULL` before calling your `TQADrawPrivateNew` method. This allows RAVE to recognize and replace functions that you didn't initialize (e.g. `drawVGouraud` or `drawVTexture`).

◊ Because your drawing engine initializes the `TQADrawContext` methods, you can load different methods depending on the type of draw context being created. For example, you may have a different line drawing function for 16 bits/pixel than for 32 bits/pixel. By testing the depth of the target TQADevice and then loading the method that matches that depth, you can avoid having to test the display depth every time your line drawing code is called.

In addition to your TQADrawPrivateNew method, you must implement a TQADrawPrivateDelete method which is called by QADrawContextDelete(). This method must free any memory or resources allocated by your TQADrawPrivateNew. For example:

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

Your new and delete method function pointers (MyDrawPrivateNew() and MyDrawPrivateDelete(), in the examples above) are communicated to RAVE during the registration process; see Registering your drawing engine, later in this chapter, for more information.

Texture and Bitmap New/Detach/Delete

These methods are called by the public RAVE functions which manage textures and bitmaps, e.g. QATextureNew() will call your TQATextureNew method. You must always implement TQABitmapNew, TQABitmapDetach and TQABitmapDelete methods. If your drawing engine supports texture mapping, you must also implement TQATextureNew, TQATextureDetach, and TQATextureDelete.

These methods are functionally identical to the public functions, except they don't include the engine parameter (engine is necessary in the public call so the manager knows to which engine the call should be passed).

Like TQADrawPrivateNew and TQADrawPrivateDelete, these methods are communicated to RAVE during the registration process, discussed later.

Adding Gestalt

To allow the application to evaluate your drawing engine, you must register a TQAEngineGestalt method. This is prototyped in Drive3D_system.h:

```
typedef TQAError (*TQAEngineGestalt) (
   TQAGestaltSelector selector,  /* Gestalt parameter being requested */
   void               *response);/* Buffer that receives response */
```

The TQAEngineGestalt method is functionally identical to the QAEngineGestalt() function: It receives a selector, and returns a response.

For example, assume your drawing engine supports texture mapping, and accelerates Gouraud shading and line drawing. Apple has assigned you a vendorID of 5, and your internal engineID number is 1001. A suitable TQAEngineGestalt function would be:

```
TQAError MyEngineGestalt (
   TQAGestaltSelector selector,  /* Gestalt parameter being requested */
   void               *response) /* Buffer that receives response */
{
   const static char  *myEngineName = "My Engine Name";

   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);
}
```

◊ When `vendorID` and `engineID` are identical for two drawing engines, RAVE registers only the most recent version. This decision is made by examining the `revision` number from your `TQAEngineGestalt` method — a larger number is newer.

Your `TQAEngineGestalt` method function pointer is communicated to RAVE during drawing engine registration.

Checking the TQADevice

RAVE needs a method to determine if your drawing engine can draw to a specific TQADevice. This is performed by the `TQAEngineDeviceCheck` method, prototyped in Drive3D_system.h:

```
typedef TQAError (*TQAEngineDeviceCheck) (
   const TQADevice    *device);       /* Target device */
```

Your `TQAEngineDeviceCheck` method should simply return `kQANoErr` if you can draw to the indicated TQADevice, or `kQAError` if you can not.

Registering your drawing engine

Congratulations! You have now written all the RAVE methods required to register your drawing engine. The final step is to call `QARegisterEngine()`, a RAVE manager function prototyped in Drive3D_system.h:

```
TQAError QARegisterEngine (
   TQAEngineGetMethod    engineGetMethod);  /* getMethod method */
```

`engineGetMethod` is a function in your engine which the RAVE manager can query to retrieve your engine methods[1]. For the engine examples we've given in this section, a suitable `TQAEngineGetMethod` function would be:

```
TQAError MyEngineGetMethod (
   TQAEngineMethodTag    methodTag,       /* Method being requested */
   TQAEngineMethod       *method)         /* (Out) 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);
}
```

There are two ways you can register your engine. During your initial debug, you may find it convenient to link your drawing engine with your test application, rather than build it as a shared library. When working this way, you will need to explicitly call `QARegisterEngine()` as part of your application initialization code.

Once your drawing engine is stable, you can switch to building it as a shared library. When the RAVE manager shared library is loaded by an application, it searches for and loads RAVE

---

[1] The drawing methods do *not* use this mechanism; see the earlier section on TQADrawPrivateNew for more discussion.

drawing engines as part of its initialization process. To have RAVE load your drawing engine, you must:

- Build your drawing engine as a shared library.
- Set the creator of your shared library to 'tnsl'.
- Have your shared library's initialization method call `QARegisterEngine()`.
- Put your drawing engine either in the current folder (the first location searched), or the Extensions folder.

Chapter 10: Porting OpenGL Hardware

This chapter discusses some specific topics of interest to IHVs who are implementing a RAVE drawing engine for hardware based on an OpenGL rasterization model.

Transparency

RAVE supports two transparency models, pre-multiplied and interpolated[2]. The equations for these two blending modes are shown below (see Chapter 6: Rendering transparency for a more detailed discussion of these modes). `Dst` indicates the data previously in the frame buffer; `Src` indicates the new incoming data:

Pre-multiplied

```
a = 1 – (1 – aSrc) * (1 – aDst)
r = rSrc + (1 - aSrc) * rDst
g = gSrc + (1 - aSrc) * gDst
b = bSrc + (1 - aSrc) * bDst
```

Interpolated

```
a = 1 – (1 – aSrc) * (1 – aDst)
r = aSrc * rSrc + (1 - aSrc) * rDst
g = aSrc * gSrc + (1 - aSrc) * gDst
b = aSrc * bSrc + (1 - aSrc) * bDst
```

OpenGL provides both of the blend functions shown above for `rgb` — however, the `a` blend function (which is the same for both modes) is *not* supported. This means that neither of these transparency models can be directly implemented by OpenGL hardware.

◊ It is possible, however, to emulate the RAVE transparency models on OpenGL hardware. Two methods are described here, one for frame buffers that don't store an alpha channel, and one for frame buffers that do.

RGB blending only

Drawing engines which don't store an alpha channel can easily implement these transparency models by simply ignoring the alpha channel formula. RAVE's transparency modes are then equivalent to the following OpenGL blending modes:

Pre-multiplied:

```
glBlendFunc (GL_ONE, GL_ONE_MINUS_SRC_ALPHA);
```

Interpolated:

```
glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Drawing engines which use this method should *not* set the `kQAOptional_BlendAlpha` bit of the `optionalFeatures` parameter returned by `QAEngineGestalt()`. This indicates to the application that blending of the alpha channel is not supported.

---

[2] If ZcTag_Blend has been set to ZcBlend_OpenGL, blending is performed according to the OpenGL™ specification (presumably this won't cause any porting difficulty).

ARGB blending via multiple passes

It's possible to correctly perform the transparency blending function for both `rgb` and `a` by designing the drawing engine to rasterize each transparent object more than once, each time altering the blending mode, object alpha and buffer write masks. The pseudo-code below demonstrates this method:

```
/* First pass. Perform rgb blending. Disable Z buffer writes and
 * alpha channel writes during this pass */

glColorMask (true, true, true, false);   /* Disable alpha write */
glDepthMask (false);                     /* Disable Z write */
if (premultpliedTransparency)
{
   glBlendFunc (GL_ONE, GL_ONE_MINUS_SRC_ALPHA);
}
else
{
   glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
}
< render object >

/* Second pass. During this pass we set the frame buffer alpha
 * to (1-aDst)*(1-aSrc). This requires re-rendering the object
 * with its alpha changed to 1-a, and some creative use of the
 * blending modes */

glColorMask (false, false, false, true); /* Write alpha only */
glDepthMask (false);                     /* Disable Z write */
glBlendFunc (GL_ONE_MINUS_DST_ALPHA, GL_ZERO);
< render object with alpha replaced with 1-a >

/* Third pass replaces the (1-aDst)*(1-aSrc) result in the alpha channel
 * with the final result of 1-(1-aDst)*(1-aSrc). This requires
 * re-rendering the object with its alpha values changed to 1, and
 * yet more creative use of the blending modes. If desired, Z buffer
 * writes are enabled during this pass. */

glColorMask (false, false, false, true); /* Write alpha only */
glDepthMask (true);                      /* Enable Z write */
glBlendFunc (GL_ONE_MINUS_DST_ALPHA, GL_ZERO);
< render object with alpha replaced with 1 >
```

Texture mapping

RAVE supports several different texture mapping modes, the choice of which is controlled by the `kQATag_TextureOp` state variable. This bit mask variable can be set to any combination of the `kQATextureOp_Modulate`, `kQATextureOp_Highlight` and `kQATextureOp_Decal` flags. Chapter 6: Rendering with texture mapping provides a detailed description of these modes; the sections below describe how to emulate these modes on an OpenGL rasterizer.

kQATextureOp_Modulate

`kQATextureOp_Modulate` can be approximated by the GL_MODULATE mode, using the `kd_r`, `kd_g` and `kd_b` values from the `TQAVTexture` data as the modulating color. Note that there is a slight functional difference, as GL_MODULATE does not allow the modulating color magnitude to be greater than 1.0, a feature that RAVE supports to provide improved image realism.

◊ It is recommended that new hardware designs support a maximum modulation amplitude greater than 1.0 (2.0 seems to be a sufficient).

kQATextureOp_Highlight

`kQATextureOp_Highlight` can be emulated by performing two rendering passes. The first pass renders the texture mapped object (optionally with GL_MODULATE), while the second pass adds the highlight value. Pseudo-code for this is shown below:

```
/* First pass. Render the texture mapped object. If
 * kQATextureOp_Modulate is true, use the kd_r, kd_g and kd_b values
 * from the TQAVTexture data to modulate the texture color via
 * GL_MODULATE. Z buffer write is disabled during this pass. */

glDepthMask (false);                    /* Disable Z write */
< render texture mapped object >

/* Second pass. Re-render the object as Gouraud shaded, with
 * the ks_r, ks_g and ks_b values from the TQAVTexture data as the
 * object color. Re-enable the Z write during this pass, and
 * set the blend function to additive rendering. */

glDepthMask (true);                     /* Enable Z write */
glBlendFunc (GL_ONE, GL_ONE);           /* Add highlight color */
< render highlight color as a Gouraud shaded object >
```

kQATextureOp_Decal

When `kQATextureOp_Decal` is `true` and `kQATextureOp_Modulate` is `false`, the OpenGL GL_DECAL mode is equivalent to the RAVE `kQATextureOp_Decal` mode. If `kQATextureOp_Highlight` is `true`, an additional rendering pass will be required to add the highlight color, as shown in the previous section.

◊ Unfortunately, we have not yet been able to find a reasonable combination of OpenGL commands which accurately renders the case when *both* `kQATextureOp_Decal` and `kQATextureOp_Modulate` are `true`. Vendors should determine for themselves if there is some means of making their hardware implement this mode correctly. If not, we recommend that `kQATextureOp_Decal` take precedence over `kQATextureOp_Modulate` — if both are `true`, `kQATextureOp_Modulate` should be ignored.

◊ Note that ignoring `kQATextureOp_Modulate` is *not* the ideal solution — Apple recommends that vendors modify their future products to support these modes simultaneously.

[One simple modification to an OpenGL rasterizer that enables simultaneous `kQATextureOp_Decal` and `kQATextureOp_Modulate` is to provide a switch that inverts the alpha opacity test of the texture map. This allows two rendering passes to be performed, one for the pixels covered by the opaque regions of the texture map, and the other for the pixels which are rendered with the Gouraud interpolated color.]

Index