QuickDraw 3D RAVE 1.6

New Feature Specification

3/5/1999
WARNING: This document is not yet final.

QuickDraw 3D
Apple Computer

Stephen Luce (sluce@apple.com)
Robert Dierkes (dierkes.r@apple.com)
Brian Greenstone

TABLE OF CONTENTS

INTRODUCTION

This document assumes familiarity with RAVE 1.5 (see the RAVE 1.5 ERS), and the intended audience of this document includes RAVE engine developers and RAVE application developers. Some sections may be more relevant to one group or the other.

The feature list for RAVE 1.6 reflects the feedback from hardware developers and software developers who write engines for RAVE and program with RAVE. Note that all API additions are 100% backwards compatible and do not require current hardware engines to be revised to be compatible with RAVE 1.6. All API's that require new engine support have been made optional. You should note, however, that a few name changes have taken place that will require source code changes unless the C preprocessor macro `RAVE_OBSOLETE` is defined to be 1.

It is expected that most engines will be revised to support these new features. However, care should be taken by application developers to ensure backward compatibility with 1.5 engines that are currently shipping with most 3D hardware. For each of the new features in RAVE 1.6, a gestalt option has been added so that you will be able to determine if a RAVE engine supports any of the various 1.6 features.

IMPORTANT: Always test your code with the debug version of the RAVE manager. This will help to quickly reveal performance issues and other more serious problems.

PAGE FLIPPING VS. BLITTING

This section is a clarification of an issue from RAVE 1.5 and is not RAVE 1.6 specific.

Most of the time RAVE engines copy (or blit) the back buffer to the screen at the end of the rendering loop, however, this is not always the case. If a RAVE engine supports full screen page flipping and a full screen draw context is created page flipping may be enabled. This is an advantage because page flipping is (usually) faster than blitting. Some RAVE engines may have specialized hardware that allows them to do page flipping within a window.

This is an important issue to consider because after a blit the back buffer will have the same contents as the front buffer (screen). After a page flip the back buffer has the contents of the previous frame. The application should not make any assumptions about the contents of the back buffer at the beginning of the rendering loop.

NEW OPTIONAL FEATURE BITS

So many new features were added to RAVE 1.6 that we ran out of bits in the `kQAGestalt_OptionalFeatures` enum list, therefore, a new Gestalt has been created called `kQAGestalt_OptionalFeatures2`. This gestalt is used exactly like `kQAGestalt_OptionalFeatures` except that you must be careful to only ask for options of the form "`kQAOptional2_xxx`". If you pass one of the `kQAOptional_xxx` flags instead of `kQAOptional2_xxx`, you will get a meaningless result from the Gestalt.

The Optional2 values are described where needed in this document, but the following list summarizes these new values:

```
enum {
      kQAOptional2_None                              = 0,
      kQAOptional2_TextureDrawContexts               = (1 << 1),
      kQAOptional2_BitmapDrawContexts                = (1 << 2),
      kQAOptional2_Busy                              = (1 << 3),
      kQAOptional2_SwapBuffers                       = (1 << 4),
      kQAOptional2_Chromakey                         = (1 << 5),
      kQAOptional2_NonRelocatable                    = (1 << 6),
      kQAOptional2_NoCopy                            = (1 << 7),
      kQAOptional2_PriorityBits                      = (1 << 8),
      kQAOptional2_ FlipOrigin                       = (1 << 9),
      kQAOptional2_ BitmapScale                      = (1 << 10),
      kQAOptional2_ DrawContextScale                 = (1 << 11),
      kQAOptional2_ DrawContextNonRelocatable        = (1 << 12)
};
```

Note that these are not the only new optional flags in RAVE 1.6. Before we ran out of bits in the old `kQAOptional_xxx` list, we added several new option bits into the old list. The new version of the `kQAOptional_xxx` list is as follows, and the meanings of the new flags are described throughout this document:

```
enum {
	kQAOptional_None                        = 0,
	kQAOptional_DeepZ                       = (1 << 0),
	kQAOptional_Texture                     = (1 << 1),
	kQAOptional_TextureHQ                   = (1 << 2),
	kQAOptional_TextureColor                = (1 << 3),
	kQAOptional_Blend                       = (1 << 4),
	kQAOptional_BlendAlpha                  = (1 << 5),
	kQAOptional_Antialias                   = (1 << 6),
	kQAOptional_ZSorted                     = (1 << 7),
	kQAOptional_PerspectiveZ                = (1 << 8),
	kQAOptional_OpenGL                      = (1 << 9),
	kQAOptional_NoClear                     = (1 << 10),
	kQAOptional_CSG                         = (1 << 11),
	kQAOptional_BoundToDevice               = (1 << 12),
	kQAOptional_CL4                         = (1 << 13),
	kQAOptional_CL8                         = (1 << 14),
	kQAOptional_BufferComposite             = (1 << 15),
	kQAOptional_NoDither                    = (1 << 16),
	kQAOptional_FogAlpha                    = (1 << 17),
	kQAOptional_FogDepth                    = (1 << 18),
	kQAOptional_MultiTextures               = (1 << 19),
	kQAOptional_MipmapBias                  = (1 << 20),
	kQAOptional_ChannelMask                 = (1 << 21),
	kQAOptional_ZBufferMask                 = (1 << 22),
	kQAOptional_AlphaTest                   = (1 << 23),
	kQAOptional_AccessTexture               = (1 << 24),
	kQAOptional_AccessBitmap                = (1 << 25),
	kQAOptional_AccessDrawBuffer            = (1 << 26),
	kQAOptional_AccessZBuffer               = (1 << 27),
	kQAOptional_ClearDrawBuffer             = (1 << 28),
	kQAOptional_ClearZBuffer                = (1 << 29),
	kQAOptional_OffscreenDrawContexts       = (1 << 30)
};
```

NEW GESTALTS

In addition to the new Optional Gestalt bits, some new Gestalts have also been added. You saw above that the new Gestalt `kQAGestalt_OptionalFeatures2` was added to support the new Optional2 bits, but here is a listing of all of the Gestalts in RAVE 1.6. The meanings of the new Gestalts are described throughout this document.

```
enum TQAGestaltSelector {
	kQAGestalt_OptionalFeatures				= 0,
	kQAGestalt_FastFeatures					= 1,
	kQAGestalt_VendorID					= 2,
	kQAGestalt_EngineID					= 3,
	kQAGestalt_Revision					= 4,
	kQAGestalt_ASCIINameLength				= 5,
	kQAGestalt_ASCIIName					= 6,
	kQAGestalt_TextureMemory				= 7,
	kQAGestalt_FastTextureMemory				= 8,
	kQAGestalt_DrawContextPixelTypesAllowed			= 9,
	kQAGestalt_DrawContextPixelTypesPreferred		= 10,
	kQAGestalt_TexturePixelTypesAllowed			= 11,
	kQAGestalt_TexturePixelTypesPreferred			= 12,
	kQAGestalt_BitmapPixelTypesAllowed			= 13,
	kQAGestalt_BitmapPixelTypesPreferred			= 14,
	kQAGestalt_OptionalFeatures2				= 15,
	kQAGestalt_MultiTextureMax				= 16,
	kQAGestalt_NumSelectors					= 17,
	kQAGestalt_EngineSpecific_Minimum			= 1000
};
typedef enum TQAGestaltSelector TQAGestaltSelector;
```

Note that an Engine Specific Minimum Gestalt value has been added. This is so RAVE Engines can safely implement their own custom Gestalt values for their own purposes in the same way that the Tags allow this for custom Tag values.

One other important thing to be aware of when using these Gestalts is that not all RAVE engines may recognize all of these Gestalts. Engines which have not been updated to RAVE 1.6 will obviously be unaware of the new Gestalts which have been added. Your code should always check if QAEngineGestalt returns kQANotSupported which indicates that the RAVE engine has no idea what that Gestalt value is asking for.

NEW TAGS

A whole bunch of new Tags have been added to RAVE 1.6 to support the new features. The following lists all of the Tags that exist in RAVE 1.6 and the new tags are described throughout this document. Those tags shown in blue are new in 1.6.

Integer Tags

```
enum TQATagInt
{
	kQATag_ZFunction		= 0,
	kQATag_Antialias		= 8,
	kQATag_Blend			= 9,
	kQATag_PerspectiveZ		= 10,
	kQATag_TextureFilter		= 11,
	kQATag_TextureOp		= 12,
	kQATag_CSGTag			= 14,
	kQATag_CSGEquation		= 15,
	kQATag_BufferComposite		= 16,
	kQATag_FogMode			= 17,
	kQATag_ChannelMask		= 27,
	kQATag_ZBufferMask		= 28,
	kQATag_ZSortedHint		= 29,
	kQATag_ChromakeyEnable		= 30,
	kQATag_AlphaTestFunc		= 31,
	kQATag_DontSwap			= 32,
	kQATag_MultiTextureEnable	= 33,
	kQATag_MultiTextureCurrent	= 34,
	kQATag_MultiTextureOp		= 35,
	kQATag_MultiTextureFilter	= 36,
	kQATag_MultiTextureWrapU	= 37,
	kQATag_MultiTextureWrapV	= 38,
	kQATag_MultiTextureMagFilter	= 39,
	kQATag_MultiTextureMinFilter	= 40,
	kQATag_BitmapFilter		= 54,
	kQATag_DrawContextFilter	= 55,
	kQATagGL_DrawBuffer		= 100,
	kQATagGL_TextureWrapU		= 101,
	kQATagGL_TextureWrapV		= 102,
	kQATagGL_TextureMagFilter	= 103,
	kQATagGL_TextureMinFilter	= 104,
	kQATagGL_ScissorXMin		= 105,
```

```
	kQATagGL_ScissorYMin			= 106,
	kQATagGL_ScissorXMax			= 107,
	kQATagGL_ScissorYMax			= 108,
	kQATagGL_BlendSrc			= 109,
	kQATagGL_BlendDst			= 110,
	kQATagGL_LinePattern			= 111,
	kQATagGL_AreaPattern0			= 117,
	/* ...kQATagGL_AreaPattern1-30 */
	kQATagGL_AreaPattern31			= 148,
	kQATagGL_LinePatternFactor		= 149,
	kQATag_EngineSpecific_Minimum		= 1000
};
typedef enum TQATagInt TQATagInt;
```

Pointer Tags

```
enum TQATagPtr
{
	kQATag_Texture				= 13,
	kQATag_MultiTexture			= 26
};
typedef enum TQATagPtr TQATagPtr;
```

Float Tags

```
enum TQATagFloat
 {
	kQATag_ColorBG_a			= 1,
	kQATag_ColorBG_r			= 2,
	kQATag_ColorBG_g			= 3,
	kQATag_ColorBG_b			= 4,
	kQATag_Width				= 5,
	kQATag_ZMinOffset			= 6,
	kQATag_ZMinScale			= 7,
	kQATag_FogColor_a			= 18,
	kQATag_FogColor_r			= 19,
	kQATag_FogColor_g			= 20,
	kQATag_FogColor_b			= 21,
	kQATag_FogStart				= 22,
	kQATag_FogEnd				= 23,
	kQATag_FogDensity			= 24,
	kQATag_FogMaxDepth			= 25,
	kQATag_MipmapBias			= 41,
	kQATag_MultiTextureMipmapBias		= 42,
	kQATag_Chromakey_r			= 43,
	kQATag_Chromakey_g			= 44,
```

```
        kQATag_Chromakey_b              = 45,
        kQATag_AlphaTestRef             = 46,
        kQATag_MultiTextureBorder_a     = 47,
        kQATag_MultiTextureBorder_r     = 48,
        kQATag_MultiTextureBorder_g     = 49,
        kQATag_MultiTextureBorder_b     = 50,
        kQATag_MultiTextureFactor       = 51,
        kQATag_BitmapScale_x            = 52,
        kQATag_BitmapScale_y            = 53,
        kQATagGL_DepthBG                = 112,
        kQATagGL_TextureBorder_a        = 113,
        kQATagGL_TextureBorder_r        = 114,
        kQATagGL_TextureBorder_g        = 115,
        kQATagGL_TextureBorder_b        = 116
};
typedef enum TQATagFloat TQATagFloat;
```

About "GL" Tags

In an earlier version of RAVE we added some Tags and called them `kQATagGL_xxx`. You will note that in RAVE 1.6, new Tags like the ones for Texture Compositing do not have "GL" in their tag name even though their older counterparts still do. For example, the old texture Tag `kQATagGL_TextureBorder_a` has a Multi-Texture equivalent called `kQATag_MultiTextureBorder_a`. Note that we took the GL out of the name.

It was determined that there were not only too many "GL" Tags, but most of these GL Tags were not really OpenGL compliant or even close to their OpenGL counterpart. Therefore, the decision was made to stop using the `kQAGLTag_xxx` naming and to simply name all further Tags as `kQATag_xxxx`.

NEW PIXEL TYPES

Several new pixel types have been added to RAVE in order to ease porting of existing games to RAVE. In addition, some pixel types allow lighting effects when used with texture compositing.

The Pixel Types

`kQAPixel_RGB8_332`	8 bits/pixel, R=7:5, G=4:2, B=1:0
`kQAPixel_ARGB16_4444`	16 bits/pixel, A=15:12, R=11:8, G=7:4, B=3:0
`kQAPixel_ACL16_88`	16 bits/pixel, A=15:8, CL=7:0, 8 bit alpha + 8 bit color lookup
`kQAPixel_I8`	8 bits/pixel, I=7:0, intensity map (grayscale)
`kQAPixel_AI16_88`	16 bits/pixel, A=15:8, I=7:0, intensity map (grayscale)
`kQAPixel_YUVS`	16 bits/pixel, QD's kYUVSPixelFormat (4:2:2, YUYV ordering, signed UV)
`kQAPixel_YUVU`	16 bits/pixel, QD's kYUVUPixelFormat (4:2:2, YUYV ordering, unsigned UV)
`kQAPixel_YVYU422`	16 bits/pixel, QD's kYVYU422PixelFormat (4:2:2, YVYU ordering, unsigned UV)
`kQAPixel_UYVY422`	16 bits/pixel, QD's kUYVY422PixelFormat (4:2:2, UYVY ordering, unsigned UV)

The term "YUV 4:2:2" is a term that is used by the video industry and does not refer to a bit depth for a pixel format. This has the potential to cause great confusion.

New Gestalts

New gestalt selectors have been added to allow an application to know what pixel types the RAVE engine supports. The value returned by these gestalt selectors is a bit field where each bit corresponds to the respective pixel type. For instance, if a RAVE engine only supported the pixel types `kQAPixel_RGB16` and `kQAPixel_ARGB16` then the value it would return would be `(1L << kQAPixel_RGB16) | (1L << kQAPixel_ARGB16)`.

`kQAGestalt_DrawContextPixelTypesAllowed` Calling `QAEngineGestalt` with this selector will return all the draw context pixel types that are supported by the RAVE engine.

`kQAGestalt_DrawContextPixelTypesPrefered` Calling `QAEngineGestalt` with this selector will return all the draw context pixel types that are preferred by the RAVE engine. This list will not contain any pixel types that cause a serious rendering slow down.

`kQAGestalt_TexturePixelTypesAllowed` Calling `QAEngineGestalt` with this selector will return all the texture pixel types that are supported by the RAVE engine.

`kQAGestalt_TexturePixelTypesPrefered` Calling `QAEngineGestalt` with this selector will return all the texture pixel types that are preferred by the RAVE engine. This list will not contain any pixel types that cause

a serious rendering slow down or any pixel types that are internally expanded by the RAVE engine (using more memory) or any pixel types that must be truncated by the RAVE engine (causing a loss of quality).

`kQAGestalt_BitmapPixelTypesAllowed` Calling `QAEngineGestalt` with this selector will return all the bitmap pixel types that are supported by the RAVE engine.

`kQAGestalt_BitmapPixelTypesPrefered` Calling `QAEngineGestalt` with this selector will return all the bitmap pixel types that are preferred by the RAVE engine. This list will not contain any pixel types that cause a serious rendering slow down or any pixel types that are internally expanded by the RAVE engine (using more memory) or any pixel types that must be truncated by the RAVE engine (causing a loss of quality).

If `QATextureNew` or `QABitmapNew` are called with a pixel type that is unsupported by the RAVE engine the error `kQANotSupported` will be returned.

TEXTURE/BITMAP PRIORITY

The `flags` value passed to `QATextureNew` or `QABitmapNew` now uses the upper 4 bits as a 4-bit value signifying texture priority. Setting this priority value helps hardware drivers determine where to upload textures when AGP is present on the user's computer. AGP memory is a section of your regular system RAM that the 3D-accelerator card can access much more quickly than going through the PCI bus. This allows 3D accelerators cards to essentially have more VRAM than is present on the card itself.

Since AGP RAM is not as fast as true VRAM (but much faster than non-AGP system RAM), it is still preferable to have textures resident on the card's own VRAM. However, when true VRAM runs low, it may be necessary for the driver to load some textures into AGP RAM. By prioritizing your textures, you can help the driver determine which textures should go into VRAM and which into AGP RAM. Textures which are used often should be given high priority so that they will attempt to be loaded into VRAM, and textures which are not used often should be given lower priorities so that they will be the first to get moved to AGP RAM if VRAM runs low.

There is a macro in the `Rave.h` header file to assist you in easily setting the priority bits with the flags parameter:

```
flags |= QACalculatePriorityBits ( priority);
```

You use this macro in the following way:

```
unsigned long flags = kQATexture_Mipmap;             // Initialize  flags

flags |= QACalculatePriorityBits ( 4);               // Set medium-high priority
QATextureNew(myEngine, flags, pixelType, image, &raveTexture);
```

or like this:

```
unsigned long flags = kQATexture_NoCompression;      // Initialize  flags
                                                     // Set high priority
QATextureNew(myEngine,
             flags | QACalculatePriorityBits ( 8), pixelType, image, &raveTexture);
```

The valid range for the priority values is 0 to 15. 0 is top priority (or "ranking") and 15 is the lowest priority.

Note that since 0 is top priority, it is not actually necessary to set the priority value since the default will always be "top priority" of 0 as long as all 4 bits of the flags field are 0.

MULTIPLE MONITORS, ONE RAVE ENGINE

This section pertains to RAVE engine developers only. In RAVE prior to version 1.6 there had been a problem when a user had a machine with two or more monitors using the same 3D hardware on each monitor. In this case there was just one RAVE engine and multiple 3D accelerators. This caused a problem when, for instance, a RAVE application called `QATextureNew`. The RAVE engine did not know which 3D accelerator to create the texture on. This problem also applied to bitmaps, color tables, and gestalt calls.

The solution to this problem is for the RAVE engine to register with the RAVE manager once for each accelerator. Two new RAVE manager functions have been added for this purpose:

```
TQAError QARegisterEngineWithRefCon(TQAEngineGetMethod      engineGetMethod,
                                    long   refCon);

long QAGetCurrentEngineRefCon(void)
```

During your library initialization routine if you detect more than one accelerator card on the system call `QARegisterEngineWithRefCon` once for each accelerator. Pass a different `refCon` value for each accelerator. This will create one RAVE engine (`TQAEngine`) for each accelerator. When any of your engine methods are called you may determine which accelerator the call is intended for by calling `QAGetCurrentEngineRefCon`.

Note: QARegisterEngine(engineGetMethod) is equivalent to QARegisterEngineWithRefCon(engineGetMethod, 0).

FOG

Fog is used in many 3D applications as a cool visual effect and as a method for hiding the ugly clipping that occurs at the far edge of the rendered scene. It can be used to simulate not only fog but also darkness or other environments such as underwater.

Using fog is very easy to do, but a few things to keep in mind are:

- Fog in RAVE is controlled via state variables.
- Fog is applied after any texture lookup, after alpha test, and before alpha blend.
- Fog is applied using the following equation:

```
output color = input color * fog value + fog color * (1.0 - fog value)
```

A fog value of 1.0 indicates no fog. A fog value of 0 indicates full fog.

The fog value is calculated differently depending upon the fog mode. There are five modes for fog at this time.

```
none:                   fog value = 1
alpha:                  fog value = vertex alpha
linear:                 fog value = (fog end - depth) / (fog end - fog start)
exponential:            fog value = exp(-fog density * depth)
exponential squared:    fog value = exp(-fog density * depth * fog density *
                        depth)
```

Please note that in these equations depth is defined as `1.0 / invW`.
After the fog value is computed it is then clamped to the range 0.0 to 1.0 inclusive.

Fog State Variables & Flags

These are the new fog state variables.

`kQATag_FogMode`	(`int`) One of `kQAFogMode_xxxx`
`kQATag_FogColor_a`	(`float`) Fog color alpha
`kQATag_FogColor_r`	(`float`) Fog color red
`kQATag_FogColor_g`	(`float`) Fog color green
`kQATag_FogColor_b`	(`float`) Fog color blue
`kQATag_FogStart`	(`float`) Fog start *
`kQATag_FogEnd`	(`float`) Fog end *
`kQATag_FogDensity`	(`float`) Fog density
`kQATag_FogMaxDepth`	(`float`) Maximum value for 1.0 / invW

These are the values for the state variable `kQATag_FogMode`.

`kQAFogMode_None`	`0`	no fog
`kQAFogMode_Alpha`	`1`	fog value is alpha
`kQAFogMode_Linear`	`2`	fog = (end - z) / (end - start)
`kQAFogMode_Exponential`	`3`	fog = exp(-density * z)
`kQAFogMode_ExponentialSquared`	`4`	fog = exp(-density * z * density * z)

* Fog start and fog end are only used with fog mode linear. Fog density is only used for fog mode exponential and exponential squared.

The reason for the state variable `kQATag_FogMaxDepth` is that some 3D hardware cards use an internal fog table. Without knowing the range of depth values to expect, it would be hard to know how to build the fog table. For instance, QuickDraw 3D scales depth values to the range 0.0 to 1.0 before sending them to RAVE. This behavior is not required. Some RAVE applications use depth values scaled to their world space, i.e. the range 0.0 to yon (the far clipping plane). The yon value can be any positive number. Therefore the `kQATag_FogMaxDepth` value is purely informational. It is provided to the RAVE engine to aid in the creation of an internal fog table.

Here are the engine gestalt values for fog.

`kQAOptional_FogAlpha`	This bit is set if the engine supports fog mode alpha.
`kQAOptional_FogDepth`	This bit is set if the engine supports fog mode linear, exponential and exponential squared.
`kQAFast_FogAlpha`	This bit is set if the engine accelerates fog mode alpha.
`kQAFast_FogDepth`	This bit is set if the engine accelerates fog mode linear, exponential and exponential squared.

Special Considerations

When the fog mode is set to `kQAFogMode_Alpha` the alpha values of the vertices are used for fog and thus are not used for alpha. To be more specific, when using `kQAFogMode_Alpha`, alpha blending based upon vertex alpha is disabled. Alpha blending based upon texture alpha is not disabled by using `kQAFogMode_Alpha`.

When the fog mode is not set to `kQAFogMode_Alpha` the alpha values of the vertices are not used for fog and thus are used for alpha. To be more specific, when using any of the depth based fog modes, alpha blending based upon vertex alpha is enabled as it normally would be.

If you wish to use fog and alpha blending you cannot use `kQAFogMode_Alpha`.

Fog Example

The following function shows how to set linear yellow fog that starts halfway to the yon plane and achieves full density at the yon plane. It assumes that w ranges from 0.0 to 1.0.

```
/********************** SETRAVE FOG ******************************/

void SetMyFog(TQADrawContext *drawContext)
{
	QASetInt(drawContext, kQATag_FogMode, kQAFogMode_Linear);

	QASetFloat(drawContext, kQATag_FogColor_r, 1.0);
	QASetFloat(drawContext, kQATag_FogColor_g, 1.0);
	QASetFloat(drawContext, kQATag_FogColor_b, 0);
```

```
        QASetFloat(drawContext, kQATag_FogStart, .5 );
        QASetFloat(drawContext, kQATag_FogEnd, 1.0);
        QASetFloat(drawContext, kQATag_FogMaxDepth, 1.0);

        QASetFloat(drawContext, kQATag_FogDensity, 1.0);
}
```

SINGLE PASS TEXTURE COMPOSITING

Single pass texture compositing allows two or more different textures to be applied to the same object. Texture compositing can be used to blend a regular texture mapped object with an environment map, reflection map, projected map, or detail maps. Additionally it can be used to achieve very flexible lighting effects. New texture map formats, including `kQAPixel_I8` have been added for use as an intensity map.

For the purpose of clarity, a few terms need to be defined:

Multi-Texturing : In RAVE and OpenGL, texture compositing is generally referred to as Multi-Texturing.

Base Texture: This is the texture, which is applied to an object in the usual way. This is the single texture that has always existed in RAVE.

Multi-Texture : A multi-texture refers to one of the additional composited textures you wish to apply to an object. When you apply one multi-texture to an object, this means that you are applying the Base Texture plus one additional multi-texture.

Layer: Since you can have multiple multi-textures applied to an object, the layer simply refers to a particular multi-texture.

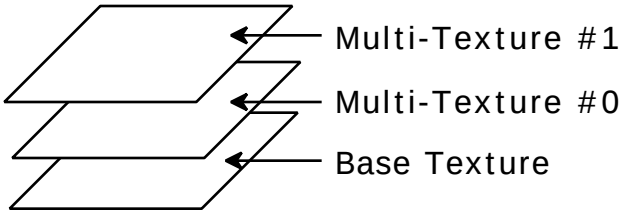

Diagram showing 3 composited textures: 1 base and 2 multi-textures

RAVE allows an unlimited number of multi-textures to be assigned to an object. The maximum number of textures that can be applied to an object is limited only by the technical restraints of the hardware being used. For example, the ATI Rage II cards do not support single pass texture compositing at all, therefore, no multi-textures can be used. The ATI Rage Pro and Rage 128 cards, however, support one multi-texture. Future hardware may support more multi-texture layers, therefore, RAVE 1.6 has been designed to account for this possibility.

Determining The Engine Capabilities

The first thing you will want to do before using multi-textures is to determine how many multi-textures your drawing engine supports. This information is acquired with a new gestalt:

```
kQAGestalt_MultiTextureMax
```

This gestalt will return the number of multi-textures that the current engine supports. A value of 0 indicates that this engine cannot do multi-

texturing (the Rage II will return this), thus, at best only a single base texture can be applied to an object. A value of 1 indicates that one multi-texture is supported, so that you can blend the main texture with one multi-texture. A value of 5 would indicate that you could combine your main texture with 5 additional multi-texture layers.

Working With Multi-Texture Layers

SETTING THE NUMBER OF LAYERS TO USE

Since RAVE supports an unlimited number of texture layers, you need to tell the drawing engine how many layers you are currently using for the current object. A new Tag has been added so that you can tell the drawing engine how many texture layers to apply to this object:

```
kQATag_MultiTextureEnable
```

If you set this value to 0, then this effectively disables texture compositing and only the base texture will be used when rendering the object. A value of 1 or more indicates how many multi-texture layers to composite with the base texture. So, if you want to composite a total of 3 textures, you would do:

```
QASetInt(context, kQATag_MultiTextureEnable, 2);
```

This tells the RAVE drawing engine to use the base texture plus 2 multi-textures.

THE CURRENT LAYER

Each multi-texture layer needs its own texture pointer, u/v's, blending mode, etc. There are new Tags and functions in RAVE 1.6 for working with multi-texture layers, and the most important new Tag is the one which lets you specify which layer you are currently working with:

```
kQATag_MultiTextureCurrent
```

If your drawing engine only supports one multi-texture layer, then you should set this value to 0 since you will always be working with layer #0. If, however, your engine supports more than one layer, you will have to change this value from 0...(n-1) to set which layer you are currently working with.

Remember that to work with the Base texture, you use the old functions and Tags as has always been done with RAVE. These new functions and Tags are only for working with the additional multi-texture layers.

TEXTURE LAYER PARAMETERS

Once you have set the current texture layer with `kQATag_MultiTextureCurrent`, you need to set the texture's parameters. Several new Tags exist which parallel Tags for the base texture that has always existed in RAVE:

```
kQATag_MultiTextureOp
```

```
kQATag_MultiTextureFilter
kQATag_MultiTextureWrapU
kQATag_MultiTextureWrapV
kQATag_MultiTextureMagFilter
kQATag_MultiTextureMinFilter
```

There is also a new kind of `TQATagPtr` for multi-textures which is used to set the pointer to the current texture to use for the current texture layer:

```
kQATag_MultiTexture
```

Use this Tag Pointer the same way as you used `kQATag_Texture` except that this applies to the current multi-texture layer instead of the base texture.

When using texture compositing, additional parameters must be supplied to the RAVE engine for each of the textures. These include `uOverW`, `vOverW`, and `invW` parameters:

```
typedef struct TQAVMultiTexture
{
	float  invW;
	float  uOverW;
	float  vOverW;
} TQAVMultiTexture;
```

This information is passed to RAVE by calling a new function:

```
void   QASubmitMultiTextureParams (
	const TQADrawContext      *drawContext,       /* Draw context */
	unsigned long             nParams,            /* Number of params */
	const TQAVMultiTexture    *params);           /* params */
```

You call `QASubmitMultiTextureParams` before calling `QADrawTriTexture`, `QADrawTriMeshTexture`, or `QADrawVTexture`. The `params` array is an array of

TQAVMultiTexture, one element for each vertex you are about to Draw. If you are passing a triangle to RAVE via QADrawTriTexture, then nParams will always be 3. If, however, you are passing a TriMesh or other array of vertices to RAVE, then the number of parameters will depend on the quantity of vertices being passed to RAVE for drawing.

If you are applying more than one multi-texture layer to your object, then you will have to call QASubmitMultiTextureParams multiple times, once for each texture layer. For example, to draw a triangle with a total of 3 textures (one base, and 2 multi-textures), you would do something like this:

```
QASetInt(context, kQATag_MultiTextureEnable, 2);     /* use 2 extra layers */

QASetInt(context, kQATag_MultiTextureCurrent, 0);    /* edit layer 0 */
QASetPtr(context, kQATag_MultiTexture, texture0);    /* point to texture 0 */
QASetInt(context, kQATag_MultiTextureOp,             /* blend mode */
                    kQAMultiTexture_Add);
QASubmitMultiTextureParams(context, 3, &params);     /* submit uv's for layer 0 */

QASetInt(context, kQATag_MultiTextureCurrent, 1);    /* edit layer 1 */
QASetPtr(context, kQATag_MultiTexture, texture1);    /* point to texture 1 */
QASetInt(context, kQATag_MultiTextureOp,             /* blend mode */
                    kQAMultiTexture_Modulate);
QASubmitMultiTextureParams(context, 3, &params);     /* submit uv's for layer 1 */

QADrawTriTexture(context, v1, v2, v3, flags);        /* draw the triangle */
```

New State Variables & Flags

The method in which the multiple textures are combined to compute the final pixel color is defined by:

`kQATag_MultiTextureOp`

This state variable may take on the following values:

`kQAMultiTexture_Add`	This specifies that the 2 texels are added to form the final pixel color. This can be used to simulate effects such as spotlights.
`kQAMultiTexture_Modulate`	This specifies that the 2 texels are multiplied to form the final pixel color. This can be used to simulate effects such as shadows and bumpmaps.
`kQAMultiTexture_BlendAlpha`	This specifies that the 2 texels are blended according to the second texels alpha. This can be used to simulate effects such as decals.
`kQAMultiTexture_Fixed`	This specifies that the 2 texels are blended by a fixed factor. The floating point state variable `kQATag_MultiTextureFactor` is used to determine the blending factor.

New Tags

```
kQATag_MultiTextureFilter
kQATag_MultiTextureOp
kQATag_MultiTextureWrapU
kQATag_MultiTextureWrapV
kQATag_MultiTextureMagFilter
kQATag_MultiTextureMinFilter
kQATag_MultiTextureBorder_a
kQATag_MultiTextureBorder_r
kQATag_MultiTextureBorder_g
kQATag_MultiTextureBorder_b
kQATag_SecondaryMipmapBias
kQATag_MultiTextureFactor
```

New Gestalts

`kQAOptional_CompositeTextures`	This bit is set if the engine supports texture compositing.
`kQAFast_MultiTextures`	This bit is set if the engine accelerates texture compositing.

TEXTURE MIPMAP SELECTION BIAS

An application may wish to effect when a transition to a new mipmap page occurs. This can allow certain textures to look crisper at the cost of aliasing. For example, textures that contain text need to be crisp so that the text can be easily read. The state variable `kQATag_MipmapBias` is a hint to the engine as to when a transition should occur. The range is from 0.0 to 1.0. The greater the bias the more the transition from a larger map to a smaller map is delayed. This in effect means that values closer to 1 are crisper and values closer to 0 have less aliasing. The default value should be 0.5.

New Tags

`kQATag_MipmapBias`	The mipmap page bias factor.
`kQATag_MultiTextureMipmapBias`	Same, but for multi-texture layer.

New Gestalts

`kQAOptional_MipmapBias`	This bit is set if the engine supports mipmap selection bias.

TEXTURE ANIMATION

Some RAVE applications may wish to animate textures. To be more specific, they need to change the contents of a texture frequently. Prior to RAVE 1.6 the only way to do this was to call `QATextureDelete` and `QATextureNew`. Due to the overhead of these calls this is not the most efficient way to perform texture animation, therefore, two new functions have been added to provide for texture animation in a more efficient manner:

```
TQAError QAAccessTexture(const   TQAEngine     *engine,
                         TQATexture            *texture,
                         long                  mipmapLevel,
                         long                  flags,
                         TQAPixelBuffer        *buffer);
```

The application calls this function when it wishes to have access to the texture's buffer. If possible the RAVE engine should return (in `buffer`) the location and format of the texture in texture memory.

Depending on the hardware, this may not always be possible. The texture may be stored in a proprietary format. The texture may be in a memory area that is not addressable by the application (perhaps some sort of texture cache). If this is the case, the RAVE engine should create a buffer in main memory, copy the texture into it, and return this location.

If the flag `kQANoCopyNeeded` is specified this indicates that the application plans to update the entire texture (or a rectangular portion of it) and a copy of the original texture is not needed.

```
TQAError QAAccessTextureEnd(     const TQAEngine     *engine
                                 TQATexture          *texture,
                                 const TQARect       *dirtyRect);
```

After the application has finished accessing the texture is must call `QAAccessTextureEnd`. It should pass in `dirtyRect` the area of the texture that it has modified. Passing `NULL` indicates that the whole texture has been modified.

WARNING: No attempt should be made to access the buffer after `QAAccessTextureEnd` is called. The buffer may not be in the same location or may no longer be valid. If further access to the buffer is needed `QAAccessTexture` must be called again.

Special Considerations

If the texture was created with the flag `kQATexture_NoCompression` the buffer returned by `QAAccessTexture` should have the same pixel type as the original texture.

If the texture was created without the flag `kQATexture_NoCompression` or with the flag `kQATexture_HighCompression` then buffer returned by `QAAccessTexture` may have a different pixel type than the original texture.

In most cases if the application wishes to use this feature it should create the texture with the `kQATexture_NoCompression` flag.

The duration between `QAAccessTexture` and `QAAccessTextureEnd` should be kept as short as possible. This is due to the possibility that some other process (such as rendering or memory compaction) may be blocked while waiting for the buffer access to finish.

Due to the possibility that the RAVE engine may have to allocate a working buffer to provide texture access, only one texture should be accessed at a time.

BITMAP ANIMATION

Bitmap updating is accomplished in the same fashion as texture animation. All of the same warnings and considerations apply.

The functions used to access bitmap buffers are `QAAccessBitmap` and `QAAccessBitmapEnd`.

```
TQAError QAAccessBitmap(const  TQAEngine       *engine
                           TQABitmap           *bitmap,
                           long                flags,
                           TQAPixelBuffer      *buffer);

TQAError QAAccessBitmapEnd(const TQAEngine     *engine
                           TQABitmap           *bitmap,
                           const TQARect       *dirtyRect);
```

OFFSCREEN DRAW CONTEXTS

In order to create an offscreen draw context, a new device type has been added:

```
typedef struct TQADeviceOffscreen
{
      TQAImagePixelType          pixelType;
} TQADeviceOffscreen;
```

This is intended to be used as an offscreen drawing context. The RAVE engine will allocate the buffer for this draw context.

Creating An Offscreen Draw Context

To create an offscreen draw context the application creates a `TQADevice` with a device type of `kQADevice_Offscreen`. The application specifies the desired pixel type in the `TQADeviceOffscreen` structure. The application then calls `QADrawContextNew` in the normal fashion. Application access to this buffer is provided by `QAAccessDrawBuffer` or through the use of notice methods.

A cache draw context is intended to be used to initialize a draw context. When a cache draw context is created it has the same pixel type as the screen that it is associated with.

When an offscreen draw context is created the pixel type is specified by the application.

New Gestalt Value

`kQAOptional_OffscreenDrawContexts`	An offscreen draw context is similar to a cache draw context. There are some important differences.

TEXTURE / BITMAP DRAW CONTEXT

There are times when an application may wish to render into a texture or a bitmap. This technique may be used to simulate a video screen, a rear view mirror, dynamic 3D sprites or other effects.

Creating A Texture Draw Context

To create a texture draw context, first create an offscreen draw context then create a texture using `QATextureNewFromDrawContext`. This will create a texture that shares its pixel buffer with the offscreen draw context. Under circumstances where the actual pixel buffer cannot be shared, the contents will be copied between buffers whenever changes are made to achieve the same net effect.

If supported by the RAVE engine, an offscreen draw context may be used as a texture.

```
TQAError QATextureNewFromDrawContext(const TQADrawContext         *drawContext,
                                     unsigned long       flags,
                                     TQATexture          **newTexture);
```

The offscreen draw context must have a valid height, width and pixel type to be used as a texture.

If supported by the RAVE engine, an offscreen draw context may be used as a bitmap:

```
TQAError QABitmapNewFromDrawContext(TQADrawContext          *drawContext,
                                    unsigned long           flags,
                                    TQABitmap               **newBitmap);
```

The offscreen draw context must have a pixel type to be used as a bitmap.

Note To Rave Engine Developers

Note to RAVE engine developers: Whenever possible the texture and the offscreen draw context should share the same buffer. If this is not possible (due to hardware restrictions) any changes made to one buffer should be copied to the other buffer. At this time there is no direct way to create a mipmapped texture from an offscreen draw context.

Note to RAVE engine developers: Whenever possible the bitmap and the offscreen draw context should share the same buffer. If this is not possible (due to hardware restrictions) any changes made to one buffer should be copied to the other buffer.

Special Considerations

After the texture has been rendered, the draw context may be deleted with `QADrawContextDelete`. The texture and its contents will remain. Also, if the texture is no longer needed but the draw context is still needed, then the texture may be deleted with `QATextureDelete`. To be more specific, the

texture and the draw context may be deleted in either order. Deleting one will leave the other intact. The behavior with bitmaps is the same.

Only one texture or bitmap may be attached to any one draw context in this fashion.

New Gestalt Values

```
KQAOptional2_TextureDrawContexts
KQAOptional2_BitmapDrawContexts
```

MISC DRAW CONTEXT ADDITIONS

New Functions

```
TQABoolean QABusy(const TQADrawContext        *drawContext);
```

This function returns `kQATrue` if the draw context is busy, `kQAFalse` if the draw context is idle. This function does not wait for drawing to finish.

```
TQAError QASwapBuffers(    const TQADrawContext       *drawContext,
                                const  TQARect              *dirtyRect);
```

This causes the front buffer to be updated. The `dirtyRect` parameter indicates how much of the draw context needs to be copied to the front buffer. If page flipping is enabled this function causes the front buffer and the back buffer to be flipped.

New State Variable Tag

`kQATag_DontSwap`

Setting this variable to true indicates to the RAVE engine that is should NOT swap buffers automatically during (or after) `QARenderEnd()` unless specifically instructed to do so by `QASwapBuffers()`.

New Gestalt Values

```
kQAOptional2_Busy
kQAOptional2_SwapBuffers
kQAOptional2_DrawContextScale
kQAOptional2_DrawContextNonRelocatable
```

New Flags For Context Creation

Two new flags have been added for the creation of draw contexts.

`kQAContext_Scale`

This flag is used to create a scaled draw context. It is described below.

`kQAContext_NonRelocatable`

This flag is used to create a draw context with non-movable buffers. That is to say that the back buffer and the z-buffer will not move in memory while the draw context is in existence. Be sure to check for the optional2 gestalt bit `kQAOptional2_DrawContextNonRelocatable` before attempting to use this feature.

Scaled Draw Contexts

It is now possible for a RAVE engine to support scaled draw contexts. A scaled draw context may be used for full scene anti-aliasing or for pixel-doubling or other related effects. Put simply, a scaled draw context is a draw context in which the front buffer is a different size than the back buffer. At the end of a rendering loop when the back buffer is copied to the front buffer scaling is applied to resize the image to fit the front buffer.

If the front buffer is smaller than the back buffer then the image is scaled down. Full scene anti-aliasing can be performed in this way. This can improve the quality of the scene. The drawbacks are that the back buffer and z-buffer are larger, thus taking more memory from the accelerator and possibly requiring more time to render the scene.

If the front buffer is larger than the back buffer then the image is scaled up. Pixel-doubling can be performed in this way. This can reduce the amount of memory used by the back buffer and the z-buffer. This can also improve render times. A drawback is that the quality of the scene is reduced.

A common drawback of either method is that the scaled blit (or copy) to the front buffer may be slower than the standard non-scaled blit on some accelerators. Although, this is not always the case.

For obvious reasons scaled draw contexts must also be double buffered.

Creating A Scaled Draw Context

Before you attempt to create a scaled draw context, first determine that the drawing engine can support this feature. This can be done by checking the engine's "optional2" gestalt value for `kQAOptional2_DrawContextScale`.

To create a scaled draw context you call `QADrawContextNew` with the `kQAContext_Scale` flag. You must also pass a pointer to an array of two `TQARects` in the second parameter. The first element of this array contains the size of the back buffer. The second element of this array contains the size and location of the front buffer.

The following code snippet shows an example of creating a scaled draw context.

```
TQAError error;
TQARect drawContextRects[2];
unsigned long optionalFeatures2;

error = QAEngineGestalt(
       theEngine,
       kQAGestalt_OptionalFeatures2,
       &optionalFeatures2);

if(error != kQANoErr) {
       // this engine does not support kQAGestalt_OptionalFeatures2
       // handle the error
       // be aware that kQAGestalt_OptionalFeatures2 is a new gestalt selector
       // it is likely to return an error code of kQANotSupported
}

if(!(optionalFeatures2 & kQAOptional2_DrawContextScale)){
       // this engine does not support scaled draw contexts
       // handle the error
}

// the size of the back buffer is in element 0 of the array
drawContextRects [0] = backBufferRect;
// the size and location of the front buffer is in element 1 of the array
drawContextRects [1] = frontBufferRect;


// create the draw context with the kQAContext_Scale flag
error  = QADrawContextNew(
       &theDevice,
       &drawContextRects,   // pass the address of the array of two TQARects
       &theClip,
       &theEngine,
       kQAContext_DoubleBuffer | kQAContext_Scale,
       &theDrawContext);
```

Be aware that all drawing to the scaled draw context is done using the size of the back buffer, not the size of the front buffer. For example, if the front buffer is 640 pixels wide and the back buffer is 320 pixels wide, then an object drawn at column 305 would appear at column 610 of the front buffer.

The filtering of this scaled blit can be controlled by the new integer state variable `kQATag_DrawContextFilter`. Valid values for this variable are `kQAFilter_Fast`, `kQAFilter_Mid` and `kQAFilter_Best`.

ACCESS TO THE DRAWING BUFFER

Some applications may wish to have direct access to the draw buffer of a draw context. This can be used for (but is not limited to) the following reasons:

- Clearing the draw context to a specific image at the beginning of the rendering loop.
- Performing antialiasing, motion blur, depth of focus or other special effects.
- Copying the rendered image.

New Functions

```
TQAError QAAccessDrawBuffer(     const TQADrawContext       *drawContext,
                                 TQAPixelBuffer             *buffer);
```

The application calls this function when it wishes to have access to the draw context's pixels. Upon return buffer will contain a description of the drawing buffer.

```
TQAError QAAccessDrawBufferEnd(const    TQADrawContext      *drawContext,
                               const  TQARect               *dirtyRect);
```

The application calls this function when it is finished accessing the pixels of the draw context. The application should set `dirtyRect` to indicate how much of the draw context it has changed. Passing `NULL` indicates the entire draw context may have changed.

The duration between `QAAccessDrawBuffer` and `QAAccessDrawBufferEnd` should be kept as short as possible. This is due to the possibility that some other process (such as rendering or memory compaction) may be blocked while waiting for the buffer access to finish.

Cautions

To insure that all drawing has finished the application should call `QASync` before accessing the draw buffer.

No attempt should be made to access the buffer after `QAAccessDrawBufferEnd` is called. The buffer may not be in the same location. If further access to the buffer is needed `QAAccessDrawBuffer` must be called again.

CHANNEL & Z-BUFFER MASKS

Channel masks and the z buffer mask control the writing of data to the draw buffer channels and the z-buffer. Possible uses include anaglyph (red/blue glasses) 3D rendering. These masks control both drawing and clearing.

New Tags & Gestalts

`kQATag_ChannelMask`	Valid values are any combination of the following:
`kQAChannelMask_r`	`1 << 0`
`kQAChannelMask_g`	`1 << 1`
`kQAChannelMask_b`	`1 << 2`
`kQAChannelMask_a`	`1 << 3`

A value of 1 in any bit position indicates that writing is enabled for that channel.

`kQATag_ZBufferMask`	
`kQAZBufferMask_Disable`	`0`
`kQAZBufferMask_Enable`	`1`

`kQAOptional_ChannelMask`
`kQAOptional_ZBufferMask`

BUFFER CLEARING

There are times when you may wish to have the draw buffer and/or z-buffer of your Draw Context cleared.

```
TQAError QAClearDrawBuffer(drawContext, rect, initialContext);
```

Clears the specified area of the draw buffer of the specified draw context to the value contained in the state variables `kQATag_ColorBG_a`, `kQATag_ColorBG_r`, `kQATag_ColorBG_g`, `kQATag_ColorBG_b` unless `initialContext` is not `NULL`. If `initialContext` is a valid Draw Context, then the `drawContext` is cleared to `initialContext`. This works the same as `QARenderStart()`. When initialContext is `NULL`, the clear color can be masked using `kQATag_ChannelMask`.

```
TQAError QAClearZBuffer(drawContext, rect, initialContext);
```

Clears the specified area of the z buffer of the specified draw context to the value contained in the state variable `kQATagGL_depthBG` when `initialContext` is `NULL`. Can be masked using `kQATag_ZBufferMask`.

New Gestalt Values

```
kQAOptional_ClearDrawBuffer
kQAOptional_ClearZBuffer
```

Z-SORTED GESTALT VALUES

Since some RAVE engines can sort the triangles submitted to RAVE and some cannot, an optional flag was added, `kQAOptional_ZSorted`, that indicates whether the engine can depth sort triangles or not. An additional flag, `kQAFast_ZSorted` indicates that z-sorting is specifically accelerated. If this flag is not set then it may be faster for the application to sort the triangles itself.

For engines that can depth sort triangles the application may set a hint indicating whether or not triangles are being submitted in a depth sorted order. This is done via the state variable, `kQATag_ZSortedHint`.

For compatibility with QuickDraw 3D application which use semi-transparent materials, all RAVE engines should support z-sorting since QuickDraw 3D applications have no way of sorting the triangles themselves and QuickDraw 3D itself has no mechanism for this either.

To clarify, a RAVE engine should return true for the `kQAOptional_ZSorted` gestalt if and only if the RAVE engine has the ability to cache and sort transparent triangles such that transparency will appear correct no matter what order the transparent triangles are submitted in. The user can disable or enable this sorting ability with the `kQATag_ZSortedHint` tag.

CHROMAKEY

Chromakey is used to apply transparency to a shape based upon a reference color instead of (or in conjunction with) an alpha channel.

```
kQATag_Chromakey_r
kQATag_Chromakey_g
kQATag_Chromakey_b
kQATag_ChromakeyEnable
```

New Gestalt Values

`KQAOptional2_Chromakey` This bit is set if the engine supports Chroma key.

ALPHA TEST

Allows you to set up an alpha compare function (e.g. never, always, equal, greater than, etc.) and reference value to test incoming alpha values. If the test fails, the value is rejected, otherwise, it is accepted. Can be used to create see-through decal transparency effects, among other things.

`kQATag_AlphaTestFunc` One of the following values:

```
kQAAlphaTest_None
kQAAlphaTest_LT
kQAAlphaTest_EQ
kQAAlphaTest_LE
kQAAlphaTest_GT
kQAAlphaTest_NE
kQAAlphaTest_GE
kQAAlphaTest_True
```

`kQATag_AlphaTestRef` A floating point value from 0.0 to 1.0

New Gestalt Values

`kQAOptional_AlphaTest` This bit is set if the engine supports alpha test.

SCALED BITMAP DRAWING

RAVE engines may now support scaled drawing of bitmaps.

To draw scaled bitmaps use the following new state variables.

`kQATag_BitmapScale_x`	horizontal bitmap scale factor, default value is 1.0
`kQATag_BitmapScale_y`	vertical bitmap scale factor, default value is 1.0

`kQATag_BitmapFilter` One of the following values:
`kQAFilter_Fast`
`kQAFilter_Mid`
`kQAFilter_Best`

New Gestalt Values

`kQAOptional2_BitmapScale`	This bit is set if the engine supports scaled bitmap drawing.
`kQAFast_BitmapScale`	This bit is set if the engine accelerates scaled bitmap drawing.

NAME CHANGES

A few name changes have taken place to clarify the API. The old names are still available to help backward compatibility when the macro `RAVE_OBSOLETE` is defined to be 1.

Two notification methods have been renamed to more accurately reflect their intended use:

`kQAMethod_BufferInitialize` has changed to `kQAMethod_ImageBufferInitialize`.

`kQAMethod_BufferComposite` has changed to `kQAMethod_ImageBuffer2DComposite`.