RenderWare
Graphics

White Paper

RenderWare Graphics and OpenGL

Contact Us

Criterion Software Ltd.

For general information about RenderWare Graphics e-mail info@csl.com.

Developer Relations

For information regarding Support please email devrels@csl.com.

Sales

For sales information contact: rw-sales@csl.com

Acknowledgements

With thanks to RenderWare Graphics development and documentation teams.

Table of Contents

1.1 Introduction

On a number of platforms, RenderWare Graphics utilizes OpenGL to render geometry. OpenGL is a very rich and flexible API and some of its features are not exposed through the RenderWare Graphics API. Furthermore, it is conceivable that there are some tasks that RenderWare Graphics performs that could be accomplished more efficiently with an OpenGL specific feature. For example, to enable wireframe rendering, it is far easier to call:

```
glPolygonMode(GL_FRONT, GL_LINE);
```

than to process each triangle of a geometry and submit lines to RenderWare Graphics using immediate mode functions.

As a result, provisions have been made to allow integration of OpenGL code and RenderWare Graphics code into a single application. This white paper aims to describe how this can be achieved in a safe and efficient manner.

RenderWare Graphics provides a wrapper over some of the OpenGL API and this is also described here to ensure that conflicts do not occur when native OpenGL code is used in a RenderWare Graphics application.

1.2 OpenGL Specific Coding Guidelines

Adding OpenGL calls to a RenderWare Graphics application is a simple task but there are some guidelines that are recommended to be adhered to. These are:

- Predicate OpenGL specific code so as not to adversely affect builds for other targets/platforms. This can be done by checking for the definition of a platform specific define that is declared in **`rwcore.h`** as shown below:

```
#ifdef OPENGL_DRVMODEL_H

/* OpenGL specific code */

#endif /* OPENGL_DRVMODEL_H */
```

- Further to this, the head of a C file that makes calls to OpenGL functions should include the following:

```
#ifdef OPENGL_DRVMODEL_H

#ifdef WIN32
#include <windows.h>
#endif /* WIN32 */

#include <gl/gl.h>

#endif /* OPENGL_DRVMODEL_H */
```

This code is portable across all platforms that support RenderWare Graphics for OpenGL.

- Only call OpenGL functions when the engine is in a started state (i.e. after **`RwEngineStart`** has been called and before **`RwEngineStop`** has been called). This is because the OpenGL graphics context is not created until **`RwEngineStart`** has been called.

- When linking an OpenGL application that uses RenderWare Graphics, the linker does not require **`opengl32.lib`** (on Windows) as this library is already incorporated into **`rwcore.lib`**. However, if functions from the GLU are called, then the application will need to be linked against **`glu32.lib`** (on Windows).

1.3 Setting OpenGL Render State

RenderWare Graphics calls various OpenGL functions to set render state. Many of these are called from within **RwRenderStateSet**. For example, calling:

```
RwRenderStateSet(rwRENDERSTATESHADEMODE,
                 (void *)rwSHADEMODEGOURAUD);
```

will result in a call internally to:

```
glShadeModel(GL_SMOOTH);
```

The application can also set OpenGL render state. However, RenderWare Graphics tracks most render states internally so if the application were to call:

```
glShadeModel(GL_FLAT);
```

then RenderWare Graphics would be unaware of this, and subsequent calls to:

```
RwRenderStateSet(rwRENDERSTATESHADEMODE,
                 (void *)rwSHADEMODEGOURAUD);
```

would not have any effect as RenderWare Graphics would consider Gouraud shading to already be enabled. As a result, the application must take great care to ensure that RenderWare Graphics' state cache does become out of synch with OpenGL's internal state.

1.3.1 State enable/disable

RenderWare Graphics also tracks some OpenGL states that are enabled/disabled through **glEnable** and **glDisable**. These are cached internally by RenderWare Graphics to avoid costly redundant state changes.

Instead of using **glEnable**, an application may use **RwOpenGLEnable** with an appropriate token from the **RwOpenGLStateToken** enumeration. Similarly there are **RwOpenGLDisable** and **RwOpenGLIsEnabled** functions.

For example, to ensure lighting is enabled, one would call:

```
RwOpenGLEnable( rwGL_LIGHTING );
```

1.3.2 Pushing and Popping OpenGL Render State

One useful mechanism that OpenGL provides to safeguard against corruption of render state is attribute bits. Let's consider the example given above where we wish to explicitly set a flat shading mode using OpenGL but without impinging upon RenderWare Graphics' private state cache. The following code would be considered safe:

```
/* Take a snapshot of lighting state here */
glPushAttrib(GL_LIGHTING_BIT);

    glShadeModel(GL_FLAT);

    /* Render flat shaded geometry here */

/* Pop the lighting state which includes the shade model */
glPopAttrib();
```

The run-time overhead of the **`glPushAttrib`** function for an arbitrary set of attribute bits is not under consideration here. Hence it is left to the discretion of the developer whether they make use of this feature of OpenGL or not.

See OpenGL reference materials for more information on these functions.

1.4 Setting Up OpenGL Transformations

OpenGL supports three matrix stacks by default – the projection matrix stack, the model-view matrix stack and the texture matrix stack. RenderWare Graphics only manipulates the first two internally; the texture matrix stack is manipulated by some plugins to achieve special effects.

There is a useful function that will apply a RenderWare Graphics matrix (of type **RwMatrix**) and multiply it with the top element of the current matrix stack (as specified by **glMatrixMode**). The prototype for this function is:

```
void _rwOpenGLApplyRwMatrix(RwMatrix *matrix);
```

Therefore, to set up a transformation using a RenderWare Graphics frame, do the following:

```
glPushMatrix();

    glLoadIdentity();
    _rwOpenGLApplyRwMatrix(RwFrameGetLTM(myFrame));

    /* Render transformed geometry here */

glPopMatrix();
```

✎ Remember that it is important to preserve RenderWare Graphics' internal representation of OpenGL state. This is why the current matrix stack is pushed and popped in the above example code. After it has been executed, the OpenGL state machine is in exactly the same state as it was before it was executed.

However, since both RenderWare Graphics and OpenGL are both based on right handed co-ordinate systems, it is actually fairly trivial to use OpenGL matrix transformation functions directly.

1.5 Submission of OpenGL Geometry

RenderWare Graphics submits geometry in three different ways when using OpenGL as its polygon engine. These are: OpenGL immediate mode, vertex arrays and the NVIDIA specific extension **`GL_NV_vertex_array_range`** (when present). RenderWare Graphics also provides a state caching wrapper for vertex arrays that is used internally for improved performance.

1.5.1 OpenGL Immediate Mode

RenderWare Graphics submits RwIm3D and RwIm2D geometry via calls to OpenGL's immediate mode functions. So a call to:

```
RwIm3DRenderTriangle(0, 1, 2);
```

will internally result in three calls to:

```
glColor4ubv((GLubyte*)currentColorPtr);
glTexCoord2fv((GLfloat*)currentUVPtr);
glVertex3fv((GLfloat*)currentCoordPtr);
```

once for each vertex of the triangle.

If you use similar calls in application code, then it is probably best to guard against corruption of current state by using:

```
/* Save OpenGL state */
glPushAttrib(GL_CURRENT_BIT);

    /* Draw a point at the origin. */
    glBegin(GL_POINTS);
        glVertex3f(0, 0, 0);
    glEnd();

/* Restore to previous state */
glPopAttrib();
```

✎ RenderWare Graphics does not actually track the current vertex/color/texture coordinate/etc internally so pushing and popping these particular states is likely to be unnecessary but is done here to demonstrate good practice.

1.5.2 OpenGL Vertex Arrays

Under OpenGL, RenderWare Graphics' retained mode pipelines (the atomic and world sector pipelines) submit geometry via OpenGL's standard vertex array constructs. Vertex arrays are enabled and disabled via the OpenGL functions **`glEnableClientState`** and **`glDisableClientState`** respectively.

If geometry is submitted application-side via vertex arrays, then it should be ensured that all client states that are enabled are disabled again once the vertex arrays have been dispatched to OpenGL. It is recommended that this is achieved using the following coding cliché:

```
GLfloat myVerts[3] = { 0, 0, 0 };

/* Save OpenGL state */
glPushClientAttrib(GL_CLIENT_VERTEX_ARRAY_BIT);

    /* Set up and dispatch vertex arrays.  For example… */
    glVertexPointer(3, GL_FLOAT, 0, (const Glvoid *)myVerts);

    /* Draw a point at the origin. */
    glDrawArrays(GL_POINTS, 0, 1);

/* Restore to previous state */
glPopClientAttrib();
```

An alternative to the OpenGL vertex array API is a state caching wrapper that RenderWare Graphics provides and uses in its standard world and plugin code. This is described in *1.5.4 State caching vertex array wrapper*.

However, there are circumstances under which the code displayed above will not give the expected results – namely, when the OpenGL driver implements the vertex array range extension.

1.5.3 The Vertex Array Range Extension

RenderWare Graphics' retained mode pipelines have been augmented to support NVIDIA's Vertex Array Range (VAR) extension that gives a dramatic performance increase in geometry throughput where available. RenderWare Graphics utilizes this extension to allocate the vertices of atomics and world sectors in AGP memory or video memory. This allows vertices to be fed to a hardware transform and light unit far faster than if they had to be transported from system memory. This section explains how to utilize this extension inside a RenderWare Graphics application.

✎ If the NVIDIA VAR extension cannot be initialized for any reason, RenderWare Graphics falls back to system memory vertex arrays. Unless you are writing rendering pipelines, you do not need to know where vertex arrays are being allocated.

When **`RwEngineStart`** is called by the application, RenderWare Graphics allocates a 4MB (by default) pool of memory for storing these vertices if the extension is available. The size of this pool is variable, however, and can be set via the function:

```
void
_rwOpenGLVertexHeapSetSize( const RwUInt32 size );
```

This function must be called before **`RwEngineStart`** and takes a single parameter representing the desired size of the memory pool in bytes.

✎ Albeit unlikely, an application can completely exhaust the vertex array pool during run-time. If this is due to large (or many) static vertex arrays resident in the pool then the only method to avoid this is to compile your application with an increase to the pool size using

_rwOpenGLVertexHeapSetSize. If this is due to large (or many) dynamic vertex arrays in the pool, RenderWare Graphics will try to adjust for this dynamically by recreating the pool at *double* its previous size at the end of the current frame.

All the following RenderWare Graphics VAR functions assume that the VAR extension is supported. To ensure this, always test **_rwOpenGLVertexHeapAvailable()** before executing any of the functions.

If vertex arrays are used application-side when this extension is enabled, they must point to data allocated from the vertex heap. As a result, the example code using vertex arrays to render a point at the origin in the last section will need to be modified to work on all drivers. This is shown below:

```
void *vertHeapVerts;
GLfloat myVerts[3] = { 0, 0, 0 };

/* Check for the existence of the vertex heap */
if ( _rwOpenGLVertexHeapAvailable() )
{
    RwInt32 memSize;

    /* Calculate the amount of vertex heap memory required */
    memSize = numVerts * sizeof(myVerts);

    /* Allocate a dynamic block in the vertex heap
     * for the vertices */
    vertHeapVerts = _rwOpenGLVertexHeapDynamicMalloc(memSize,
                                                     TRUE);

    if (vertHeapVerts == NULL)
    {
        /* No mem!!  Handle this sensibly… */
    }

    /* Transfer our vertices to the vertex heap */
    memcpy( vertHeapVerts, systemRamVerts,
            numVerts * sizeof(MyVertexDesc) );

    myVerts = (GLfloat*)vertHeapVerts;
}

/* Save OpenGL state */
glPushClientAttrib(GL_CLIENT_VERTEX_ARRAY_BIT);

    /* Set up and dispatch vertex arrays.  For example… */
    glVertexPointer(3, GL_FLOAT, 0, (const Glvoid *)myVerts);

    /* Draw a point at the origin. */
    glDrawArrays(GL_POINTS, 0, 1);

    /* Set the NV fence to ensure no memory overwrites */
    _rwOpenGLVertexHeapSetNVFence( vertHeapVerts );

/* Restore to previous state */
glPopClientAttrib();
```

After this code is called, the vertex array in question is defined both in system memory and the vertex heap. However, only the vertex heap vertex arrays should be specified via **`glVertex/TexCoord/Color/NormalPointer`** and dispatched via **`glDrawArrays`** or **`glDrawElements`**.

There are two types of vertex heap memory blocks that can be obtained: static and dynamic. The above example requests a dynamic block.

The difference between the two is that static blocks remain in the vertex array until explicitly freed. Dynamic blocks are allocated on request but may be freed internally to allow future block requests, both static and dynamic. Dynamic blocks can also be freed explicitly. The free interfaces are:

```
void
_rwOpenGLVertexHeapDynamicFree( void *videoMemory );

void
_rwOpenGLVertexHeapStaticFree( void *videoMemory );
```

Note that for a dynamic block to be freed internally by the memory manager, the block must first be discarded. This avoids problems in which a dynamic block is still in use but the memory manager wants to free it and is unable to inform the application of that action. To discard a dynamic block, use

```
void
_rwOpenGLVertexHeapDynamicDiscard( void *videoMemory );
```

The NVIDIA fence extension is also supported internally by RenderWare Graphics. If the extension is not supported by the host video card, the fence related functions do nothing.

Fences force vertex array blocks to finish rendering before releasing their video memory. By default, statically allocated blocks do not have fences; dynamic blocks are optional via the second parameter to the **`_rwOpenGLVertexHeapDynamicMalloc`** function.

```
void *
_rwOpenGLVertexHeapDynamicMalloc( const RwUInt32 size,
                                  const RwBool generateFence );
```

If a vertex array block has a fence then it should be set soon after its vertex data is submitted to OpenGL. This is achieved through the function **`_rwOpenGLVertexHeapSetNVFence`**, as in the example code above. Free'ing a vertex array block will also force the fence to finish before freeing the video memory.

The RenderWare Graphics for OpenGL VAR memory management system will nearly always catch the NVIDIA fences. However, if the application wants to update the vertex data faster than the GPU can render it, there is a synchronization problem. If this is the case, the function

```
void
_rwOpenGLVertexHeapFinishNVFence( void *videoMemory );
```

can be used to force the application to wait on the GPU to finish rendering. Note that this will most likely cause a stall and reduce performance.

1.5.4 State caching vertex array wrapper

A group of functions and macros that wrap the OpenGL vertex array API can be found in **`rwcore.h`**. In a debug SDK, the wrapper appears as a set of functions while in a release SDK, these are exposed as macros to achieve the greatest performance by inlining their code.

The vertex array caching system functions by

1. Not pushing/popping attributes.
2. Enabling a client state when that state's data exists, and only disabling that client state when its data is no longer required.

Hence, for example, if all geometry contains vertex colors throughout the lifetime of the application, **`glEnableClientState( GL_COLOR_ARRAY )`** is called just once, and **`glDisableClientState( GL_COLOR_ARRAY )`** is called only when the application shuts down.

Note that if **`glEnableClientState`** or **`glDisableClientState`** are called *outside* of the wrapper, a conflict may occur with the internally cached state and program execution is not guaranteed to be correct.

The wrapper handles the following vertex data

- Positions
- Normals
- Vertex colors
- Texture coordinates (in all available texture units)

and provides access to system, NVIDIA Vertex Array Range (VAR), and ATI Vertex Array Object (VAO) vertex arrays.

Due to the functionality difference between VAO and the other vertex array types, there is a slightly different wrapper interface. However, the underlying state caching system is available to all.

The system and VAR wrappers require a memory pointer (called **`memAddr`**). This is the address of the beginning of the state data.

The VAO wrappers require a VAO name (as provided by the VAO extension functions) (called **`vaoName`**) and an integer offset (called **`offset`**) into that VAO to the beginning of the state data.

The requirements for the arguments to the wrapper functions are the same as those to the OpenGL vertex array functions. Refer to the OpenGL Red Book for details.

All wrapper functions that specify vertex data require an enable test as their first argument. The exception to this rule is that vertex positions are assumed to be *always* present and therefore do not require an enable test.

Positions

Vertex positions are specified through

```
void
RwOpenGLVASetPosition( const RwUInt32 numComponents,
                       const RwInt32 baseType,
                       const RwUInt32 stride,
                       const void *memAddr );
void
RwOpenGLVASetPositionATI( const RwUInt32 numComponents,
                          const RwInt32 baseType,
                          const RwUInt32 stride,
                          const RwUInt32 vaoName,
                          const void *offset );
```

where

`numComponents` is the number of components in the position vector.

`baseType` is the OpenGL type of the position data.

`stride` is the stride, in bytes, of the vertex data to locate each position vector. Remember that a zero stride has a special meaning in OpenGL.

Normals

Vertex normals are specified through

```
void
RwOpenGLVASetNormal( const RwBool enableTest,
                     const RwInt32 baseType,
                     const RwUInt32 stride,
                     const void *memAddr );
void
RwOpenGLVASetNormalATI( const RwBool enableTest,
                        const RwInt32 baseType,
                        const RwUInt32 stride,
                        const RwUInt32 vaoName,
                        const void *offset );
```

where

`enableTest` is a Boolean expression that enables the normal vertex state if evaluated to TRUE, and disables it if evaluated to FALSE.

baseType is the OpenGL type of the normal data.

stride is the stride, in bytes, of the vertex data to locate each normal vector. Remember that a zero stride has a special meaning in OpenGL.

Colors

Vertex colors are specified through

```
void
RwOpenGLVASetColor( const RwBool enableTest,
                    const RwUInt32 numComponents,
                    const RwInt32 baseType,
                    const RwUInt32 stride,
                    const void *memAddr );
void
RwOpenGLVASetColorATI( const RwBool enableTest,
                       const RwUInt32 numComponents,
                       const RwInt32 baseType,
                       const RwUInt32 stride,
                       const RwUInt32 vaoName,
                       const void *offset );
```

where

enableTest is a Boolean expression that enables the color vertex state if evaluated to TRUE, and disables it if evaluated to FALSE.

numComponents is the number of components in the color vector.

baseType is the OpenGL type of the color data.

stride is the stride, in bytes, of the vertex data to locate each color vector. Remember that a zero stride has a special meaning in OpenGL.

Texture coordinates (first texture unit only)

Vertex texture coordinates for the first texture unit (or the only texture unit for those systems not supporting multitexturing) are specified through

```
void
RwOpenGLVASetTexCoord( const RwBool enableTest,
                       const RwUInt32 numComponents,
                       const RwInt32 baseType,
                       const RwUInt32 stride,
                       const void *memAddr );
void
RwOpenGLVASetTexCoordATI( const RwBool enableTest,
                          const RwUInt32 numComponents,
                          const RwInt32 baseType,
                          const RwUInt32 stride,
                          const RwUInt32 vaoName,
                          const void *offset );
```

where

enableTest is a Boolean expression that enables the texture coordinate vertex state (for texture unit 0) if evaluated to TRUE, and disables it if evaluated to FALSE.

numComponents is the number of components in the texture coordinate vector.

baseType is the OpenGL type of the texture coordinate data.

stride is the stride, in bytes, of the vertex data to locate each texture coordinate vector. Remember that a zero stride has a special meaning in OpenGL.

In a multitexturing enabled environment, you must ensure that the first texture unit is currently selected before using these functions.

Texture coordinates (arbitrary texture unit)

Vertex texture coordinates for an arbitrarily selected texture unit are specified through.

```
void
RwOpenGLVASetMultiTexCoord( const RwBool enableTest,
                            const RwInt8 activeTexUnit,
                            const RwUInt32 numComponents,
                            const RwInt32 baseType,
                            const RwUInt32 stride,
                            const void *memAddr );
void
RwOpenGLVASetMultiTexCoordATI( const RwBool enableTest,
                               const RwInt8 activeTexUnit,
                               const RwUInt32 numComponents,
                               const RwInt32 baseType,
                               const RwUInt32 stride,
                               const RwUInt32 vaoName,
                               const void *offset );
```

where

enableTest is a Boolean expression that enables the texture coordinate vertex state (for texture unit **activeTexUnit**) if evaluated to TRUE, and disables it if evaluated to FALSE.

activeTexUnit is the zero-based texture unit index of the texture unit you want to set the texture coordinates for. Note that this *must* be the currently selected texture unit.

numComponents is the number of components in the texture coordinate vector.

baseType is the OpenGL type of the texture coordinate data.

stride is the stride, in bytes, of the vertex data to locate each texture coordinate vector. Remember that a zero stride has a special meaning in OpenGL.

For additional information about multitexturing, see *1.6.2 Multitexturing*.

Disabling vertex client states

Although the functions shown above disable the appropriate client states when their corresponding **enableTest** arguments evaluate to FALSE, there may be occasions when it is useful to disable a client state separately from this interface. Therefore there are client state disabling functions that map to the caching system in the wrapper available.

```
void
RwOpenGLVADisablePosition( void );

void
RwOpenGLVADisableNormal( void );

void
RwOpenGLVADisableColor( void );

void
RwOpenGLVADisableTexCoord( const RwUInt8 texUnit );
```

Note that the same rule concerning the currently selected texture unit applies to **RwOpenGLVADisableTexCoord** as before.

Rendering vertex arrays

There is no wrapper needed for rendering vertex arrays specified using the above functions. Hence use **DrawElements** for indexed data or **DrawArrays** for unindexed data as usual.

Example

Consider vertex data to be arranged in an interleaved format containing unlit, unindexed 3D positions and a single set of texture coordinates. These vertices form individual triangles. The stride of this vertex data is

```
vSize = sizeof(RwV3d) + sizeof(RwTexCoords);
```

The position data begins at byte offset **0**. The texture coordinate data begins at byte offset **sizeof(RwV3d)**.

A memory buffer has been allocated (in either system or VAR memory) at address **vBuffer**. This is a *pointer* of type **RwUInt8**.

The following code snippet can be used to render the data:

```
RwOpenGLVADisableNormal();
RwOpenGLVADisableColor();
RwOpenGLVASetPosition( 3, GL_FLOAT, vSize, vBuffer + 0 );
RwOpenGLVASetTexCoord( TRUE, 2, GL_FLOAT, vSize, vBuffer +
                                                  sizeof(RwV3d) );
DrawArrays( … );
```

If the VAO extension were available instead, and a VAO allocated with the name **vaoName** then the following code snippet is equivalent to that above:

```
RwOpenGLVADisableNormal();
RwOpenGLVADisableColor();
RwOpenGLVASetPositionATI( 3, GL_FLOAT, vSize, vaoName, 0 );
RwOpenGLVASetTexCoordATI( TRUE, 2, GL_FLOAT, vSize, vaoName,
                          sizeof(RwV3d) );
DrawArrays( … );
```

Note that it generally useful to cache the offsets of each vertex element to avoid unnecessary computation. This is particularly useful for system and VAR vertex arrays.

1.5.5 World helper functions

In addition to the vertex array wrapper described in the previous section, a set of common functions in retained mode (world) are also provided to help the pipeline author. These can be found in **rpworld.h**.

For material setup

```
void
RpOpenGLWorldSetMaterialProperties( const void *materialVoid,
                                    const RwUInt32 flags );
```

that, given a dynamically lit scene, sets the ambient and diffuse material properties from an **RpMaterial**. This function is associated with the cached state of **rwGL_COLOR_MATERIAL**. (See *1.3.1 State enable/disable*.)

```
void
RpOpenGLLightSetAttenuationParams( void * const voidLight,
                                   const
                                   RpOpenGLLightAttentuation
                                   *params );

RpOpenGLLightAttentuation
RpOpenGLLightGetAttenuationParams( const void * const
                                   voidLight );

void
RpOpenGLLightSetSoftSpotExponent( void * const voidLight,
                                  const RwReal exponent );

RwReal
RpOpenGLLightGetSoftSpotExponent( const void * const voidLight );
```

These functions set lighting parameters for **RpLight**s.

1.6 Using OpenGL Extensions

There are many extensions to OpenGL that RenderWare Graphics does not expose as a platform independent API. However, it is still possible to use them from within a RenderWare Graphics application.

Note that RenderWare Graphics exposes a number of ARB and non-ARB extensions in the **`RwOpenGLExtensions`** structure. This is defined in **`rwcore.h`**. A global stealth variable called **`_rwOpenGLExt`**, of this type, can be queried in any RenderWare Graphics OpenGL application.

Let us consider a simple example.

1.6.1 Example of Using an OpenGL Extension with RenderWare Graphics

OpenGL supports the majority of the blend modes that RenderWare Graphics exposes. However, some additional modes are exposed via the `GL_NV_blend_square` extension implemented by NVIDIA. This extension provides four additional blending factors: `SRC_COLOR` and `ONE_MINUS_SRC_COLOR` for source blending factors, and `DST_COLOR` and `ONE_MINUS_DST_COLOR` for destination blending factors.

After calling **`RwEngineStart`**, the application can call:

```
RwBool haveBlendSquare =
                RwOpenGLIsExtensionSupported(
                              "GL_NV_blend_square");
```

to ascertain whether the underlying OpenGL implementation supports the extension or not. If it does, then the application can call:

```
if (haveBlendSquare)
{
    glPushAttrib(GL_COLOR_BUFFER_BIT);

        /* …for example */
        glBlendFunc(GL_SRC_COLOR, GL_DST_COLOR);

        /* Render alternatively blended geometry here */

    glPopAttrib();
}
else
{
    /* Implement fallback effect here */
}
```

Obviously, this is a very simple example of using an extension. However, more complex uses are indeed possible. The windowing system bindings to OpenGL (`wgl`, `agl`, `glx`, etc) support the extraction of function pointers to additional functions that implement various extensions. These can be used in a RenderWare Graphics application also.

See OpenGL documentation for more information on extensions.

1.6.2 Multitexturing

The **`GL_ARB_multitexture`** extension is available as part of the core of OpenGL 1.2 and above, and is used where possible by the RenderWare Graphics system to improve performance.

RenderWare Graphics provides three additional functions that can be used in a single texture or multitexturing environment, but are most useful in the latter.

```
void
RwOpenGLSetTexture( RwTexture *texture );
```

`RwOpenGLSetTexture` takes an **`RwTexture`** and binds it to the currently selected texture unit.

```
RwUInt8
RwOpenGLSetActiveTextureUnit( const RwUInt8 textureUnit );
```

`RwOpenGLSetActiveTextureUnit` sets the current texture unit to the specified zero-based index. In a single texture environment, this function will do nothing. In a multitexturing environment, if **`textureUnit`** lies between 0 and **`_rwOpenGLExt.MaxTextureUnits`**, will set the current texture unit appropriately. To check whether this function has successfully changed the texture unit, the return value is the zero-based index of the current texture unit.

Note that this function sets both **`glActiveTextureARB`** and **`glClientActiveTextureARB`**.

```
RwUInt8
RwOpenGLGetActiveTextureUnit( void );
```

`RwOpenGLGetActiveTextureUnit` returns the zero-based index of the current texture unit.

Example

This code snippet sets up two textures if there are at least 2 texture units available, otherwise it will fall back to a multi-pass method using a single texture unit.

```
if ( _rwOpenGLExt.MaxTextureUnits >= 2 )
```

```
{
     /* set up other vertex data */

     RwOpenGLSetActiveTextureUnit( 0 );
     RwOpenGLSetTexture( myTexture1 );
     RwOpenGLVASetMultiTextureCoord( TRUE, 0, … );

     RwOpenGLSetActiveTextureUnit(1);
     RwOpenGLSetTexture( myTexture2 );
     RwOpenGLSetMultiTextureCoord( TRUE, 1, … );

     /* render */
}
else
{
     /* do a multi-pass method */
}
```