RenderWare Graphics

White Paper

RenderWare Graphics and Direct3D

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

RenderWare Graphics provides a wrapper for some of the Direct3D API and this white paper describes how to optimize your application to get the maximum performance from your PC.

Most of the code examples in this document use Direct3D 8 names, but they can apply to Direct3D 9 by changing the “8” to a “9”.

1.2 Direct3D Specific Coding Guidelines

Adding Direct3D calls to a RenderWare Graphics application is a simple task, although some guidelines should be adhered to:

- Predicate Direct3D specific code so as not to adversely affect builds for other targets/platforms. To do this, predicate on a platform-specific definition, declared in **rwcore.h**:

```
#ifdef D3D8_DRVMODEL_H

/* Direct3D 8 specific code */

#endif /* D3D8_DRVMODEL_H */
```

- Further to this, a source file that makes calls to Direct3D functions should include the following:

```
#ifdef D3D8_DRVMODEL_H

#include <d3d8.h>

#endif /* D3D8_DRVMODEL_H */
```

- Direct3D functions can not be called until the RenderWare Graphics engine has been started (i.e. after **RwEngineStart**, but before **RwEngineStop**, has been called). This is because the Direct3D device is not created until **RwEngineStart** has been called.

- When linking a Direct3D application that uses RenderWare Graphics, the link settings do not require **d3d8.lib** as this library is already incorporated into **rwcore.lib**. In the Direct3D 9 version of RenderWare Graphics, **d3d9.lib** is included in **rwcore.lib** while **d3dx9.lib** is included in **rpworld.lib**.

1.3 Setting Direct3D Render States

RenderWare Graphics calls various Direct3D functions to set render states. Many of these are called from within **RwRenderStateSet**. For example, calling:

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

will result in a call, internally, to:

```
RwD3D8SetRenderState(D3DRS_SHADEMODE, D3DSHADE_GOURAUD);
```

In RenderWare Graphics for Direct3D, render state and texture state changes are buffered until a "draw primitive" call is made, and then, only the necessary state changes are made. This means that if multiple calls to set a particular render/texture state have occurred since the last primitive was rendered, only the *last* instance is processed. This can considerably improve efficiency.

Note that in order to use the buffered (or lazy) updating of render states, the only functions that can be used to change the render states are **RwRenderStateSet** and **RwD3D8SetRenderState**.

Similarly, for texture state lazy updates, the only function that can explicitly be used is **RwD3D8SetTextureStageState**.

1.4 Direct3D Performance

This section gives you some recommendations to help you get the maximum performance from your application when using RenderWare Graphics for Direct3D. Most of these recommendations just highlight the differences between the PC and other platforms.

- **Enable back face culling**:

Most of the current video cards support back face culling in hardware. It costs nothing to enable and greatly reduces fill rate.

- **Alpha blending is expensive**:

Unlike other platforms, on the PC alpha blending is quite expensive. This is important when rendering particle systems. We recommend that any particles close to the camera should be culled, as they will be very expensive in fill rate.

- **Use mipmapping**:

Using mipmaps will reduce the amount of the video card texture cache used when rendering small triangles, improving performance. Use the **mipmap** example provided in the RenderWare Graphics SDK to check which filter modes gives you more frames per second on different video cards. You will notice that using **rwFILTERMIPLINEAR** is much faster than **rwFILTERLINEAR**.

- **Use compressed textures**:

Compressed textures are faster to load from disk and faster for rendering. Direct3D supports the formats **DXT1** to **DXT5**. Each of these formats have different characteristics, the most important being the number of bits used for the alpha channel. RenderWare Graphics supports reading compressed textures from **DDS** files using the function **RwD3D8DDSTextureRead**. If a video card does not support compressed textures, this function decompresses the texture automatically to the most suitable pixel format. In addition, compressed textures are supported on platform-specific texture dictionaries.

- **Palettized textures**:

Direct3D only supports one active palette at a time, so when using multitexturing both textures *could* reference the same palette. That means the **RpMatFX** plugin on Direct3D has to use two passes, instead of just one, if both the base texture and the effect texture are palettized. In addition, in some video cards, when using a non-mipmapped linear filter, rendering a geometry with palettized textures may be slower than with non-palettized ones. Most modern video cards do not support any form of palettized textures. Some hardware manufactures are even removing the support for them on older hardware in the latest versions of their drivers.

- **Batch, batch, batch**:

Nowadays, it may be slower to set up the rendering and to send the information to the video card than the actual rendering itself. So, try to group your triangles and to

submit them in big chunks. Rendering less than 100 triangles at a time is a waste of CPU.

- **Use indexed triangle strips**:

Modern video cards have a vertex cache where they temporarily store the results from the transformation and lighting of a vertex. On subsequent vertex transformations, the video card checks if the previous results are already in the cache. Indexed primitives should be ordered to reuse previously used vertices as much as possible. This vertex cache only works if using indexed primitives.

However, some video cards require an upload of the indices into video memory every time an indexed primitive is rendered. In many situations, the main CPU bottleneck is inside the function that sends the index data across the bus.

In order to reduce the bottleneck when sending the index data, while maximizing the performance of the vertex cache, it is highly recommended that you use cache aware indexed triangle strips. RenderWare Graphics provides the **`RtVCat`** toolkit to help you create these cache aware triangle strips.

- ***Z* buffer optimizations**:

Modern video cards implement several kinds of optimizations to remove pixels that are occluded by previously rendered ones, so fully occluded triangles are cheap to render. To help the video card remove occluded pixels quickly, render any objects that are close to the camera before the objects that are further away.

- **Only enable the alpha test when needed**:

Some modern video cards have a different hardware path when alpha test is enabled, effectively disabling any performance increase from Z buffer optimizations. We recommend that you only enable alpha test when really needed, for example, when the percentage of pixels that will fail the alpha test is greater than the percentage of pixels that will fail the Z test. You can disable the alpha test by setting the function to "always":

```
RwRenderStateSet(rwRENDERSTATEALPHATESTFUNCTION,
                 (void *)rwALPHATESTFUNCTIONALWAYS);
```

- **Avoid switching resources**:

Switching any active resource, including render targets, textures, vertex buffers, etc., is always expensive. It is usually faster to have one large texture per atomic, than to have many textures shared across different atomics, forcing RenderWare Graphics to switch the active texture for the same atomic several times. The PC has more system memory and video memory than other platforms, so uploading resources to video memory is not usually the main bottleneck.

- **Group and sort**:

In order to minimize the number of resource switches, try to group your objects by resource usage and render them in a sensible order. For example:

1.- Render all opaque world sectors, sorted front to back.

2.- Render all opaque atomics sorted by type (main characters, enemies, etc.) and front to back.

3.- Render all transparent world sectors, sorted back to front.

4.- Render all transparent atomics, sorted by type and back to front.

1.5 Direct3D Bottlenecks

There are many types of bottlenecks in a 3D application. This section provides guidelines on how to deal with them.

- **Fill rate**:

This is the main bottleneck in most 3D applications. An easy way of detecting if you are fill rate limited is just to resize the window to half the size and to compare the frames per second rates. If the application is running twice as fast, or more, than before, you are limited by fill rate.

Alpha blended triangles are the most common cause of fill rate problems, the other main cause is normally excessive amount of overdraw. Also be careful about the fill rate wasted on rendering to camera textures.

- **Vertex transformation and lighting (or vertex shaders)**:

This bottleneck was common when video cards did not support vertex transformation and lighting operations in hardware, but nowadays it is rare. Try to remove lights from the scene or use a lower level of detail in your 3D models. Also, try to optimize your indexed triangle strips to improve the usage of the video card vertex cache.

- **Pixel shaders**:

This bottleneck is rare unless you are rendering the whole scene using long and complex pixel shaders. If switching to a simple color operation, “base texture * diffuse”, increases your frames per second, then you should consider using a much simpler pixel shader for all objects on the background, or only complex color operations for the main character.

- **Texture access**:

The application is limited by the size of the textures and the filtering modes if, when scaling down all your textures to a quarter of the original size, the performance improves by a proportional scale factor. Try not to use trilinear or anisotropic filters in those situations. It is also better to scale down the textures depending on the video card. If there are occasional slowdowns depending on the camera position and orientation that disappear when using smaller textures, then it is possible the textures do not fit in the video card memory and Direct3D is constantly uploading and switching textures across the bus. Always consider using compressed textures.

- **CPU**:

This bottleneck is very common in 3D games when updating bone hierarchies, updating procedural geometry, etc. Use a good profiling application to check which portion of code the CPU is spending most of the time processing and try to optimize that code.

1.6 Vertex Shaders vs. VU code

When porting your application from the PS2 to PC, or vice versa, you need to consider the differences between Vertex Shaders and VU code. These differences do not affect the creation of your assets, but they do have an effect on the implementation of the rendering pipelines for each platform.

Vertex Shader definition:

- Vertex shaders only work at the vertex level, without any knowledge of previously processed vertex or connectivity information. The only data accessible from a vertex shader is the per vertex information you wrote to the vertex buffer and the vertex shader constants.

- The global data accessible inside a vertex shader (for example, light colors, skin bones, material color, etc.) is stored in an array of 4D floating point vectors. The size of this array is video card dependent. The minimum size of the array is 96 constants and the maximum value supported today is 256 constants. This range is fine for static objects, but it's quite low when rendering skinning characters with a possible minimum value of just 28 bones if using a three vectors per bone matrix.

- You can not create geometry, or change render states, inside a vertex shader. A vertex shader program just transforms the current vertex from a specific 3D space to an homogeneous clipping space, and passes texture coordinates and vertex color information to the pixel rasterizer.

VU code definition:

- The PlayStation2 VectorUnit1 (VU1) micro programs work at the batch level. A batch is a collection of vertex attribute data which is able to fit into the VU1 micro memory.

- The VU1 has 16KB of micro memory, and can be divided into any buffering form the micro program desires. This improves the parallelism of the rendering pipeline.

- Extra data may also be uploaded into micro memory allowing constants and render states to be used dynamically whilst the micro program is executing. Typical data uploaded in this way is lighting information, culling and clipping information and transform matrices.

- The micro program itself isn't executed per vertex but as a stand alone co-processor program with condition tests and jumps, capable of dual executing vector instructions of quad words (xyzw) and integer operations. This greatly increases processing features and complexity, allowing data explosion (generation), mesh refinement and procedural mesh generation.