Migration from software rendering to 3D accelerated graphics using OpenGL ES

Khronos Kolumn

Copyright © 2009, Imagination Technologies Ltd. All Rights Reserved.

This publication contains proprietary information which is protected by copyright. The information contained in this publication is subject to change without notice and is supplied 'as is' without warranty of any kind. Imagination Technologies and the Imagination Technologies logo are trademarks or registered trademarks of Imagination Technologies Limited. All other logos, products, trademarks and registered trademarks are the property of their respective owners.

Filename	:	Migration from SW to 3D HW.Khronos Kolumn.1.5f.External.doc
Version	:	1.5f External Issue (Package: POWERVR SDK 2.05.25.0804)
Issue Date	:	07 Jul 2009
Author	:	POWERVR

Contents

List of Figures

List of Equations

Error! No table of figures entries found.

List of Code

Error! No table of figures entries found.

1. Introduction

Hardware (HW) rendering provides many advantages over software (SW) rendering, the biggest of which are covered in here; but also there are some points that need to be understood as you make that change. As a hardware graphics technology company, developing POWERVR technology, Imagination Technologies has spent a great deal of time easing migration for clients from SW rendering to HW rendering. This article attempts to cover the major implications we've discovered of switching, and to point out the most important lessons involved.

1. HW rendering is not simply faster SW

1.1. Performance

1.1.1. Parallel Processing

With HW accelerated 3D rendering, you have two processors to execute your software; the HW renderer takes all the load of rendering away from the CPU, freeing up CPU time.

The CPU must still calculate transformation matrices, and set up state in and send render commands to the graphics processor to make it do work, but all the calculations necessary to render polygons are removed from the CPU – and that's a big load. If the HW renderer also has HW vertex-transform capability, that's another big chunk removed from the CPU workload.

All the resulting spare processing power of the CPU can be put to good use for more complex physics, AI or other effects, or simply to allow the program to run at a higher frame-rate. Or alternatively, the program could leave the CPU and/or graphics processor partially idle, saving on battery power.

1.1.2. Specialized Rendering Hardware

If, in a SW rendering system, half the CPU time is spent rendering, then in a HW rendering system, that workload is removed. If the HW can render the graphics just as fast as the CPU could, this means the program could run at twice the frame-rate. (It wouldn't actually be exactly double – the two processors do need to spend some small amount of time communicating.)

Realistically, the graphics chip, being a specialised processor, will be much faster than the CPU was at rendering the scene; in other words, the app is running at double the frame-rate, but the graphics processor is idle for some percentage of each frame. This headroom allows the scene to become more complicated – higher polygon counts for your 3D objects, better effects, translucency... you could also render more objects, although this will in turn increase the load on the CPU, since more matrices need to be calculated, perhaps more physics and AI, etc – but you can handle that, because you've just removed half the CPU load.

Another benefit of the graphics processor is that it can and does have specialised caches, such as a texture cache and a geometry cache. Since all caches contain only relevant data, instead of everything going through the CPU cache, code executes faster.

1.2. Features

Strictly, SW rendering can always support more features than HW; this follows simply from the fact that the CPU is a general purpose processor while the graphics chip is specialised. However, it is rarely the case that this is true; it is simply not possible to achieve acceptable frame-rates for user-interaction while supporting even moderately advanced features in a software renderer.

For example a software renderer can use any method it likes to generate pixel values in the display; it could use a voxel engine, or even full-blown ray-tracing. Or it could render polygons using a standard rendering pipe approach, then use OS 2D support to draw text, for example, over the top. A HW renderer using the OpenGL ES 1.0 or 1.1 API's is restricted to a reasonably flexible but nonetheless fixed pipeline of transformed polygons and some number of blended textures, along with blending to the frame buffer.

However, rendering HW is rapidly gaining flexibility; with OpenGL ES 2.0 adding support for vertex shaders and pixel/fragment shaders, the possibilities of HW renderers are being massively opened up and already companies are debuting programmable shader based graphics technologies, like Imagination Technologies' POWERVR SGX, to take advantage of these OpenGL ES 2.0 capabilities. On the PC, this point has already been reached with OpenGL 2.0 and DirectX 9's shader model 3.

One feature that is highly unlikely to make it into a SW renderer is anti-aliasing; whereas it can be supported in HW as standard and optimised to work well. In SW, it's practically always going to be too expensive.

Another issue with features in software is that they must be implemented. Unless the SW engine is also using OpenGL ES or some other SW graphics library, any and all features that you desire in your SW engine must be implemented. With the HW accelerator, it's simply a matter of enabling them.

To quote from <http://www.khronos.org/opengles/>: “OpenGL ES 1.1 emphasizes hardware acceleration of the API, while OpenGL ES 1.0 focuses on enabling software-only implementations.”

1.3. Quality

Dedicated rendering HW supplies a certain base level of quality that would not be used in a SW renderer due to processing restraints.

Often the most obvious visual difference between SW and HW renderers is the existence of bilinear filtering. SW renderers written for real-time purposes usually cannot afford to spend time filtering texture-reads – it’s much quicker to carry out a simple array look-up, resulting in visibly blocky textures. Another feature is MIP-mapped textures, the purpose of which is to reduce noise as textures recede into the distance; SW renderers frequently do not want to spend the time deciding which MIP-map level to use. MIP-mapping is instead a performance *advantage* for HW renderers, and they can do bilinear filtering - and sometimes even trilinear filtering - for free.

If you do not enable MIP-mapping, you’ll get permanently visible noise in the distance; larger textures or the polygons moving further from the camera worsen this. If you do enable MIP-mapping, 33% extra texture memory will be used; sometimes, in some places, a texture may look a little blurry; but most importantly renders will be faster.

Platforms which use SW rendering are often fixed-point systems; indeed, sometimes a HW renderer is paired with a CPU which has no floating-point capability. However, a HW renderer is, internally, floating point. The ramification of this is that a HW renderer will produce higher accuracy results, with better blending and interpolation; and if there is also a HW transform unit it will achieve smoother vertex motion (no fixed-point quantisation errors) and avoid the ever-present danger of fixed-point maths under- or over-flows that can easily cause tricky to solve problems.

2. Coding Considerations

2.1. Use both processors

When using HW to render your scene, it is important to maintain the parallelism of the two processors in your system. Anything which forces them to synchronise will reduce your performance; for this reason is it best to draw your entire display using 3D. You should never use OS 2D functionality to draw to the render target, nor indeed any other CPU-based processing. In short: do not read or write to render targets using the CPU!

If you want to draw text, menu systems or any other UI element over the result of a HW render, use polygons; send each character of a string as a textured quad, or even build a dedicated texture for the word/sentence and render it with a single quad.

Anything which ‘locks’ the render target forces the CPU to sit idle until the graphics processor has completed the render; and after that, the graphics processor is sitting idle until the CPU gets around to submitting more polygons.

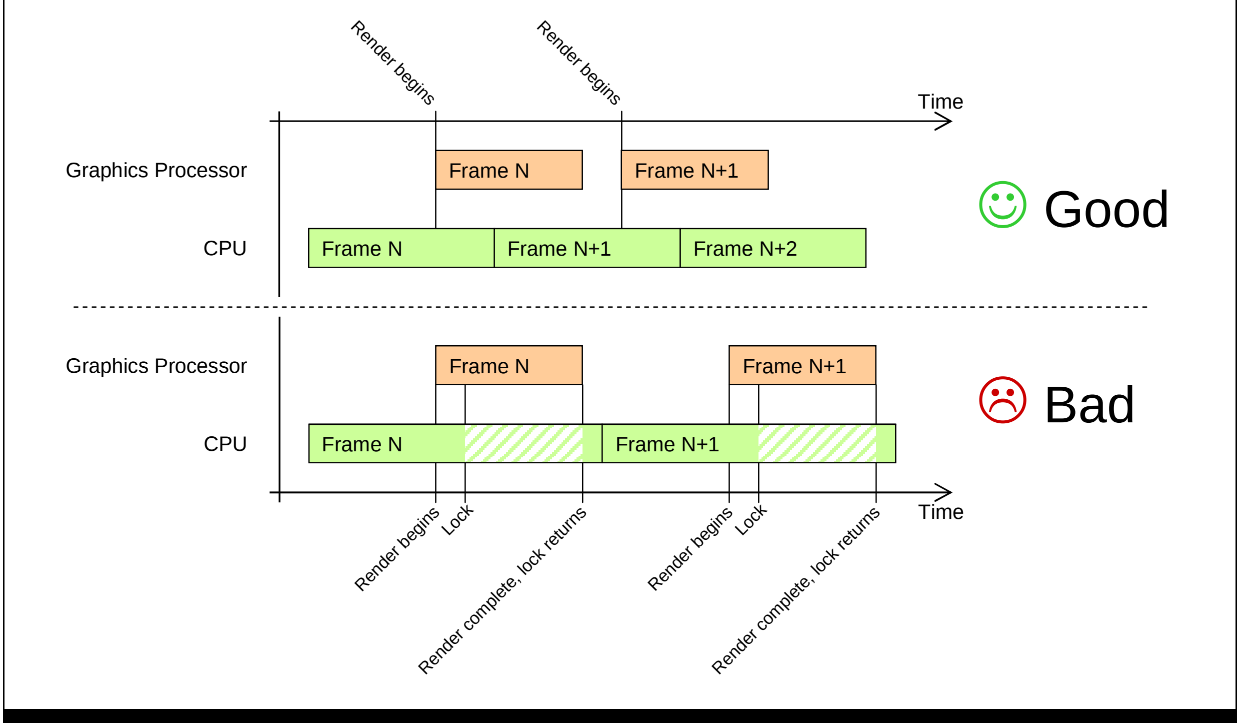

Figure 1 Graphics processor and CPU parallelism

This is illustrated in Figure 1. In the top chart, the CPU does not perform any actions which require CPU/graphics processor synchronisation. In the bottom chart, let us assume that the program renders a 3D scene, then uses the OS to draw 2D text on top: you can see that there’s a small amount of parallelism between the CPU and the graphics processor, then the CPU stalls waiting for the render to complete, then continues to draw the text; and the graphics processor is idle for approximately half the time. It is plain to see that, in this example, exploiting parallelism is allowing three frames to be rendered in less time than it takes to render two in the situation where the two processors must synchronise.

A HW renderer is relatively slow to render an individual triangle. What makes it fast is that it is a long pipeline, and can stream many triangles through it at once; it has a high latency, but also a high bandwidth.

It is important to realise that in a system with a dedicated graphics accelerator, you are dealing with a parallel processing computer.

2.2. Primitive Batching

With HW rendering it is more important to submit fewer, larger groups of polygons to the rendering API, e.g. OpenGL ES. This is partly an effect of the long pipeline of the renderer; the more work it is given in each batch of primitives, the better the performance achieved.

Another way of saying that is to consider minimising the number of calls to the rendering API, which reduces CPU work. It is not impossible to write an application the speed of which scales linearly with the number of 3D API function-calls made per frame.

So, for example, if you have a 500-triangle mesh, and you are submitting it as triangle-strips, there will come a point where the performance loss of API calls will outweigh the performance benefit of stripped triangles – if the average number of triangles per strip is five, a very good number, there are 100 primitive submissions. An easily visualised scenario is when rendering two-triangle strips, i.e. quads, perhaps for a UI. It may be faster to use a single call and render the mesh as a triangle list instead. These are example numbers and you should test your target platform to obtain real numbers.

2.3. Texture quality and performance

Use MIP-mapping – while it's up for debate whether or not it looks better (i.e. do you prefer the occasional blurred part, or noise in the distance?), it is faster on HW renderers. Also be sure to enable bilinear filtering, except for any textures where you specifically need point-sampling. For the current generation of accelerators, use trilinear filtering sparingly.

2.4. Experiment; test everything

When dealing with HW acceleration, sometimes it is hard to know which of two options will be better for performance. Try both – and benchmark the results.

3. Conclusion

The shift from SW to HW renderers is inevitable in all significant markets. Just as the mobile market is currently, the PC market went through this shift many years ago, and there are many lessons which have already been learnt, the major points of which could be boiled down to this: HW rendering is better, and always maintain graphics processor/CPU parallelism.

Author:

Aaron Burton, leading developer relations engineer, (POWERVR), Imagination Technologies.
Imagination Technologies is the leading supplier of 3D mobile graphics technology. Its POWERVR MBX IP is used in: Texas Instrument's OMAP2; Renesas' SH-Mobile3 and SH7770; Philip's Nexperia PNX4008; Freescale's i.MX31; and Intel's 2700G.