PowerVR

3D Application Development Recommendations

Filename	:	PowerVR. 3D Application Development Recommendations.mht
Version	:	1.2f (PowerVR SDK 2.03.23.1162)
Issue Date	:	13 Jun 2008
Author	:	PowerVR

Contents

List of Figures

1. Introduction

This document provides an initial introduction to developing 3D applications for PowerVR MBX platforms by providing a number of golden rules to obtain high quality and high performance 3D graphics. Additionally, in-depth performance recommendations are offered and porting guides from PC/console - hardware (HW) or software (SW) 3D engines - to PowerVR MBX are also included. Finally various online resources and contact details are provided.

1. Golden Rules

The following set of rules is essential to obtain high performance and high image quality when developing or porting applications to a PowerVR MBX enabled mobile platform:

1.1. Batching

Each API call introduces an amount of overhead. To reduce the CPU load, the number of API calls should be minimised. This can be achieved through the concept of batching - grouping operations together, as much as possible, into larger batches. Batching can be done on multiple levels, for example based on render state (e.g. keep all opaque objects together); texture (e.g. render all objects with the same texture in one go) - consider using a texture atlas (one large texture containing multiple sub textures) to achieve this; fog colours (aim to use the same colour or group all objects with the same colour); etc. Overall aim should be to reduce the number of API calls - so instead of 100 draw calls each drawing a single object, optimally aim to have only a single draw call which draws all 100 objects.

1.1.1. API Overhead

Between the application and the 3D accelerator is a driver layer which basically translates the industry standard API calls into a format understood by the hardware. This translation requires resources and thus results in extra CPU work for each API call issued. Given that the CPU is one of the possible limiting factors within the platform it is essential to try and minimise the number of API calls thus minimizing the CPU load created by API overhead.

One of the most critical paths for performance is the submission of geometry through a number of draw calls. For example the code below could be used to draw 250 butterflies with 16 different textures:

```
for (int i = 0 ; i < 250 ; i++ )
{
	glLoadMatrixx( ButterflyMatrix[ i ] );
	glBindTexture( GL_TEXTURE_2D,Texture[ i % 16 ] );
	glVertexPointer( ... );
	glDrawArrays( ... );
}
```

The above code contains a loop which contains four OpenGL ES API calls; each call will introduce an amount of API overhead. This specific example will result in 1000 API calls each of which has a performance cost.

To reduce the number of API calls required we need to introduce the concept of “batching”. Batching is in effect nothing more than grouping, instead of drawing each butterfly separately “batching” aims to group bits of geometry together in an effort to reduce the number of draw calls. A first level of grouping would be to draw all butterflies with the same texture in a single draw call. While this is an improvement it is not yet fully optimal. For this a second concept needs to be introduced which is a “Texture Atlas” or “Texture Page”. A texture atlas is a form of batching but specifically for textures, instead of having a single texture per butterfly it is possible to group multiple textures within one single larger texture. So for our 16 butterflies, rather than using 16 different textures each containing a single butterfly, it is possible to create a single large texture containing a grid of 16 butterflies. This approach can be summarised as follows:

- Use a single large texture containing all 16 butterfly textures. This is known as a “Texture Page” or “Texture Atlas”.
- Generate an array of texture coordinates once - which maps each of the 250 butterflies to the correct area of the page/atlas.
- For each frame update the vertex buffer using a software transform to move each butterfly to its correct position and apply the correct flapping of the wings.
- Bind the Texture Atlas Texture.
- Draw all butterflies with a single draw command.

Using the above approach it is possible to draw 250 butterflies with 16 different textures in a single set of four API calls resulting in a massively reduced CPU load. This concept is illustrated in Figure 1.

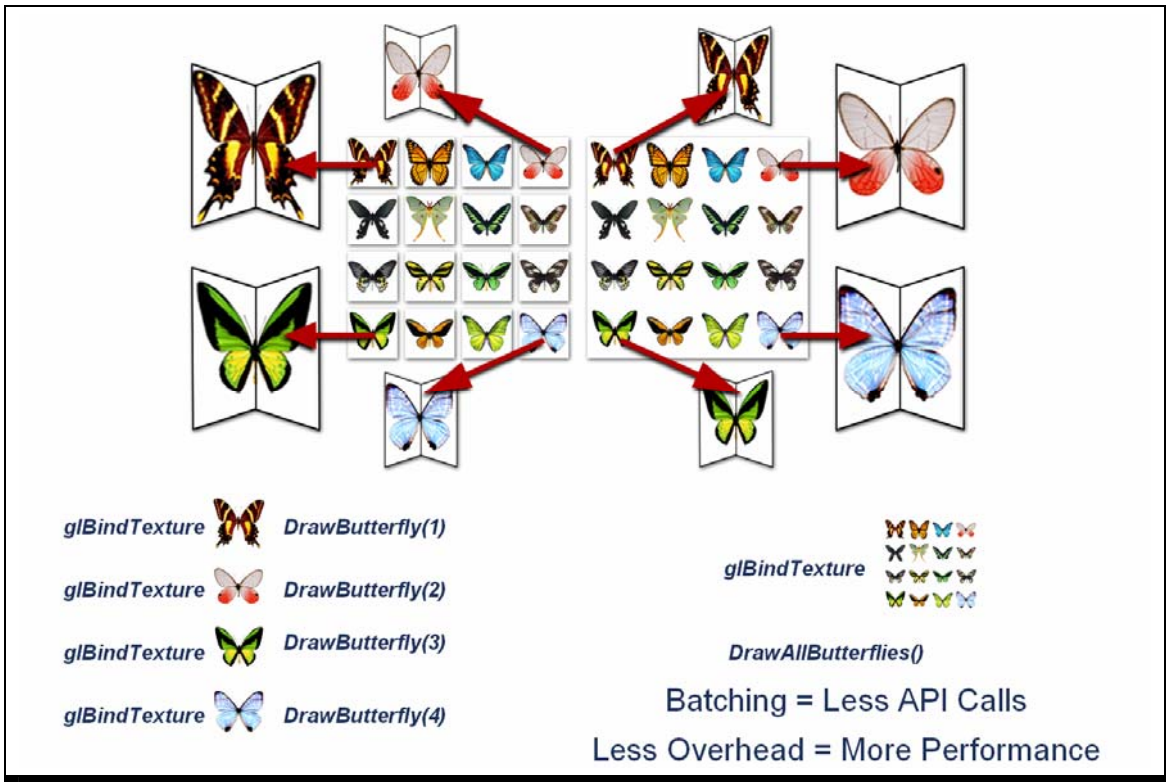

Figure 1: Batching Concept

1.2. Opaque objects must be correctly flagged as opaque

PowerVR MBX implements an advanced and highly efficient form of Hidden Surface Removal that ensures that pixels hidden behind opaque pixels are not processed by the hardware (an optimisation providing increased effective fillrate) thus offering valuable bandwidth and power savings. Maximum benefit is only possible when GL_ALPHA_TEST, GL_BLEND, GL_LOGIC_OPS (OpenGL ES naming standard) are enabled only when required – specifically when the object is not opaque. For optimal performance take into account batching: first render all opaque objects and at the end of the frame render all translucent objects.

1.3. Avoid mixing 2D and 3D operations within the same frame

For maximum 3D performance, 2D/OS graphical user interface (GUI) calls should be avoided since often these are implemented by or require interaction of the CPU with the frame buffer. This CPU interaction with the frame buffer, which should normally only be updated by the 3D graphics core, forces the synchronisation of the CPU and graphics processing unit (GPU) which will result in a massive loss of performance. 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.

This is illustrated in Figure 2. 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.

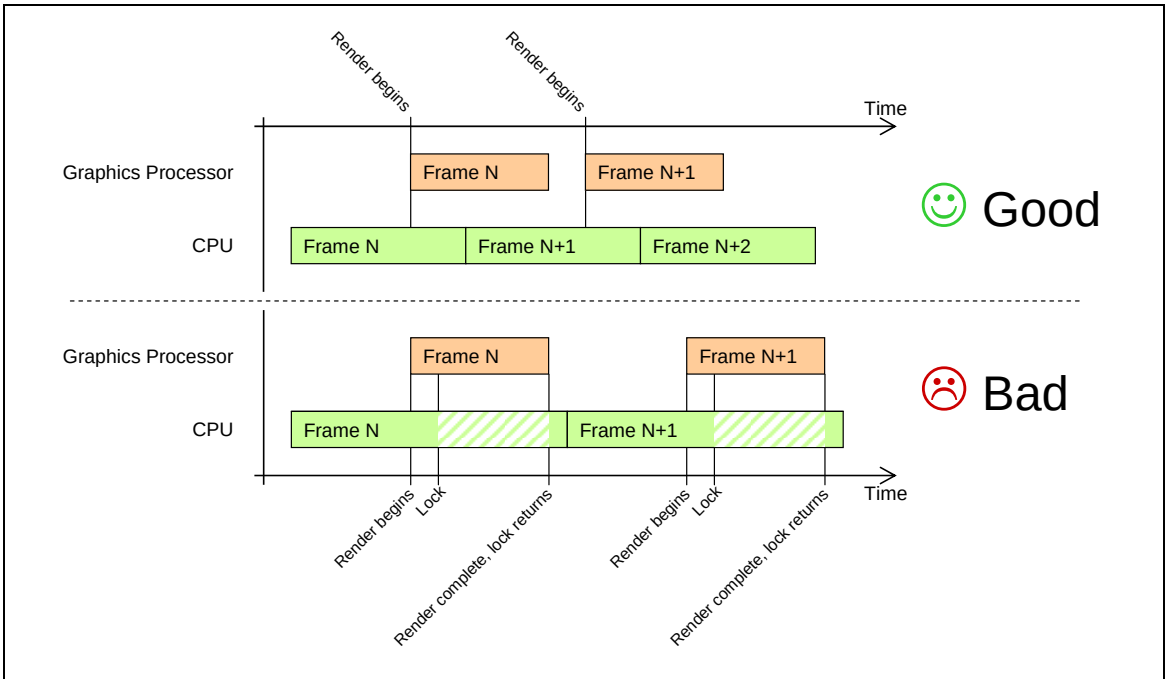

Figure 2 Graphics processor and CPU parallelism

Under normal circumstances the CPU and 2D/3D graphics core work in parallel but some API commands can break this parallelism resulting in stalling and wasted cycles for both the CPU and graphics core. To avoid this loss of valuable processing cycles avoid accessing colour or depth buffers directly. Specifically, in OpenGL ES, avoid glReadPixels(), glCopyTexImage2D() and glCopyTexSubImage2D().

1.4. Use PVRTC and bilinear filtering with MIP-mapping

Always aim to use compressed texture formats such as PVRTC2 and PVRTC4 on PowerVR MBX hardware, for best performance (due to reduced bandwidth requirements), optimum image quality, minimal storage requirements and distribution size. Palletised textures, while supported, should be avoided because they result in a loss of quality compared to PVRTC and a also loss in performance; 32-bit texture formats, such as RGBA8888, should be avoided since they are generally overkill on mobile platforms with 16-bit displays and result in a loss of performance due to increased bandwidth usage.

Always enable bilinear texture filtering with MIP-mapping, bilinear filtering is a free operation and MIP-mapping, while costing 33% more in terms of required storage, results in better image quality (less sparkling effects in the distance) and massively improved cache efficiency and thus a reduction in bandwidth usage.

1.5. Minimize mid-scene texture changes

Allocating and uploading new texture data for usage by the 3D core is a bandwidth intense operation; this means that changing the contents of an existing texture or creating a new texture during 3D rendering results in a perceptible reduction in performance. Therefore it is highly recommended to create and upload all required textures at the start of a game level or when the application loads.

1.6. Use vertex buffers if available

Driver-allocated vertex buffers (Vertex Buffer Objects in OpenGL ES) will allow optimal use of any memory transfer engine in the device.

On most PowerVR MBX systems you might not see a noticeable increase in performance using vertex buffers, but it will not degrade performance either. On newer generations of PowerVR hardware, using vertex buffers can make a noticeable difference in performance.

Regardless what platform you are targeting, using vertex buffers will make your application future proof and readily optimised for a wide range of devices and memory architectures.

2. 3D application development recommendations

2.1. Industry standard API usage

Obtaining maximum 3D performance requires access to the 3D hardware. While low-level direct access would be one way to achieve this is rarely a practical or cross-platform solution. To enable access to 3D hardware acceleration, two industry-standard application programming interfaces (APIs) have been created which specifically target 3D acceleration for mobile platforms.

OpenGL® ES, an API created by the Khronos Group, is a royalty-free, cross-platform API for full-function 2D and 3D graphics on embedded systems. The Khronos Group was founded in January 2000 by a number of leading media-centric companies and to date has more than 80 members, including Imagination Technologies, all dedicated to creating open standard APIs to enable the authoring and playback of rich media on a wide variety of platforms and devices. OpenGL ES uses a well-defined subset of desktop OpenGL, which means that any developer with OpenGL experience should have little to no trouble using this API. OpenGL-ES is supported under a large variety of operating systems including Windows CE (including Pocket PC and Windows Mobile 5.0), Linux, Symbian, uITRON, etc.

The Direct3D Mobile (D3DM) API from Microsoft will allow access to 3D hardware acceleration under Windows Mobile 5.0.

The PowerVR Family is fully compatible and compliant with both industry standard APIs allowing developers to quickly and easily unleash the massive 3D performance offered by the hardware accelerators.

Recommendations given in this article are applicable to both APIs, however OpenGL ES naming and syntax will be used throughout.

2.2. CPU arithmetic: Float versus Fixed Point

3D games require significant levels of arithmetic to create a compelling and interesting 3D world. Physics, collision detection, lighting, animation, etc. all boil down to a collection of mathematical functions that need to be evaluated.

Most mobile processors only offer native fixed point arithmetic operations, however an increasing number of them include an optional floating point co-processor such as the ARM Vector Floating Point (VFP) unit. The selection of floating or fixed point has a serious impact on the performance which can be decided as illustrated by the following example.

The PowerVR Butterflies demo shows a dynamic flock of butterflies (Figure 3). The demo was originally used on arcade level hardware and illustrates the alpha-blending capabilities of the hardware in combination with a large number of compressed textures.

Figure 3: PowerVR Butterflies Demo

The dynamic flocking algorithm is a physics style algorithm which imposes a serious amount of arithmetic on the CPU. Table 1 shows performance measured on an ARM 11 with VFP running at 300MHz for various version of the flocking algorithm.

Algorithm	FPU Support	FPS
Fully Floating Point	Software	72
Fully Fixed Point	Software	304
Fully Fixed Point with ASM Optimisations	Software	373
Fully Floating Point	Hardware VFP	415
Optimised Algorithm Fully Floating Point	Hardware VFP	1000+

Table 1: Flocking Algorithm Performance

The obtained performance measurements present a number of conclusions. First of all, using floating point arithmetic on a platform with no floating point support results in disappointing performance, in this specific case floating point is roughly five times slower than optimised fixed point and six times slower than VFP enabled floating point. Next the usage of assembly level optimisations in the critical paths can further increase the performance, for this specific example by roughly 25%. Finally and most important the usage of a fully optimised smart algorithm benefits all cases by a huge amount, while this level of optimisation was never required for the high-end arcade platform it is absolutely essential for optimal performance on a mobile device.

This poses a bit of a dilemma for multi-platform developers: selecting fixed point arithmetic leaves a possible VFP unit unused (gain of accuracy and performance) while selecting floating point arithmetic leaves the risk of unacceptable performance on some platforms. Luckily today's programming languages make it easy to abstract data types thus resulting in the possibility to select the most optimal data format at compile time. This approach allows developers to get the best of both worlds while only coding algorithms once.

2.3. Optimal Geometry Ordering

As discussed previously each API call results in a certain amount of overhead. The overhead is caused by a translation of standard API calls into commands for the hardware accelerator. The amount of overhead can be reduced by minimizing the translation work required by making sure that the data submitted to the API is already in a hardware friendly format. A large step in the right direction is to follow this simple recommendation when submitting geometry: submit sorted indexed triangles, with interleaved per-vertex data.

Using an indexed triangle list can be much more efficient than using strips. OpenGL ES and D3DM require a draw call per strip. For an average 3D model the number of triangles per strip is below 10 meaning that an object containing 1000 triangles would require 100 draw calls, one for each strip of 10 triangles. Using indexed triangles this object can be submitted in a single draw call resulting in much reduced overhead/CPU-load.

In addition, for future HW based on a post-vertex-shader cache, indexed triangle lists can provide a better vertex to triangle ratio than strips: strips have a best case of 1 new triangle per vertex submitted; pure indexing can do better through a higher amount of vertex re-use (the same vertex can be used for more than 3 triangles).

Vertex-sorting is a second step (after sorting the index array) which allows for improved memory access patterns and improves the usage of the various data caches. Both these concepts are illustrated in Figure 4.

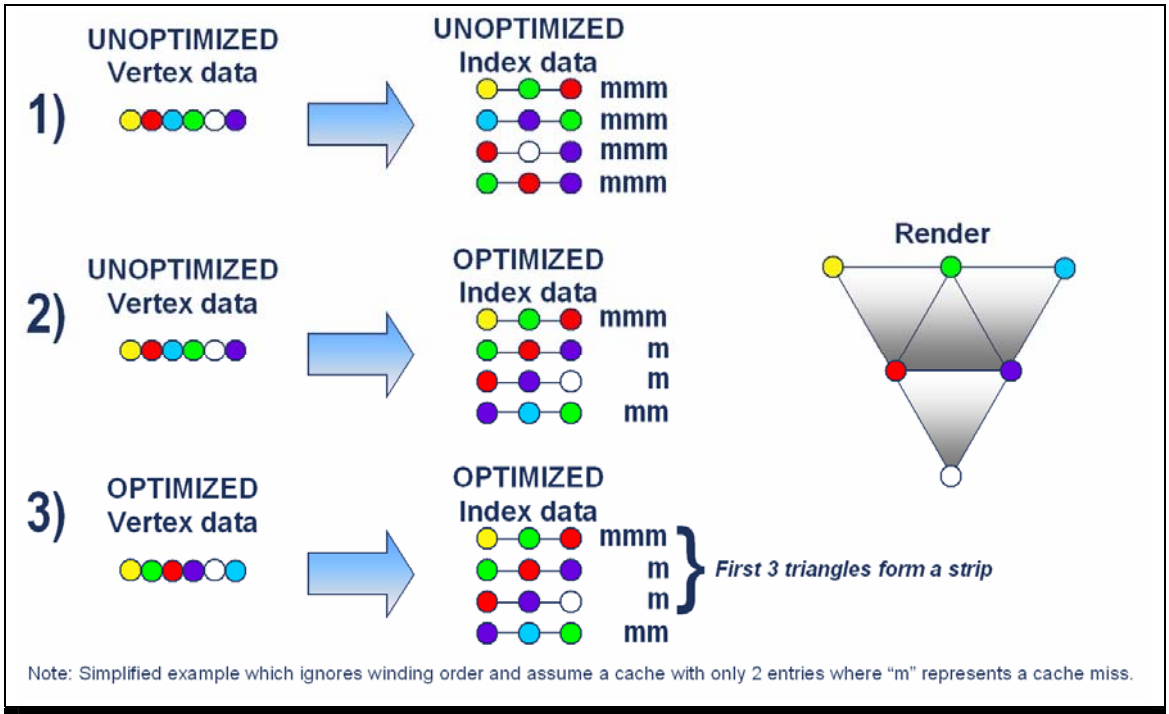

Figure 4: Advantage of using Sorted Indexed Triangle Data

When processing a vertex all per vertex elements are required; OpenGL ES allows these data elements to be spread out over memory thus resulting in scattered reads to fetch the required data. On the other hand it is also possible to interleave the data, such that data that belongs together is already together in memory. By submitting interleaved data, memory access patterns and driver overhead are improved. This concept is illustrated in Figure 5. (This is not an issue on D3DM, which always uses a single stream of interleaved vertex data in a Vertex Buffer.)

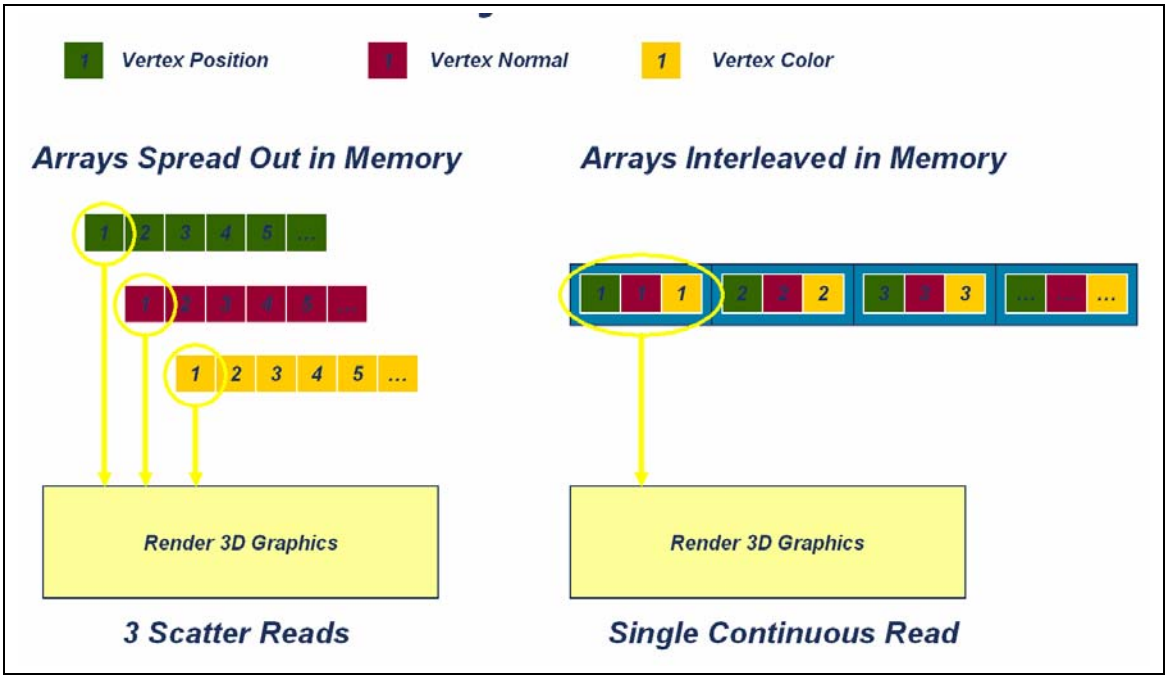

Figure 5: Interleaved Vertex Data

For peak performance use sorted "strip ordered" indexed triangles where connected indexed triangles can build strips. This further improves bandwidth efficiency of driver and hardware.

Triangle sorters to achieve these recommendations are available in the SDKs available from the PowerVR Developer Relations website.

2.4. Simplify Geometry

To further optimise geometry submission consider using a polygon reduction algorithm (e.g. when porting a game from PC to Mobile) to reduce the complexity of the object. Consider moving the detail from the vertex side to the pixel side e.g. through DOT3 per-pixel lighting or high resolution textures representing the finer geometrical details. Look into reducing the size of per vertex components e.g. using a byte format instead of a float format. Make sure not to include unused vertex elements, e.g. a normal when no lighting is enabled, also do not store constants per vertex, e.g. colour of an object. Consider the usage of Level of Detail (LOD) to avoid processing 1000's of triangles for an object 10's of pixels on the screen. See Figure 6.

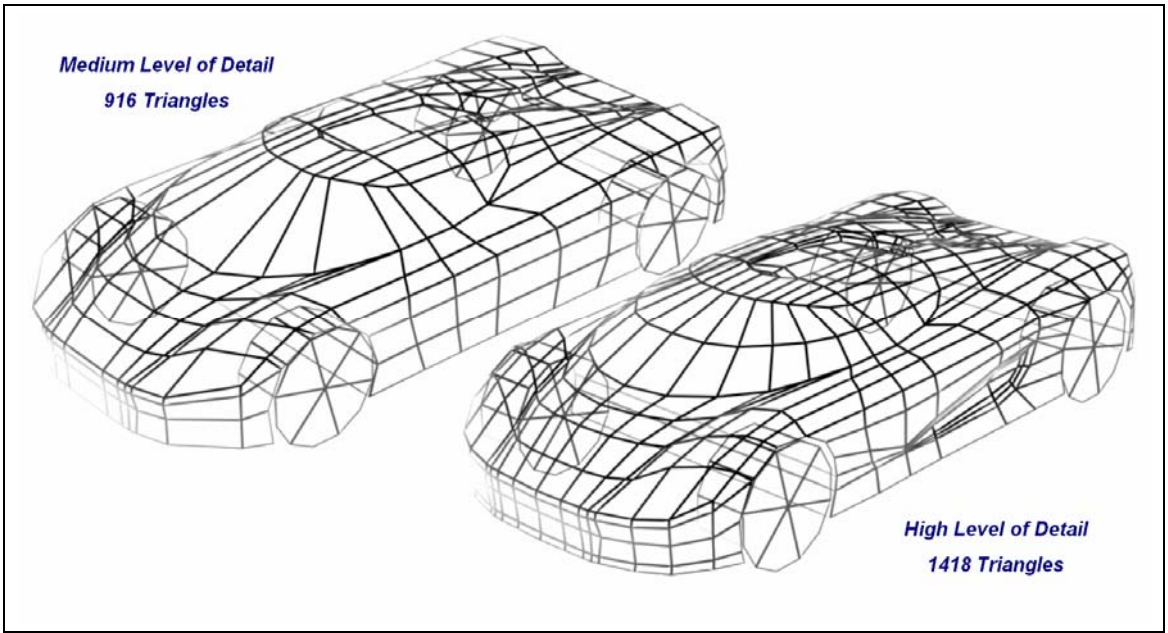

Figure 6: Level of Detail

2.5. Draw Order and Sorting

Another major advantage of PowerVR architecture is that the hardware handles Hidden Surface Removal (HSR) with peak efficiency irrespective of submission order, and so there is no need to sort objects from front to back. It is much better to create batches based on render state to minimize the amount of API overhead. The basic recommendation is to draw all opaque objects first, grouping them by the number of texture layers. For example first draw all dual textured opaque objects and then all single textured opaque objects. Finally draw all alpha blended and alpha tested objects last.

If possible sort objects by screen locality since this improves efficiency of the Tile Sorting Engine, e.g. avoid drawing a triangle bottom left and then top right of the screen. For improved memory access patterns aim to keep things together that belong together.

2.6. Hardware Transform and Lighting (VGP)

A hardware Transform and Lighting (T&L) unit, such as the PowerVR VGP, is a co-processor designed specifically for 3D graphics transformation and lighting calculations. Typically the VGP requires up to 10 times fewer clock cycles than a general purpose CPU to execute T&L arithmetic.

The VGP deals with all lighting and transformation arithmetic in parallel with the 3D core doing the rasterisation work and the CPU running the applications and operating system. The benefit of offloading this work from the CPU is illustrated in Figure 7.

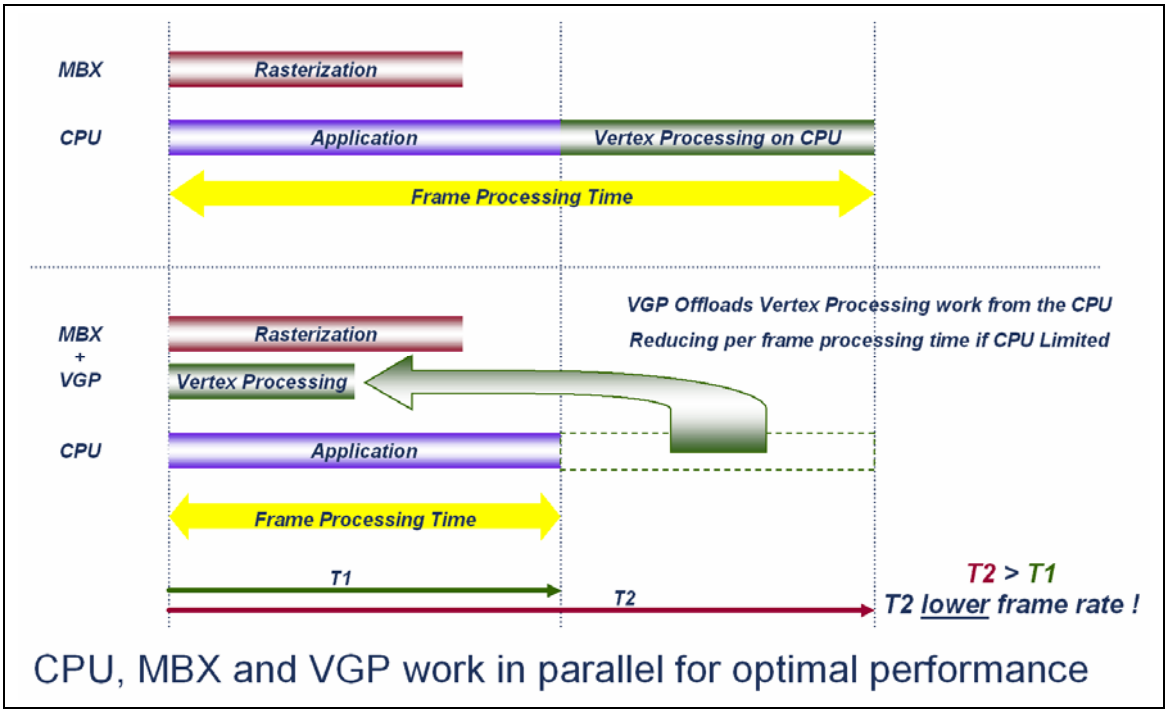

Figure 7: VGP Unit offloads work from the CPU resulting in higher performance

The PowerVR VGP unit is a fully programmable 4-way SIMD processor based on the Microsoft Vertex Shader 1.1 model. The VGP programmability offers maximum flexibility and maximum performance.

Because of the programmable nature it's essential to use optimised Vertex Programs for the different vertex processing cases required. Hacks and simplifications can be used if they look good enough (e.g. simplified lighting models). Make sure to use constants where possible.

The PowerVR SDK contains an optimising VGP compiler with detailed performance statistics which can be used to further tweak and optimise your vertex shader programs.

Note that not all PowerVR MBX platforms also have the PowerVR VGP.

2.7. Software Transform and Lighting (CPU)

On platforms with no hardware transformation and lighting unit such as the PowerVR VGP, this work is handled by the CPU. To avoid overloading the CPU it is important to be careful when using the rendering APIs transformation and lighting functionality.

OpenGL ES and D3DM lighting is quite complex and thus CPU heavy; implementations need to be conformant, so no shortcuts can be taken when implementing this functionality within the software layer of the API driver. It is thus recommended to use the simplest light type that works for your application and to use the fewest number of lights that produces acceptable results. For example parallel lights are cheaper than spotlights.

An alternative is to implement your own software lighting algorithm. This allows the developer to implement exactly the algorithm required and it's possible to take shortcuts and use hacks, as long as it does the job. Do verify that a custom implementation is faster and/or better looking than default OpenGL ES lighting.

Lighting can often be pre-computed offline and stored in a colour array or even textures. A static model with static lights does not require dynamic light calculations. Hence make sure to only enable lighting when needed, for example: on moving objects, or if the light properties are changing. Consider caching lighting calculation results if an object stays static for long periods of time.

Consider using pixel lighting techniques such as light maps or DOT3 per-pixel lighting. Figure 8 shows the difference between per-vertex and per-pixel lighting.

Figure 8: Vertex Lighting versus Per Pixel DOT3 Lighting

2.8. Texturing and Filtering

PowerVR hardware supports a special PowerVR Texture Compression (PVRTC) format, an advanced form of texture compression based on wavelet principles. Compressing texture data reduces storage requirements, bandwidth usage and distribution size of the application.

There are two levels of compression: PVRTC4 stores 4 bits per pixel while PVRTC2 uses only 2 bits per pixel. Whenever possible PVRTC2 should be used, if this highly compressed mode delivers unacceptable quality then PVRTC4 should be used. The high quality of PVRTC is illustrated in Figure 9. In the extremely rare case that PVRTC4 quality is unacceptable a 16 bit uncompressed format can be used. A 32-bit texture format is supported but usually this format is overkill given the limited colour depth of LCD screens.

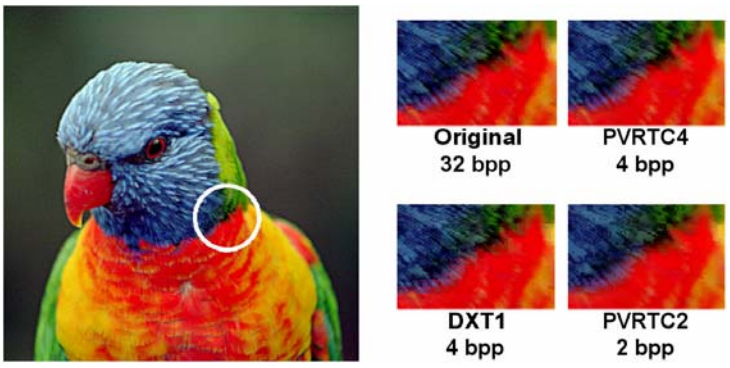

Figure 9: PVRTC Compression Quality using a Photograph

PowerVR hardware also supports several special texture formats (e.g. Luminance I8 and Luminance_Alpha IA88) which can be useful in a number of cases e.g. for shadows.

Use sensible texture sizes, there is generally little use for a 1024x1024 texture for an object that covers only a quarter of a QVGA screen. Do use large compressed textures for texture pages/atlas, up to the maximum supported resolution of 2048x2048.

Textures should be loaded up front before rendering, e.g. at level loading time. Creating or changing textures mid-action stalls the hardware and will cause stuttering performance. Especially avoid using a texture, then changing it, and using it again all within the same frame.

PowerVR MBX hardware supports two-layer multi-texturing. Multi-texturing should always be preferred over multi-pass rendering since this reduces the number of required draw calls, the vertex processing work, number of state changes and as a result the overall driver overhead and CPU load is also reduced. Using multi-texturing also avoids the risk of Z-fighting often seen with multi-pass rendering where the geometry in the two passes ends up at very slightly different depth positions (e.g. disabling lighting can result in a different driver path and hence potentially different depth values). Multi-texturing is illustrated in Figure 10.

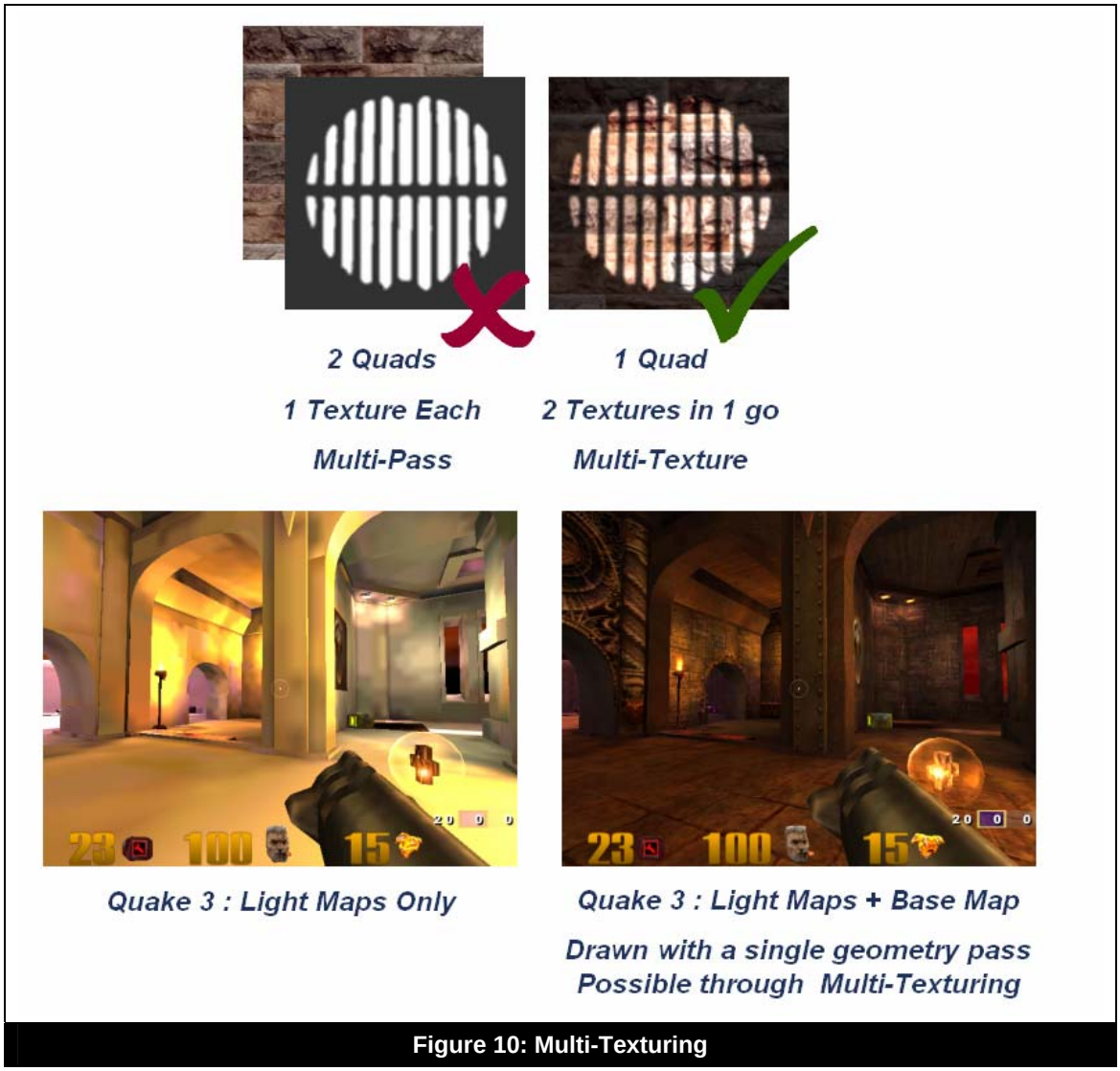

Figure 10: Multi-Texturing

Always use MIP-mapping in your applications: this prevents minified textures from causing “sparkling” artefacts. Not only will this result in better image quality there is also improved texture cache behaviour and higher performance. You get all this for just 33% more memory usage, which is a great trade-off. The best performance/quality filtering mode is bilinear filtering with MIP-mapping. This mode can be selected as follows in OpenGL ES:

```
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST);
```

Unlike software 3D engines bilinear filtering has exactly the same performance as point sampling and as such should always be used. Trilinear filtering generally has a higher fillrate cost but can be used when appropriate.

OpenGL ES keeps track of filtering settings per texture. Hence it is sufficient to set them once per texture; else the unnecessary API calls result in increased driver overhead and hence extra CPU load. Various filtering modes are illustrated in Figure 11.

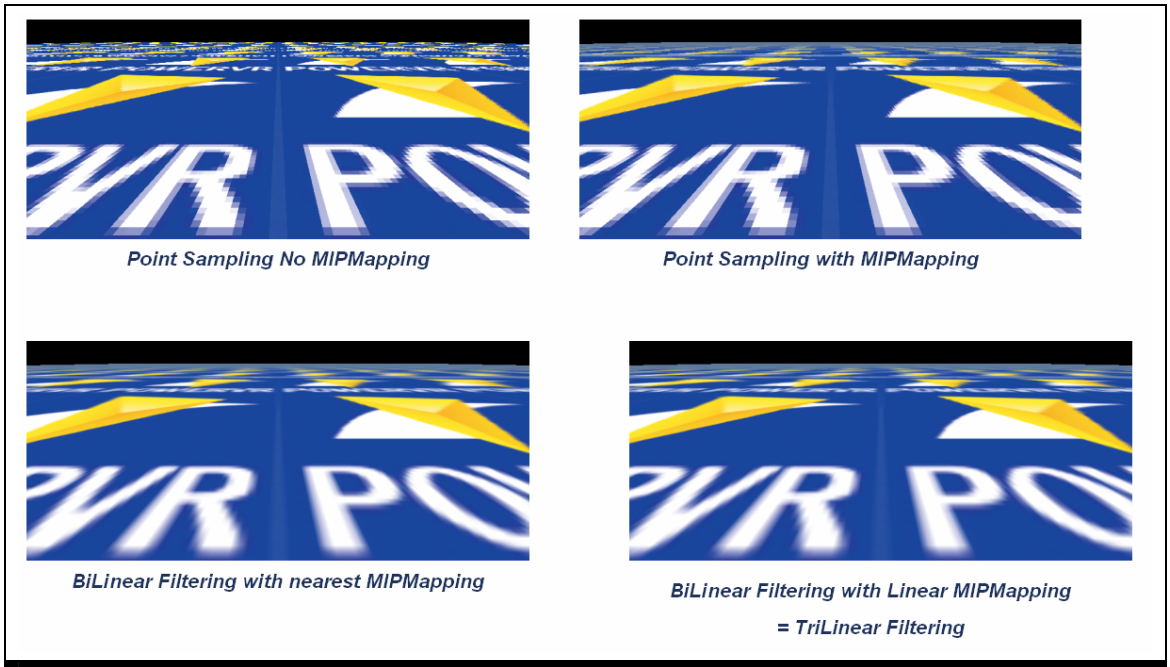

Figure 11: Texture Filtering Modes

2.9. Alpha Blending

PowerVR hidden surface removal uses opaque objects to reduce rendering cost by not processing pixels hidden behind opaque pixels – resulting in very high effective fillrate for the platform. However PowerVR HSR can only work when opaque pixels are correctly indicated by the API as being opaque, hence alpha blending and alpha testing render states should only be enabled when required. Only use alpha testing when necessary since it reduces the efficiency of all early hidden surface removal mechanisms, consider using only alpha blending.

When creating alpha-textures make sure to use a sensible background colour, avoid the usage of a very different “colour key.” Usage of a “colour key” will create an unwanted and ugly “halo” or “border” effect resulting from the texture filtering operation which bleeds the “colour key” into the visible texture area. This issue is illustrated in Figure 12.

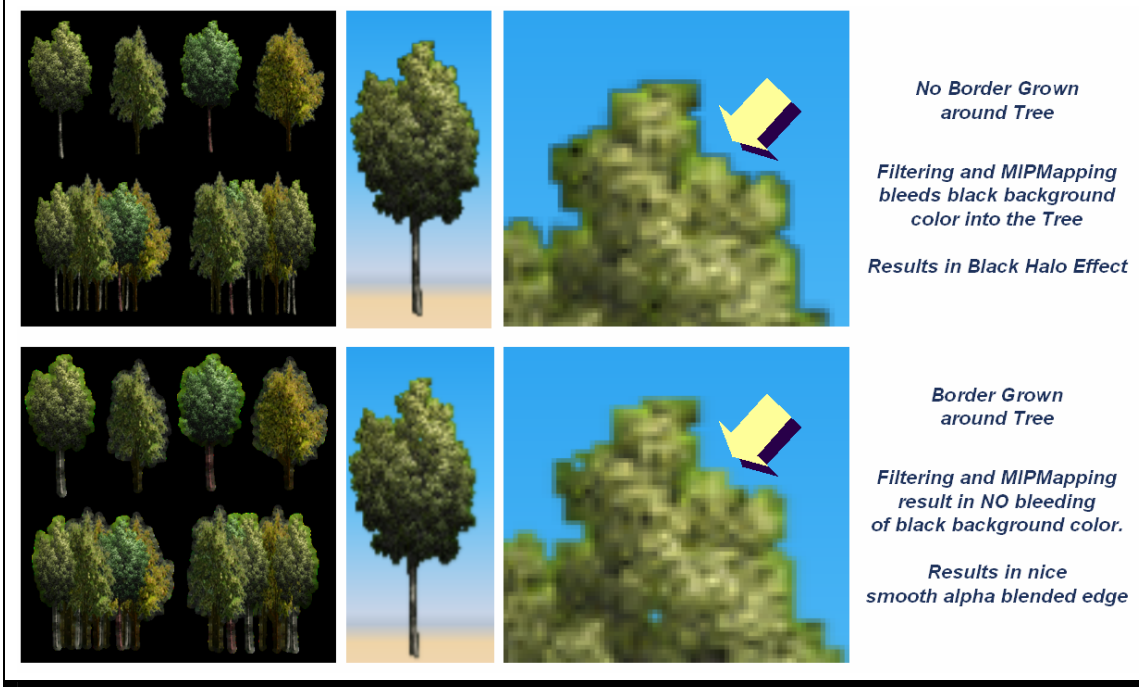

Figure 12: Alpha Blending with border growing

Halo and border effects can be avoided by “growing” the opaque colour into the transparent zones.

2.10. Power Consumption

While a dedicated hardware 3D core, such as PowerVR MBX, uses considerably less power than running a software 3D engine, the batteries in embedded devices can run out fast and recharging should occur as little as possible. On a high performance hardware 3D renderer it’s possible to create more frames per second than are really required (e.g. above LCD refresh rate or above a rate required for the application). To avoid wasting performance in these conditions it is possible to limit the 3D core peak frame rate and hence reduce the power consumption. In OpenGL ES 1.1 this can be done using the eglSwapInterval() function. For example setting a swap interval of 2 on a 60Hz display ensures the frame-rate cannot go above 30.

3. Porting Guidelines

3.1. Porting a 3D Application from PC/Console

When porting a 3D application from PC/console to a mobile platform it's essential to understand and take into account the differences between them. While mobile platforms are more restricted this does not make it impossible to port from PC/Console and end-up with stunning high-performance 3D graphics. However these limits do require extra care and consideration from an application developer to ensure high-performance on the target mobile platform.

3.1.1. Comparison between PC/console and mobile platforms

Compared to current PC and console platforms the resources and capabilities of mobile platforms are quite limited. Mobile processors run at low clock speeds (100 – 624MHz) have small cache sizes and may lack native floating-point support. The amount of memory available is limited and a large area will be reserved for the operating system and various drivers. The available bandwidth is limited by a narrow bus and low clock speed and in the majority of cases mobile platforms have a Unified Memory Architecture (UMA). This means that all devices are sharing the same single memory resource resulting in a higher number of page breaks and associated page break costs. Data storage is limited on mobile devices and data distribution is often limited by cost and carrier restrictions.

The PowerVR 3D accelerator offers a feature set not so different from the PC graphics cards from just a couple of years ago including advanced features such as multi-texturing and per-pixel DOT3 lighting. Polygon throughput and fill-rate might seem lower than past PC counterparts they are not at all unbalanced when taking into account the reduced display sizes: QVGA/VGA for a mobile device versus 1024x768/1280x1024 on PC.

3.1.2. Coding Considerations

Performance

CPU, Memory & Bandwidth

The key differences and thus most likely causes of problems between PC/console and a mobile platform is the massive difference in CPU capabilities (especially when doing floating point maths on a fixed point only platform), amount of memory available and bandwidth. There is only one single magical solution to dealing with this and that is: Stop and Think. These limitations hit at the base of any 3D/game engine you might be porting and retrofitting a solution will not be easy. Having a strategy and plan to deal with this from the start is essential since most likely you'll need to be looking at tweaking the internal data structures and concepts of the game engine. Without sufficient care you might end up with a game engine which does not run at all because it uses more memory than you have or runs at a crawl due to excessive amounts of floating point operations.

3D Core

Most PC and console game engines already do a fairly good job at driving the APIs optimally through batching, state management, etc. Some tweaking might be required on the side of the artwork taking into account the reduction in display size: reducing the size of textures, recompressing using PVRTC, reducing polygon counts, etc.

3.2. Porting a 3D application from 3D software mobile platform

Hardware rendering provides many advantages over software rendering, and there are several 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. The following points cover the major implications we've discovered of switching, and to point out the most important lessons involved; for a more detailed discussion see the document "Migration from software rendering to 3D accelerated graphics using OpenGL ES" available from the PowerVR Developer Relations website.

3.2.1. HW rendering is not simply faster SW

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.

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.

3.2.2. Coding Considerations

Use both processors

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

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.

4. Reference Material & Contact Details

PowerVR Public SDKs can be found on the PowerVR Developer Website:

http://www.powervrinsider.com

Additional OpenGL-ES Programming information can be found on the Khronos Website:

http://www.khronos.org/

Further Performance Recommendations can be found in the Khronos Developer University Library:

http://www.khronos.org/devu/library/

Developer Community Forums are available:

http://www.khronos.org/message_boards/
http://www.pocketmatrix.com/forums/viewforum.php?f=5

Additional information and Technical Support is available from PowerVR Technical Support who can be reached on the following email address:

devtech@powervr.com