PowerVR Performance Recommendations

The Golden Rules

Filename	:	PowerVR Performance Recommendations.The Golden Rules
Version	:	PowerVR SDK REL_3.5@3523383a External Issue
Issue Date	:	17 Apr 2015
Author	:	Imagination Technologies Limited

Contents

List of Figures

1. Introduction

1.1. Document Overview

The purpose of this document is to serve as a complete list of rules for developers to use when developing a graphics application. Developers should implement and observe as many of the mentioned techniques and principles within this document so as to produce well-behaved, high performance graphics applications.

1.2. Common Bottlenecks

The following rules are mainly intended to aid in resolving bottlenecks. A list of bottlenecks, arranged from most likely to least likely, is presented below:

- CPU Usage.
- Bandwidth Usage.
- CPU/Graphics Core Synchronization.
- Fragment Shader Instructions.
- Geometry Upload.
- Texture Upload.
- Vertex Shader Instructions.
- Geometry Complexity.

2. The Golden Rules

2.1. Understand Your Target Device

No two mobile devices are identical and no two graphics architectures are the same. Even when the Graphics Core architecture being targeted is thoroughly understood it is important to remember that no two System-on-Chips (SoCs) have exactly the same capability; some may have more powerful CPUs, others may have greater availability of bandwidth, etc.

Also, even between two otherwise identical devices (as SoCs share memory between components) it is possible for applications to be slowed down by other applications being run in the background, especially in regards to memory bandwidth.

The Tile Based Deferred Rendering (TBDR) system used in PowerVR hardware already helps to relieve some of these issues by ensuring memory access is kept to a minimum. However, it is still vitally important that developers take these factors into account. In order to sensibly benchmark and recognise what bottlenecks may exist, a developer must possess a thorough understanding of the architecture being targeted. To this end an architecture overview document is provided in the PowerVR Graphics SDK. Developers should seek to learn as much information about their target platforms and where they may differ as possible. The manufacturers' websites for devices are a good place to look for specifications and they may also provide other helpful developer community resources.

2.1.1. Querying the Capabilities of the Hardware

If the developer's application needs to exceed the default maximum capability values of the chosen graphics API, such as `GL_MAX_VARYING_VECTORS` in OpenGL ES 2.0, the capabilities of the device should always be checked through the graphics API. Exceeding the capabilities of the device will result in undefined behaviour.

2.2. The Principle of "Good Enough"

The principle of "Good Enough" refers to the notion that if the viewer cannot tell the difference between differently rendered images, use the cheaper option so as to not waste Graphics Core time. Real-time graphics can only achieve a finite level of image fidelity per frame whilst maintaining a reasonable frame rate. The more complex a frame the slower it can be rendered. This means that a compromise has to be made between image quality or "correctness" and speed of rendering. While making this compromise it is important to remember that rendering is only as valuable as the quality perceived by the viewer and will always be an approximation.

2.2.1. Examples

Alpha Blended Polygon Rendering Order

Many blend modes used when rendering transparent objects in a scene are submission order-dependent in their output. To get a correct and consistent output the fragments rendered using these modes should be drawn in depth order, back to front. Unfortunately, sorting these transparent layers per-fragment is difficult and prohibitively expensive. Developers solve this issue by using "good enough" compromises, such as:

- Sorting per polygon: Very expensive and problematic.
- Splitting objects into sub-meshes and sort: Still expensive and introduces some artefacts.
- Sorting by object: Cheap, but artefacts are easy to uncover.

Interpolated Vertex Values vs. Per Fragment Calculations

Many techniques, such as bump-mapping, may be more accurate with per-fragment calculation, but in practice can look entirely acceptable using the results of per-vertex calculations that are interpolated across the fragments in a polygon. This approach, due to there being fewer vertices than fragments in a typical scene can be more efficient for "good enough" results.

Lower Bit-Rate Textures

Textures that are compressed or down-sampled can look noticeably different to the developer, particularly in the preview window of a tool such as PowerVR's PVRTexTool GUI application. In a 3D scene this difference is often less obvious, to the extent that a user will not perceive any benefit from the higher bitrate, or from more expensive versions of the textures. The smaller textures are usually "good enough" for acceptable image quality while making a much more significant difference to bandwidth use and hence frame rate.

2.2.2. Profile, Profile, Profile!

Profiling tools are provided to developers so they can gain an understanding of what is happening in their application, and how it relates to the hardware the application is running on. They allow developers to determine where bottlenecks are occurring, and enable effort to be concentrated on areas that will improve rendering performance.

Ideally, optimizations should only be targeted at bottlenecks that have been identified through profiling. These optimizations should then be checked by re-profiling. This cyclical process ensures applications do not fall into the trap of cutting quality for no gain, for example, halving the number of vertices in a scene which is limited by the complexity of the fragment shader. To this end the PowerVR Graphics SDK includes a Graphics Core profiling tool called PVRTune. Further information on identifying bottlenecks with PVRTune can be found in the "PVRTune User Manual".

2.3. Promote Calculations 'up the Chain'

In general, fewer vertices exist in a scene than fragments appear on the screen. As such, processing time can be saved by performing a calculation in the vertex shader rather than in the fragment shader. This is promoting a calculation "up the chain". Figure 1 provides an illustration of the concept of "up the chain".

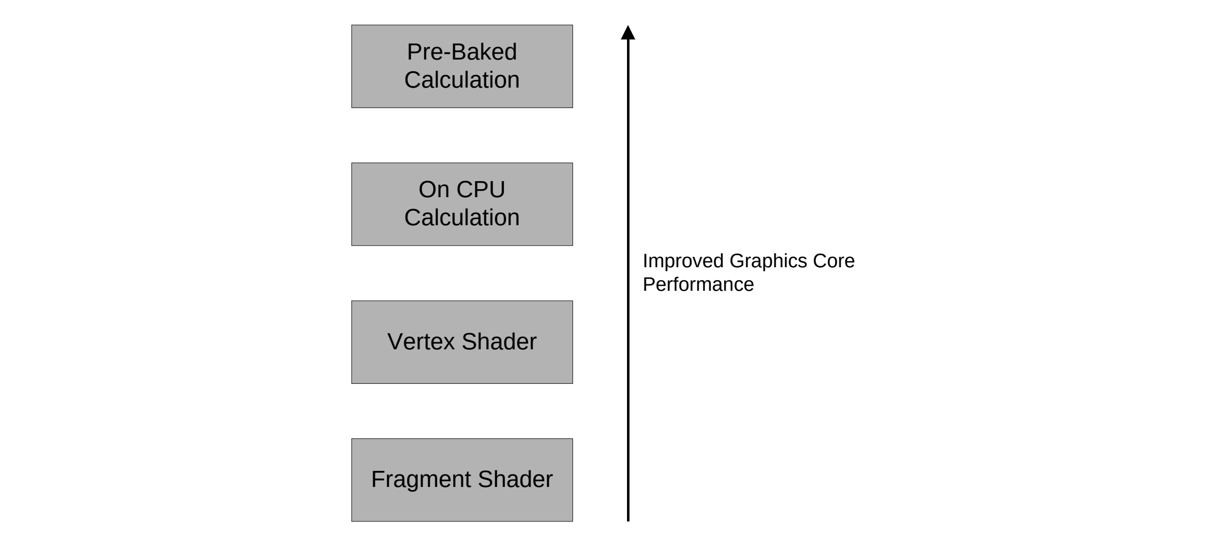

Figure 1. Generalisation of 'up the chain'

Even more Graphics Core performance can be gained by promoting a calculation off the Graphics Core altogether. For example, pre-building a matrix or pre-transforming an object into view space on the CPU. While the Graphics Core can perform these operations very rapidly, in many cases far more rapidly than the CPU, performing a calculation once on the CPU is much less intensive than performing the operation once per vertex, or once per fragment.

Pre-Baking

Pre-baking takes this concept one step further. It is likely that a fragment shader is already reading from several textures, so certain features can be pre-baked into these textures to save time, such as lighting.

Lookup Textures (Series5 and Series5XT only)

This can be taken one step further again using the concept of “lookup textures”. A texture is created to act as a lookup table for a particular mathematical function. Using this technique a mathematical function can be replaced by a texture read saving calculation time at the expense of bandwidth. It should be noted that a significant amount of work must be saved if this texture read would qualify as a ‘dependent’ texture read. Due to the higher ALU power in PowerVR Series6 onwards it is unlikely this will provide a benefit.

2.4. Avoid Accessing Render Targets

Accessing render targets from the CPU is very bad for Graphics Core performance as it breaks the parallelism between the CPU and the Graphics Core:

- Any access to the current render target will cause the driver to flush queued rendering commands and wait for rendering to finish.
- The CPU must wait for the flush to complete before it can access the buffer in question.
- The Graphics Core must wait for the CPU to finish its access before it can continue working in that buffer and may need to wait for further graphics instructions to be submitted from the CPU.

Figure 2 depicts an illustration of serialised render target access.

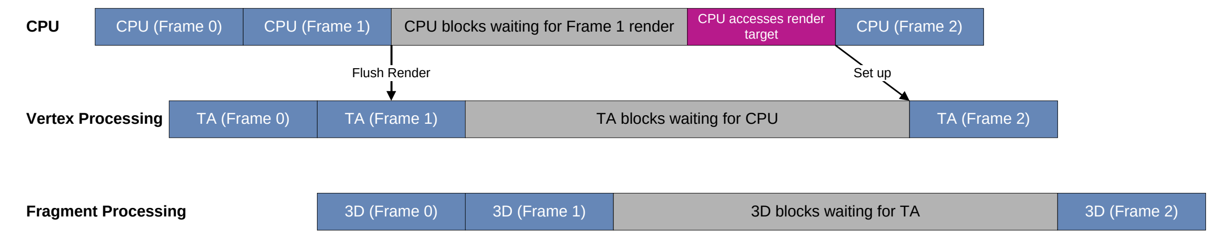

Figure 2. Serialised render target access

2.4.1. Reading Asynchronously

It is often useful or necessary to read back images from the Graphics Core to the CPU for further processing or storage. Whilst it is advisable to try avoiding these cases, sometimes these cases are unavoidable. In such instances, a number of techniques can be employed to read the image back asynchronously to avoid the cost of stalling both CPU and Graphics Core.

Circular Arrays

The most important technique when trying to avoid stalling is the use of a circular array that can be cycled through, so that no buffer is being accessed for a read before the write is complete. The basic version of this is to create a circular array of framebuffer objects, and cycle through them one by one, rendering and then reading from them in turn. The steps to do this are as follows:

1. Create a number of framebuffer objects (which will be referred to as N for this explanation), each identical for rendering.
2. For each frame, advance one place through the array of framebuffers.
3. Render the first N frames, one into each framebuffer object. Maintain the same rendering order throughout.
4. After each framebuffer object has been rendered to once, the next time you bind it you read from it first, and then render to it, giving the Graphics Core the maximum time to have finished rendering the previous frame.
5. Continue doing this until all frames are completely written.
6. Read the last N frames in the same order you have been up until this point to keep the app responsive.

This is a simple, single threaded method of improving performance whilst avoiding too much complexity in the application. The optimal value of N will vary depending on the application and device used, so should be profiled and optimised, but will usually be between 3 and 5. The more framebuffer objects that are created, the more memory will be consumed, but the application will run faster as well, to a point.

Pixel Buffer Objects

The main problem with allocating a circular array of framebuffers is that each of these framebuffers has an associated state, and each of these has its own depth buffer if necessary for optimal performance. To avoid allocating these unnecessary surfaces and the state setup associated with them, it is possible to use a circular array of Pixel Buffer Objects (PBOs) in OpenGL ES 3.x instead.

The basic premise is that the read can be queued on the Graphics Core, and it will fill the buffer with the appropriate data when the write has completed. This data can then be copied to the CPU at a later point when it is complete. Just using PBOs would not avoid the stall, as reading the data from the PBO early would cause the same pipeline flush as a normal `ReadPixels` would. So the strategy is to use the circular buffer as before with some slight modifications:

1. Create N PBOs, each with enough data storage to store the image you want to read.
2. Render each frame in turn.
3. Advance one place through the array of PBOs every frame.
4. Bind the current PBO, and call `glReadPixels` with an offset into it.
5. Continue doing this until all PBOs have been written once.
6. Every frame after the first N, when you bind the PBO and before calling `glReadPixels`, you should use `glMapBufferRange` to retrieve the image data stored within.
7. Continue rendering until the end of the application, and then read all the PBOs in the order that has been used up until this point.

Other than the use of PBOs, the principle is the same as with framebuffers, and again N will generally be between 3 and 5, ideally optimised for the target application or device.

Fence Synchronisation Objects

So far the strategies have involved tuning the application to manage framebuffer objects manually, and estimating when data will be ready for consumption without a stall. Instead, it is possible to use fence synchronisation objects (fence syncs) to tell exactly when the Graphics Core has finished writing to a buffer or framebuffer.

Fence syncs are provided in OpenGL ES 3.x directly with a call to `glFenceSync()`, or using the `EGL_KHR_fence_sync extension`, which provides `eglCreateSyncKHR()`. This can be used to produce equivalent fence sync if OpenGL ES is bound to EGL.

By inserting a fence sync immediately after the write operation (either the last draw before a `ReadPixels`, or after the `ReadPixels` if PBOs are used), when the fence sync is complete the application is free to read the values to the CPU without a stall.

The basic principle of fence syncs is to insert them after any Graphics Core command that the application wants to have completed before some other operation, which is similar to writing to a framebuffer. Fence syncs allow an application to manage its own dependency tracking rather than guessing what the driver is doing.

To make full use of this, an application needs to be prepared to grow its circular array to accommodate the information fence syncs provided, so that it completely avoids a stall. However, it often means that an application will end up using a smaller circular array as it is automatically tuned to each device that it will run on. The steps to do this with PBOs are as follows:

1. Create an initial number of PBOs, each with enough data storage to store the image.
2. Create a parallel circular array to store fence sync object handles.
3. Render each frame in turn.

4. Advance one place through the array of PBOs every frame.
5. Bind the current PBO, and call `glReadPixels` with an offset into it.
6. Create a fence sync object, and track it alongside the PBOs, advancing at the same rate.
7. Continue doing this until all PBOs have been written once.
8. For every subsequent frame, before calling `glReadPixels`, check if the associated fence sync has completed with a call to `glWaitSync()` (or `eglClientWaitSyncKHR()` with a timeout of `0`):

    - If the fence has completed, use `glMapBufferRange` to retrieve the data in the buffer and carry on as normal.
    - If it has not, create a new PBO, and bind and call `glReadPixels` on it. Then create a new fence sync object, and add them both to the circular array before the current PBO, so that they are current. The next frame should use the same fence and PBO as were initially checked in this frame.

9. Continue rendering until the end of the application, and then read all the PBOs in the order that has been used up until this point.

The only issue with this method is that there will be a stutter at the start of the application while the application is attempting to create enough PBOs to avoid stalling in future. It will take a few cycles for this to stop, but should not last more than a couple dozen frames. After that, the number of PBOs should be tuned to the number required by the application, and if anything causes it to need more at a later date, it can dynamically react.

Android: Multi-Threading, EGLImages and GraphicsBuffers

Using Android's GraphicsBuffer API, it is possible to avoid a lot of the overhead of the above techniques. It requires the use of multiple threads and will require time to get right as it involves tracking dependencies without the safety net of the driver underneath. The main advantage of doing it this way is that there is no need to create multiple buffers or textures to store data in an interim period, instead of grabbing the texture data directly.

The disadvantage is that GraphicsBuffer is an internal API used by Android, and access can only work if you link against pre-mangled names or rebuild Android. It is also not entirely portable, though it will work on PowerVR platforms. This method should thus be used with extreme care, and should only be used when memory consumption is a key issue. Significant testing is advised. However, it is important to remember that this is a very powerful technique when used correctly, often providing significant performance boosts. A number of components are required to make it work:

- `EGL_KHR_image_base`: Creation and handling of EGLImage objects.
- `GraphicsBuffer`: Access the memory store of a Graphics Core allocation directly.
- `EGL_ANDROID_image_native_buffer`: Create an EGLImage from a GraphicsBuffer.
- `GL_OES_egl_image`: Create a GL texture or renderbuffer that can be used as a framebuffer attachment.

The steps required to get this working are as follows:

Setup

1. Create a `GraphicsBuffer` object, either raw or from the `ANativeWindowBuffer` object in your application.
2. Query the `ANativeWindowBuffer` handle for the `GraphicsBuffer`, and pass that to `eglCreateImageKHR`, with a target of `EGL_NATIVE_BUFFER_ANDROID`.
3. Use `glEGLImageTargetTexture2DOES()` to create a texture handle from the `EGLImage`.
4. Create your framebuffer object, using this texture as the colour attachment.
5. Create a FIFO queue for fence sync handles.

Rendering Thread

6. Render each frame in turn to the framebuffer object.
7. Create a fence sync object, and add it to the queue.
8. Repeat until the final frame.

Second Thread

9. Call `glClientWaitSync` (or `eglClientWaitSync`) with a timeout of `GL_TIMEOUT_IGNORED` (or `EGL_FOREVER`).
10. Once a fence sync has completed, call `GraphicsBuffer::lock()` to prevent the data from being overwritten whilst being accessed.
11. Access the memory directly, and perform whatever operations are needed.
12. Call `GraphicsBuffer::unlock` to allow the Graphics Core to write into it again.
13. Remove the fence sync from the queue, and move to the next one.
14. Repeat until the final frame.

Note: As many GraphicsBuffers as needed can be created and used in this way, providing a method of using multiple render targets with this technique.

2.5. Avoid Using Alpha Test/Discard

The OpenGL ES 2.0 fragment shader operation `discard` can be used to stop fragment processing and prevent any buffer updates for the current fragment in a shader. Essentially, it provides the same functionality as the fixed function `alpha test` but in a programmable manner. It can seem like a convenient method to achieve the rendering of complex shapes without using geometry, but it is an expensive operation on all modern graphics hardware and thus is discouraged.

On modern graphics hardware, the use of alpha test requires that the fragment shader be run for a given fragment before visibility can accurately be determined. This affects performance on PowerVR hardware as the visibility information must be fed back from the fragment processing stage to the ISP before the ISP can continue to perform depth and stencil tests for other polygons in that position. This effectively removes some of the benefits of PowerVR's Hidden Surface Removal (and those of 'Early-Z' techniques on other architectures). For this reason `discard` and `alpha test` should be avoided whenever possible.

It should also be noted that if a shader contains the `discard` keyword then any object that shader is applied to will suffer the cost of alpha test, even if the keyword is inside a conditional block that the developer knows will not be hit for a draw call. The Graphics Core cannot know the result of the conditional without executing the fragment shader and so has to assume that the `discard` keyword may be hit. The solution to this is to move the use of `discard` into a separate shader.

Under fixed function APIs it is essential that alpha test be disabled for objects that do not require it. It is common for alpha test to be switched on at the beginning of a scene and left on for the entire scene. This is strongly discouraged is it may severely harm performance. The same visual effect as `discard` can often be achieved through the use of the correct Alpha Blend Mode, and setting the Alpha value to `0` where `discard` would be used. If `discard` or `alpha test` cannot be avoided then objects using these techniques should be submitted after all opaque geometry is submitted.

2.6. Avoid Unnecessary Framebuffer Reads/Writes

Tile-based architectures make use of on-chip memory, avoiding the bandwidth cost of accessing main memory. The benefit of this is lessened, however, by the fact that framebuffers that have been rendered to need to be available in subsequent renders in today's graphics APIs.

To make previous renders available when using a tile-based architecture, data must be copied to and from main memory at the start and end of each render. Most applications do not need all of this data available, and often will simply overwrite the data that was present, making the data copies a wasted effort. To get around these copies, many APIs have mechanisms which allow users to prevent this behaviour.

Modern graphics APIs have calls to clear frame buffers, which when used at the start of a render will tell the driver that it is not necessary to copy data from the previous render's output. This will prevent the system copying the data from main memory back to the chip for use in the next render.

The write to memory from on-chip buffers requires specific functionality not present in all APIs, but should be used when available. This is typically in the form of a function either called `Invalidate` or `Discard` (not to be confused with material discussed in Section 2.4).

2.6.1. OpenGL ES 2.0

To prevent reading from main memory for a new render, `glClear` should be called at the start of a new frame (after `eglSwapBuffers`) and when binding a new FrameBuffer (after `glBindFramebuffer`).

To prevent writing the frame out in the first place, the `EXT_discard_framebuffer` is provided on PowerVR platforms, which adds the function `glDiscardFramebufferEXT`. This function should be called at the end of a frame (before `eglSwapBuffers`) and before unbinding the current FrameBuffer (before `glBindFramebuffer`).

2.6.2. OpenGL ES 3.0

To prevent reading from main memory for a new render, `glClear` should be called at the start of a new frame (after `eglSwapBuffers`) and when binding a new FrameBuffer (after `glBindFramebuffer`).

To prevent writing the frame out in the first place, `glInvalidateFramebuffer` should be called at the end of a frame (before `eglSwapBuffers`) and before unbinding the current FrameBuffer (before `glBindFramebuffer`).

2.6.3. OpenGL (pre 4.3)

To prevent reading from main memory for a new render, `glClear` should be called at the start of a new frame (after `glFinish`) and when binding a new FrameBuffer (after `glBindFramebuffer`).

To prevent writing the frame out in the first place, `ARB_invalidate_subdata` may be provided, which adds the function `glInvalidateFramebufferARB`. This function should be called at the end of a frame (before `glFinish`) and before unbinding the current FrameBuffer (before `glBindFramebuffer`).

2.6.4. OpenGL 4.3+

To prevent reading from main memory for a new render, `glClear` should be called at the start of a new frame (after `glFinish/SwapBuffers`) and when binding a new FrameBuffer (after `glBindFramebuffer`).

To prevent writing the frame out in the first place, `glInvalidateFramebuffer` should be called at the end of a frame (before `glFinish/SwapBuffers`) and after unbinding the current FrameBuffer (before `glBindFramebuffer`).

2.6.5. DirectX 11.0

To prevent reading from main memory for a new render, `ID3D11DeviceContext::ClearRenderTargetView` and `ID3D11DeviceContext::ClearDepthStencilView` should be called at the start of a new frame (after `IDXGISwapChain::Present`) and when binding a new FrameBuffer (after `ID3D11DeviceContext::OMSetRenderTargets`). No method exists for preventing the write at the end of the frame in this, or previous versions of DirectX.

2.6.6. DirectX 11.1

To prevent reading from main memory for a new render, `ID3D11DeviceContext::ClearRenderTargetView` and `ID3D11DeviceContext::ClearDepthStencilView` should be called at the start of a new frame (after IDXGISwapChain::Present) and when binding a new FrameBuffer (after ID3D11DeviceContext::OMSetRenderTargets).

To prevent writing the frame out in the first place, ID3D11DeviceContext::DiscardView should be called at the end of a frame (before ID3D11DeviceContext::Present) and before unbinding the current FrameBuffer (before ID3D11DeviceContext::OMSetRenderTargets).

2.7. Take Advantage of HSR

Hidden Surface Removal (HSR) is the method by which PowerVR Graphics Cores remove unnecessary work by only processing fragments that will contribute to the final render. It is pixel perfect, and entirely submission order-independent. As such, Early-Z passes, and sorting of geometry by depth are not required. Instead, objects should be sorted by render state. This will help with batching (see Section 2.8) and will improve performance and power consumption.

2.7.1. Opaque, Alpha Test, Alpha Blend

To get the maximum benefit from HSR, opaque objects should be submitted first, then alpha test objects (where this technique cannot be avoided), and then alpha blended objects.

2.8. Batch, Batch, Batch!

2.8.1. Minimize State Changes

Wherever possible, code should be structured to avoid redundant state changes. Where this is hard to achieve, a copy of the current state can be kept in the application and a call only made if the old state and the new state are different. Ideally, the following rules should be followed:

- Set every OpenGL state at most once between draw calls.
- Set only those states that affect the next draw call.
- Do not set states that already have the desired value.

It should also be noted that, thanks to PowerVR's order independent, pixel-perfect, Hidden Surface Removal, objects do not need to be ordered by depth. This allows an application to sort by render state instead, which improves batching and minimizes state changes further. Applications should, as a priority, sort to avoid blending state or shader changes as these are the most expensive.

2.8.2. Group Meshes

If multiple meshes have static positions and orientations relative to one another, and could use the same render state, they should be combined into a single mesh. This will reduce the number of draw calls, and thus may increase performance.

2.8.3. Instancing (Series6 only)

Instancing is the practice of rendering multiple copies of the same mesh in a scene at once. By using this technique to draw repetitive geometry batching can further be improved, and thus performance.

2.8.4. Texture Atlases

A texture atlas is a single large texture that contains multiple sub-textures. With correctly calculated UVs, individual areas in the texture atlas can be used like a separate texture. This approach minimizes the number of times that textures must be rebound and hence reduces the number of draw calls an application requires, effectively batching textures.

There are two common issues that must be considered before using a texture atlas, however. The first of these is when using MIP-maps. During MIP-map generation it is possible that texels from neighbouring areas within the texture atlas can blend into each other. Some solutions to this are to leave a large border around each area within the texture atlas or to place areas with similar borders next to each other.

The second issue is in the use of texture wrap modes. When performing wrapping, OpenGL's texture coordinates only wrap over the entire texture. There is no facility to wrap over a portion of a texture. Solutions to this problem are to ensure enough redundancy in the atlas texture to avoid discrepancies, create a shader that will correct the UV mapping or to not use wrapping at all.

Information on the number of OpenGL calls an application is making, and the OpenGL render state etc., all of which relate to batching correctly, can be checked using the PVRTrace application, available as part of the PowerVR Graphics SDK.

2.9. Perform Rough Culling

Where it is feasible to do so, rough culling (Figure 3) should be performed on the CPU to avoid unnecessary geometry processing. Objects that are not going to contribute to a frame should not be submitted. A minimal approach to this is to cull objects that are behind the camera or otherwise outside the view frustum as these will be clipped away by the Graphics Core anyway. A more advanced approach would be to use a technique such as portal culling.

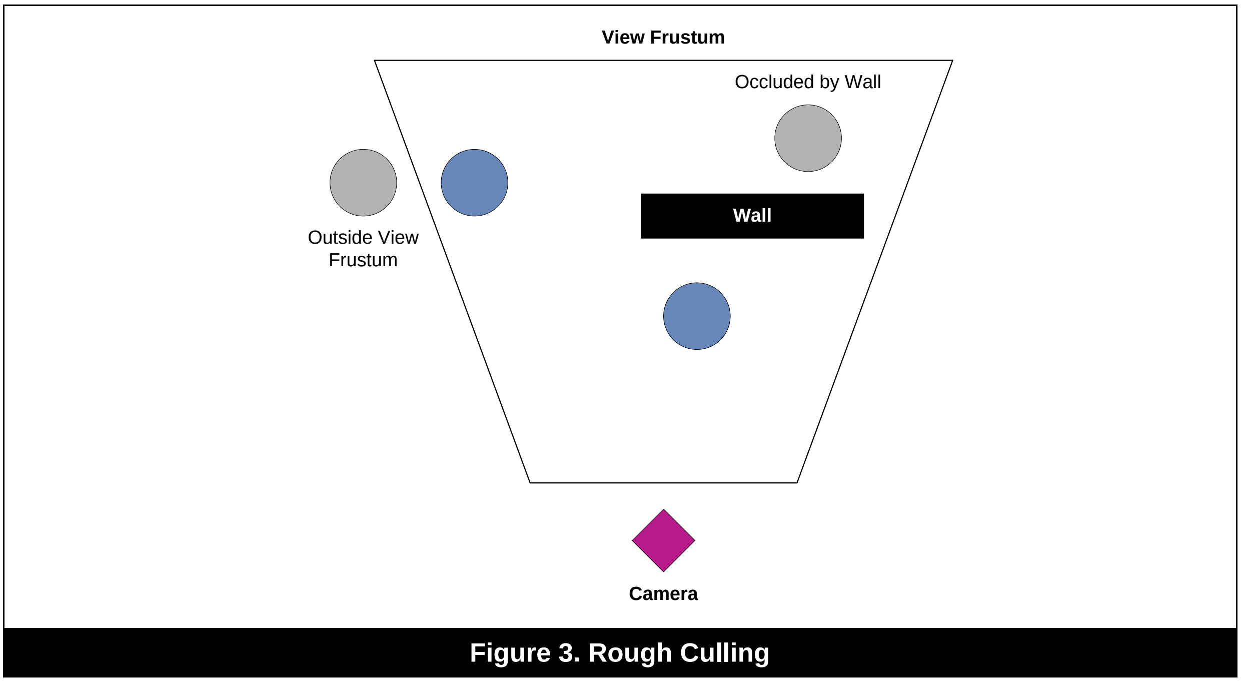

Figure 3. Rough Culling

2.10. Target a Sensible Frame Rate

A higher frame rate will improve the smoothness of any animation in an application and may improve the feel of user interaction. Forcing a lower frame rate than the maximum that the device is capable of will help with power consumption as less work needs to be done by the device to render frames over the same period of time.

As a rule, if animation is on screen and needs to be updated frequently, such as during game play, then aim to update the screen at a high, constant frame rate. If no animation is visible, such as when displaying an idle menu screen, then there is no reason to render the same image over and over. In this situation, consider lowering the frequency of screen updates until user interaction is detected. This will allow the device to expend less power over the same time period for the same result.

If an application is having trouble maintaining a constant frame rate, say it varies between 30 and 60fps, then it may be beneficial to restrict rendering to a constant 30fps, i.e., the lower end of the varying range, as this is likely to look just as smooth to the user without expending power drawing extra frames that do not enhance the quality of output.

Even if an application can run at a constant high frame rate, but this high frame rate does not enhance the user experience, then consider lowering the frame rate to reduce power consumption. Alternatively, this may be an opportunity to increase the quality of graphics in the application by increasing model complexity, texture detail, or shader sophistication.

2.11. Favour Stencil Operations

Stencil operations are carried out on-chip, per tile, and so are very cheap on PowerVR hardware. In almost every case a technique that could be performed using functions such as scissoring can also be performed using the stencil buffer. Preferring the use of the stencil buffer in these cases will almost always improve performance, but the following points should be considered first:

- **Is the Stencil Operation Required:** It is inefficient to use unnecessary calls that will not affect the final output. Thus, if an object to be drawn would fit entirely inside a stencil region then the stencil operation should not be used.
- **Where is the Performance Bottleneck:** It is essential to verify that the code that needs to be changed is actually affecting the rendering speed of the application. If there are very few stencil/scissor operations in an application then it is likely that more value will be gained from optimizing elsewhere.
- **Re-Implementing Using Stencil Operations:** As an example, a common use case would be the use of rectangles to restrict the area of the screen to be drawn to. This technique, normally performed with scissoring, can easily be solved using stencil operations and this may provide a substantial boost to rendering performance.

The procedure for implementing this technique is as follows:

1. Clear the stencil buffer.
2. Render rectangles to the stencil buffer only, each rectangle representing an area that is to be drawn to. Each rectangle should be given a unique value within the stencil buffer if overlapping is required. If no overlapping is required, or content will not spill over the edge of a stencil, then the same value may be used. It should be noted that submission order will affect overlap behaviour.
3. Render content using the stencil test value corresponding to the desired rectangle so that fragment visibility is determined by what is stored in the stencil buffer.

If more unique values are still required than the stencil buffer can contain, then the render should be resolved up to the stage where the number of available stencil values has run out. The above sequence should then be performed again for the remaining of the scene.

Furthermore, one of the bonuses of using stencil operations for this technique is that, unlike scissor methods, the stencils used can be of an arbitrary shape and size and are not just restricted to rectangles.

3. Contact Details

For further support, visit our forum:
http://forum.imgtec.com

Or file a ticket in our support system:
https://pvrsupport.imgtec.com

To learn more about our PowerVR Graphics SDK and Insider programme, please visit:
http://www.powervrinsider.com

For general enquiries, please visit our website:
http://imgtec.com/corporate/contactus.asp