3

VI

iPhone 3GS Graphics Development and Optimization Strategies

Andrew Senior

The iPhone was first released in June 2007 and has since had two major revisions: the 3G and the 3GS. In early 2008, Apple released an official iPhone SDK which enabled developers to create native applications for the device. A physical device is not required for most of development as a software implemented iPhone Simulator is provided with the SDK. Only members of the paid developer program are able to test their creations on a physical device and subsequently submit them to the App Store for distribution.

All revisions of the iPhone have three-dimensional hardware acceleration enabled by POWERVR cores. The original iPhone, iPhone 3G and first two revisions of the iPod Touch are capable of running OpenGL ES 1.1. The newer and more powerful iPhone 3GS adds OpenGL ES 2.0 capability to this (as do the 32- and 64-GB versions of the third-generation iPod Touch). The iPhone 3GS also boasts a faster CPU and double the amount of memory seen in previous models. This makes it a perfect candidate for a major gaming development platform.

This document is intended to provide developers with the essential help required to start creating graphical demos for the iPhone 3GS using OpenGL ES 2.0. At the time of writing the current version of the official iPhone SDK is 3.0. Version 3.0 was released along side the iPhone 3GS and is the first version to support OpenGL ES 2.0.

3.1 Software Development Kits

Apple provides a free SDK for the iPhone available from the Apple Developer Connection (http://developer.apple.com/iphone/). The iPhone SDK provides the developer with all the tools, libraries, and documentation needed to create any native application for the iPhone or iPod Touch. Many different types of application can be created for the iPhone, utilising one or more of the several input and output features of the device. The SDK provides a wizard utility to create an OpenGL ES application, currently only an OpenGL ES 1.1 implementation. With the recent release of the iPhone 3GS, it is now possible to benefit from the programmable graphics pipeline of its POWERVR SGX core by utilizing its OpenGL ES 2.0 support.

It is essential for any iPhone development that the iPhone SDK is installed on an Intel based Mac running Mac OS X Leopard version 10.5.7 or later.

3.1.1 iPhone SDK

The first step towards starting iPhone development is to visit the iPhone Developer Connection and sign up for a free account. Once signed in you can download the iPhone SDK. Included within the rather large download is Xcode, Apple's IDE for creating, debugging and optimising iPhone applications. The installation process of the iPhone SDK will perform a full installation of Xcode; there is no need to install it independently.

The iPhone SDK is written in Objective-C, an object-oriented extension to the C language. This can be quite frustrating to first time iPhone developers wishing to use knowledge of C/C++ to write an OpenGL ES application. It is a requirement that the developer at least understands how the Objective-C language works; an Objective-C programming guide can be obtained from the Apple Developer Connection website [Apple 09] for reference. As Objective-C is an extension of the C language it is perfectly legal to include C code in your application. An Objective-C program can also include C++ code by signifying to the compiler that you wish to use Objective-C++. In a regular Objective-C application the header and source code files have the extension .h and .m. To enable Objective-C++ it is just a matter of renaming the source file from .m to .mm, this is the only change Xcode requires in order to compile Objective-C++.

The installation process of the SDK will extract all the libraries, tools and other files to /Developer and Xcode can be found in the Applications sub-directory (/Developer/Applications). It is probably a good idea to drag Xcode onto your dock as during development you will be using it quite a lot.

3.1.2 POWERVR SDK

The POWERVR SDK for iPhone is now available free to developers from the POWERVR Insider page at http://www.powervrinsider.com. Imagination Technologies provides a comprehensive SDK offering tutorials, source code, utility applications, documentation, and a valuable tools library. Available is an OpenGL

ES 1.1 SDK for iPhone and iPod Touch, and an OpenGL ES 2.0 SDK for iPhone 3GS.

As mentioned in the previous section, newcomers to iPhone development may be put off by the Objective-C language. The POWERVR SDK encapsulates all of the system code and provides the user with a C++ interface via PVRShell. PVRShell is a C++ class used to make programming for POWERVR platforms easier and more portable. PVRShell takes care of all API and OS initialization and handles viewport creation and clearing. The application interface inherits from PVRShell and comprises five derived functions: `InitApplication`, `InitView`, `ReleaseView`, `QuitApplication`, and `RenderScene`. These functions will be invoked by PVRShell; `InitApplication` will be called before the graphics context is created and just before exiting the program `QuitApplication` will be called. `InitView` will be called upon creation of the rendering context and `ReleaseView` will be called when finishing the application, before `QuitApplication`. The previous four functions will only be called once per application execution. The main rendering loop function of the program is `RenderScene`. This function must return false when the user wants to terminate the application. PVRShell will call this function every frame and will manage the relevant OS events. There are other PVRShell functions available to the user to get information, such as input back from the iPhone: these include `PVRShellGet()` and `PVRShellSet()`. `PVRShellSet()` is recommended to be used in `InitApplication()` so the user preferences are applied at the API initialization.

It is recommended to use the PVRShell interface to promote cross platform compatibility for all POWERVR SGX hardware on Symbian, Linux, Android and WinCE as well as Mac OS. Using the PVRShell interface it is also possible to develop using the WindowsPC or LinuxPC emulation SDKs, also found at the POWERVR Insider web page. All non-system specific code will transfer between platforms seamlessly.

The POWERVR SDK contains tools and utilities to export and load compressed textures and animated three-dimensional models. Located within the Training Course folder of the SDK are many small demos each outlining a specific task or problem. The utilities bundled with the SDK will be described in the next section.

3.1.3 Tools and Utilities

Apple. Xcode is Apple's official development suite for OS X. Xcode is free and the latest developer version is included with every iPhone SDK release. Xcode includes all the tools needed to create, debug, and optimize your iPhone applications. The Xcode toolset includes the Xcode IDE, Apple's integrated development environment comprising of a text editor, compiler, and debugger. The iPhone simulator is part of Xcode, which enables testing directly on the development machine under a software implemented emulator. Generally the simulator will produce the exact same output as the physical device. To judge performance you should never use the simulator, always test on a device. Performance analysis can be run using Instruments, a profiling tool to help dynamically track processes and examine their behavior. Instruments can help you optimize your application detailing CPU and memory usage and even OpenGL ES graphics performance. Instruments can be used on the iPhone and the simulator, some restrictions apply to both. Using Instruments on the simulator you can record your input and replay it several times emulating touch input exactly as you recorded it. This is handy for testing input without all the tedious clicking and dragging. It is only possible to monitor OpenGL ES performance while running an OpenGL ES application on an actual device.

POWERVR. Provided with the POWERVR SDK is a suite of utilities including PVRTexTool, PVRShaman and PVRUniSCoEditor. These utilities provide content (textures, three-dimensional models and shaders) that can quickly and easily be integrated into PVRTools for rapid development. PVRTools is a three-dimensional graphics utility library written in C++. It includes functions to load compressed textures, three-dimensional animated models, and POWERVR PFX shader effect files.

PVRTexTool has both a GUI and a command line utility. PVRTexTool can compress any standard bitmap file (e.g., BMP, JPG, PNG, TGA, etc) into a PVRTC texture, recognized by the iPhone as a compressed image format. PVRTC has many benefits as described in Section 3.2.2. PVRTexTool can generate mipmaps for POT (power of two) textures and can compress them into either four or two bits-per-pixel PVRTCs. PVRTexTool supports normal map and sky-box generation as well as scale, rotate and flip transforms.

PVRShaman is an integrated shader development environment allowing rapid prototyping of new vertex and fragment shader programs. PVRShaman brings together geometry exported using PVRGeoPOD (or converted using Collada2POD), textures compressed using PVRTexTool, and on-the-fly editing of shader programs with editing functionality on the same level as the PVRUniSCo Editor. Shader effects can be saved as POWERVR FX files allowing easy integration with your base code.

PVRUniSCo Editor is the graphical front-end for the PVRUniSCo shader compiler. It allows easy creation and editing of OpenGL ES 2.0 shading language vertex and fragment shader programs, in addition to POWERVR FX (PFX) files.

3.2 General Optimization Strategies

This section will outline the most essential optimization techniques to speed up your OpenGL ES 2.0 applications on the iPhone 3GS. Although the shader optimization section is OpenGL ES 2.0-specific, the rest should apply to ES 1.1 developments as well.

By sticking to these golden rules you can expect your graphics applications to perform much better. Each rule will be explained in greater detail further in this chapter:

- *Do not access the framebuffer from the CPU.* Any access to the framebuffer will cause the driver to flush queued rendering commands and wait for rendering to finish, removing all parallelism between the CPU and the different modules in the graphics core.

- *Use vertex buffer objects and indexed geometry.* Vertex and index buffers will benefit from driver and hardware optimizations for fast memory transfers.

- *Batch your primitives to keep the number of draw calls low.* Try to minimize the number of calls used to render the scene, as these can be expensive. Using branching in your shaders may help to have better batching.

- *Perform rough object culling on the CPU.* Your application has more knowledge about the scene than the GPU. Whenever you have the opportunity to quickly detect that objects are invisible, do not render them!

- *Use texture compression and mipmapping.* Texture compression and mipmapping reduce memory page breaks, and will make better use of the texture cache.

- *Perform calculations per vertex instead of per fragment whenever possible.* The number of vertices processed is usually much lower than the total number of fragments, so operations per vertex are considerably cheaper than per fragment.

- *Avoid discard in the fragment shader.* POWERVR and other architectures offer performance advantages that are negated when discard is used.

3.2.1 Managing Vertex Data Efficiently

Vertex buffer objects. Geometry should be stored in a *vertex buffer object* (VBO) where possible. A VBO allows vertex array data to be stored in an optimal memory layout that drastically reduces data transfer to the GPU. You will see a noticeable increase in performance using vertex buffer objects on the iPhone 3GS.

This increase might not be as noticeable on the previous MBX enabled iPhone or iPod Touch devices but it will not degrade performance either. It is encouraged not to create a VBO for every mesh but to group meshes that are always rendered together in order to minimize buffer rebinding.

Interleaved attributes. There are several ways of storing data in memory. The two most common are interleaved and sequential arrays. Interleaved data stores all data for one vertex followed by all data for the next vertex. Storing the data in sequential arrays separates all vertex attributes into their own arrays, one array for position, one for normals, and so on. Interleaved attributes can be considered an array-of-structs while sequential arrays represent a struct-of-arrays.

Generally interleaved data provides better performance because all data for a vertex can be gathered in one sequential read which greatly improves cache efficiency. If a vertex attribute is shared across multiple meshes it can be faster to create a sequential array of its own rather than duplicating the data. The same is true if there is one attribute that needs frequent updating while other attributes remain unchanged.

Cull invisible objects. Although POWERVR SGX is very efficient in removing invisible objects, not submitting them at all is still faster. Your application has more knowledge of the scene contents and positions of objects than the GPU and OpenGL ES driver, and it can use that information to quickly cull objects based on occlusion or view direction. Especially when you're using complex vertex shaders it is important that you keep the amount of vertices submitted reasonably low.

To perform this culling it is important that your application uses efficient spatial data structures. Bounding volumes can help to quickly decided whether an object is completely outside the viewing frustum. If your application uses a static camera, perform view frustum culling offline.

Submitting geometry and rendering order. The order in which objects are submitted for rendering can have a huge impact on the number of state changes required and therefore on performance. This section outlines the fundamental rules to attain the desired result while achieving the best possible performance. As blended objects rely on the current value of the framebuffer, they would need to be rendered back-to-front.

Opaque objects are rendered without framebuffer blending or discarding pixels in the fragment shader. Always submit all opaque objects first, before transparent ones. To achieve the best possible results follow this order of object processing:

- Separate your objects into three groups: opaque, using discard, using blending.

- Render opaque objects sorted by render state.

```
#ifdef OPTIMIZED
for(int i = 0; i $<$ numParticles; i++)
{
    addParticleToInterleavedArray(particle[i]);
}
drawParticles();
#else
for(int i = 0; i $<$ numParticles; i++)
{
drawParticle(i);
}
#endif
```

Listing 3.1. Optimized batch rendering.

- Render objects using discard sorted by render state (avoid discard where possible).
- Render blended objects typically sorted back-to-front.

Sorting opaque objects by render state. Due to the advanced hidden surface removal (HSR) mechanism of POWERVR SGX opaque objects do not require to be sorted by depth. It would be considered a waste of CPU cycles sorting opaque objects front-to-back; instead objects should be sorted by render state. An example of this would be to render all objects using the same shader together. This will reduce the number of texture and uniform updates, as well as all other state changes.

Batching. Dynamically generated geometry such as a particle system should be rendered together in one *batch*. Batching reduces the number of draw calls per frame increasing performance. Opaque and transparent objects should be split up into separate batches. For best performance on POWERVR SGX you should always draw all geometry as a single indexed triangle list (see Listing 3.1).

3.2.2 Textures

A common misconception is that bigger textures always look better. Using a 1024×1024 texture for an object that never covers more than a small part of the screen just wastes storage space. Choose your textures' sizes based on the knowledge of how they will be used. Ideally you would choose the texture size so there is about one texel mapped to every pixel the object covers when it is viewed from the closest distance allowed.

NPOT textures. Non-power-of-two (NPOT) textures are supported by POWERVR SGX to the extent required by the OpenGL ES 2.0 core specification. This means

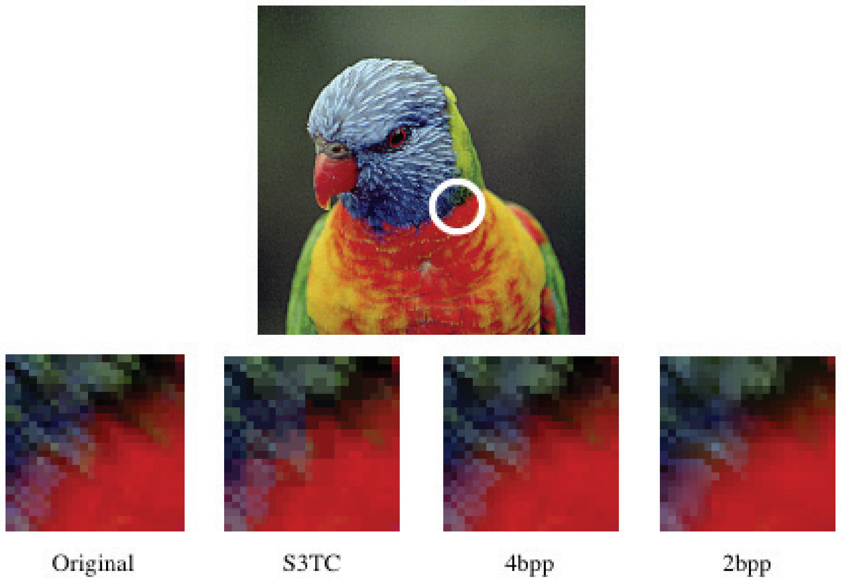

Figure 3.1. PVR texture compression comparison.

that mipmapping is not supported with NPOT textures, and the wrap mode must be set to CLAMP_TO_EDGE.

The main use of NPOT textures is for screen-sized render targets necessary for post-processing effects. Due to the mipmapping restriction it is not recommended to use NPOT textures for normal texture mapping of meshes due to memory access and cache inefficiency.

Texture formats and texture compression (PVRTC). POWERVR supports a proprietary texture compression format called PVRTC. It boasts very high quality for competitive compression ratios. As with S3TC, this compression is block based, but benefits from a higher image quality than S3TC as data from adjacent blocks are also used in the reconstruction of original texture data (See Figure 3.1). PVRTC supports opaque and translucent textures in both four bits-per-pixel and two bits-per-pixel modes. This reduced memory footprint is advantageous for embedded systems where memory is scarce, and also considerably minimizes the memory bandwidth requirements (while improving cache effectiveness). For more information about PVRTC please refer to [Fenney 03].

Mipmaps. Mipmaps are smaller, pre-filtered variants of a texture image, representing different levels of detail (LOD) of a texture. By using a minification filter mode that uses mipmaps, the hardware can be set up to automatically calculate which LOD comes closest to mapping one texel of a mipmap to one pixel in the render target, and use the according mipmap for texturing.

Using mipmaps has two important advantages. It increases performance by massively improving texture cache efficiency, especially in cases of strong minification. At the same time using pre-filtered textures improves image quality by countering aliasing that would be caused by severe under-filtering of the texture content. This aliasing usually shows itself as a strong shimmering noise on minified textures.

Mipmaps can be created offline with a utility such as PVRTexTool which can be found in the POWERVR SDK. Alternatively you can save the file storage space for mipmaps and generate them at runtime in your application. The OpenGL ES function `glGenerateMipmap` will perform this task for you, but it will not work with PVRTC textures. Those should always be offline generated. This function is also useful when you want to update mipmaps for render target textures.

In combination with certain texture content, especially with high contrast, the lack of filtering between mipmap levels can lead to visible seams at mipmap transitions. This is called mipmap banding. Trilinear filtering (`GL_LINEAR_MIPMAP_LINEAR`) can effectively eliminate these seams and thus achieve higher image quality. However, this quality comes at a cost, as whenever two mipmaps levels need to be blended together, the texture unit needs an additional cycle to fetch and filter the required data. In the worst case, trilinear achieves half the texturing performance compared to bilinear filtering. If you are using complex math-heavy shaders however, the texture unit may have cycles to spare and you can get trilinear filtering for a low bandwidth cost.

Rendering to a texture. The iPhone supports framebuffer objects (FBO) with textures as attachments; pbuffer surfaces are not supported on the iPhone. Creating an OpenGL ES view requires the generation of an FBO. The framebuffer generated should be treated as the back buffer for that view. All of the rendering on the iPhone is to an offscreen framebuffer object, this enables the OS to use fancy hardware driven transition and composition effects with high performance.

Texture atlases. By using texture atlases you can reduce the number of texture binds. Texture atlases consist of multiple texture images, usually the same size, arranged into a singular texture. If you can, use them to group texture images which are used interchangeably with the same shader program.

One issue with texture atlases you need to be aware of is that texture filtering, especially with mipmaps, may cause texels from neighbouring images to blend into each other. To avoid or reduce artifacts from this you need to either leave large enough borders around each image or place images with sufficiently similar edges next to each other. Using texture atlases also clashes with using texture repeat modes.

3.2.3 Shaders

Discard. The GLSL ES fragment shader operation `discard` can be used to stop fragment processing and prevent any buffer updates for this fragment. It provides the same functionality as the fixed function alpha test in a programmable fashion.

On mobile graphics architectures, discard is an expensive operation because it requires a fragment shader pass to accurately determine visibility of fragments. This reduces or removes any possible benefits from early depth rejection mechanisms.

The use of `discard` in shaders should be avoided wherever possible. Often the same visual effect can be achieved using the right blend mode and forcing the fragments alpha value to 0. If the use of `discard` is not avoidable, make sure the object that is using the shader is submitted after all opaque objects.

Placing a conditional branch around discard will not remove any performance issues explained earlier in this section. If discard is not needed, it should be removed from the shader entirely. A separate `discard` shader should be created if discard cannot be avoided.

Attribute data types. There are six different types of vertex attributes available in OpenGL ES2.0: `BYTE`, `UNSIGNED_BYTE`, `SHORT`, `UNSIGNED_SHORT`, `FLOAT` and `FIXED`.

Vertex shaders always expect attributes as type float, which means all types other than `FLOAT` require a conversion. The `FIXED` attribute type should be ignored when developing on the iPhone due to this. Moreover the iPhone has a dedicated FPU so there is unlikely to be a benefit from using fixed point arithmetic in general.

Attribute read times can be increased by ensuring that all attribute data is aligned to four-byte boundaries. For attributes that are less than 32 bits wide, one may have to add padding bytes to promote better attribute read times. It may be possible to pack different attribute vectors with fewer than four components together to minimize the space wasted.

Move calculations to the vertex shader. Try to move as much calculation from the fragment shader into the vertex shader as possible. There are generally a lot fewer vertices than there are fragments (pixels); by moving these calculations into the vertex shader you can drastically reduce shader cycles.

Often per-vertex equivalents can look just as good as costly per-pixel calculations. An example of this is lighting. Per-pixel lighting can look very nice, but for a real-time simulation, per-vertex is usually more than adequate.

Unrolling loops. If your application is still running slowly after you have tried the other strategies within this section, it might help to unroll any small loops in your shaders. It can help the compiler and possibly speed up the shader execution time.

3.3 Conclusion

This article addresses the main optimization strategies while developing for the iPhone and other POWERVR architectures. By following these optimization recommendations, it is possible to maximize performance of your OpenGL ES graphical demos and games.

For more support with the POWERVR SDK or PVRTools & Utilities visit the POWERVR Insider Forums at http://www.powervrinsider.com/forum/. The Khronos Group provides the OpenGL ES specification and reference manuals as well as hosting a developer support forum (http://www.khronos.org). Any questions relating to the Apple SDK, Xcode, or iPhone can be asked at the Apple Developer Connection forums at https://devforums.apple.com (paid membership required).

Bibliography

[POWERVR 09] Imagination Technologies. “POWERVR SGX OpenGL ES 2.0 Application Development Recommendations.” 2009

[Apple 09] Apple Inc. “The Objective-C 2.0 Programming Guide.” 2009. Available at http://developer.apple.com/iphone/library/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf

[Fenney 03] Simon Fenney. “Texture Compression using Low-Frequency Signal Modulation.” *Graphics Hardware*. White paper, 2003. Available at http://www.imgtec.com/whitepapers/PVRTextureCompression.pdf.