Contents

Exploring Mobile vs. Desktop OpenGL Performance

Jon McCaffrey

1.1 Introduction

The stunning rise of mobile platforms has created a nascent market for new 3D applications and games where, excitingly, OpenGL ES is the *lingua franca* for graphics. However, mobile platforms and GPUs have performance profiles and characteristics that may be unfamiliar to desktop developers. Developers hoping to make the transition from desktop to mobile need to be aware of these to create the best experience possible for the given hardware resources.

1.2 Important Differences and Constraints

1.2.1 Differences in Scale

Modern mobile devices are capable and remarkable devices. However, they face much greater limitations than desktop systems in terms of cost, die size, power consumption, and heat dissipation.

Power consumption is a major concern for mobile platforms that is much less pressing on desktop. Mobile devices must run off batteries small enough to fit in the body of the device, and a short battery life is frustrating and inconvenient to the user. Mobile hardware is built to use less power than desktop hardware via lower clock frequencies, narrower busses, smaller chips, smaller data formats, and by limiting redundant and speculative work. Display and network take a great deal of power, but OpenGL applications contribute to power consumption, especially through compute and through off-chip memory accesses.

Power consumption is doubly-impactful on mobile devices, since power consumed by the processor, GPU, and memory is largely dissipated as heat. Unlike desktop systems with active air cooling, good air circulation, and large heat sinks to radiate heat, mobile systems are usually passively cooled and have contrained bodies with little room for large sinks or radiating fins.

Excess heat generation is not only potentially damaging to components, it's also noticeable and irritating to users of hand-held products.

Die size and cost are also greatly different between mobile and desktop. High-end desktop GPUs are some of the largest mainstream chips made, with over 3 billion transistors on recent models [Walton 10]. Both the large area and the effect of area on yield mean increased cost. A discrete GPU also means a separate package and mounting, and the expected cost increase. In mobile however, the GPU is usually one component on an integrated *System-on-a-Chip (SoC)* designed for mobile and embedded applications, which means it is a fraction of the cost and area of a desktop GPU.

1.2.2 Differences in Rendering Architecture

Mobile and desktop GPUs don't differ only in scale. mobile GPUs such as the Imagination Tech SGX543MP2 used in the Apple iPhone 4S/iPad 2 and the ARM Mali-400 used in the Samsung Galaxy S2 use a *tiled* rendering architecture [Klug and Shimpi 11b]. In contrast, desktop GPUs from NVIDIA and ATI and mobile GPUs like the GeForce ULV GPU used in the Samsung Galaxy Tab 10.1 use *Immediate Mode Rendering (IMR)*.

In IMRs, vertices are transformed once and primitives are rasterized essentially in order. If a fragment passes depth-testing (assuming the platform has early-z), it will be shaded and will write to the framebuffer. However, a later fragment may over-write this pixel, nullifying the earlier work done and writing the framebuffer again. This behavior is known as overdraw. Even without overdraw, the depth buffer still must be read for later fragments generated at a pixel location in order to reject them.

Tilers instead divide the framebuffer into tiles of pixels. All draw commands are buffered, or deferred. At the end of the frame, for each tile, all geometry for the scene is re-rendered and rasterized into a framebuffer cache with that tile scissored out. Once all pixels have been resolved, the entire tile is written out to memory. This saves redundant framebuffer writes and allows for fast depth-buffer access, since depth-testing and depth writes can be performed with the local framebuffer cache. The end goal is to limit the memory bandwidth consumed by color and depth buffer access.

Tiling doesn't come for free. The scene geometry must be re-rendered for each tile, so to maintain a balanced pipeline additional vertex processing power is needed. If pre-transform vertex attribute caches aren't large enough, bandwidth will also be spent re-reading vertex data for each tile. The logic for tiling, additional vertex processing, and a fast framebuffer cache also cost transistors from raw fragment shading horsepower.

There is an additional group of tilers which use *Tile-Based Deferred Rendering*, including the SGX family. The idea is to rasterize all primi-

tives in a tile before performing any fragment shading. This allows *Hidden Surface Removal (HSR)* and depth-testing to be performed in a fast framebuffer cache before any fragment shading work is done. Assuming opaque geometry, each pixel is then shaded and writted to the framebuffer exactly once.

These architectures have different strengths and weaknesses we will consider, but it's important to note that the mobile GPU landscape is not homogenous. Optimizations may affect the different architectures very differently, so it is important to test on multiple devices for cross-plattform releases.

1.2.3 Differences in Memory Architecture

On desktop systems, middle-to-high-end GPUs are discrete devices which communicate with the rest of the system via a peripheral bus like *PCIe*. For good performance, this means that GPUs must include their own dedicated memory, since accessing system memory through a peripheral bus for all memory accesses would be too slow in terms of bandwidth and latency. While this increases cost, it is an optimization opportunity since this memory and its configuration, controller, caching, and geometry can be optimized for graphics workloads.

For example, the NVIDIA *Fermi* architecture uses GDDR5 memory that is heavily partitioned [Walton 10] to allow for very wide memory interface. There is also no competition for this bandwidth, since except for uploads from the rest of the system and scan-out for output devices, the GPU is the only user of this memory.

In mobile devices, on the other hand, the GPU is usually integrated into the same SoC as the CPU and other components, and to save cost, power, size, and complexity, shares the same RAM and memory interface. This is known as an *Unified Memory Architecture (UMA)*. A common memory type might be the low-power LPDDR2, which has a 32-bit wide interface [Klug and Shimpi 11a]. Not only is this memory general-purpose, the GPU now shares bandwidth with other parts of the system like the CPU, network, camera, multimedia, and display, leaving less dedicated bandwidth available for rendering and composition.

There are some performance advantages to a unified memory architecture, besides the savings in cost and complexity. With discrete GPU's the peripheral bus could become a bottleneck for transfers, especially for non-PCIe buses with asymmetric speeds [Elhasson 05]. With a UMA, OpenGL client and server data are in fact stored in the same RAM. Even when it is not possible to directly access server data with `glMapBufferOES`, there are fewer performance cliffs lurking in transfers between OpenGL client and server data, and using client-side buffers for dynamic vertex data or

indices may not have as great a performance penalty. Memory and command latencies from the GPU to the CPU are also likely to be less.

One current limitation is that OpenGL ES does not yet have an extension for *Pixel Buffer Objects (PBO)*, meaning that pixel and texture data must be transferred synchronously. This makes the comparatively cheap bandwidth between client and server data less useful, and also makes streaming assets during run-time more difficult.

1.3 Reducing Memory Bandwidth

Memory bandwidth pressure is one of the major performance pressures on mobile devices, especially on advanced games and other applications which must also perform heavy amounts of client-side work during the frame, or in multimedia applications which have additional bandwidth clients besides the GPU.

Besides limiting performance, memory accesses external to the GPU consume a great deal of power, sometimes moreso than the computation itself [Antochi 04].

Device	CPU Write Bandwidth	GPU Write Bandwidth
Motorola Droid X	1.4GBps	6.8GBps
LG Thunderbolt	0.866GBps	0.518GBps
Dell Inspiron 520	4.8GBps	3.8GBps
Desktop System	14.2GBps	25.7GBps

Table 1.1. Write bandwidth for CPU and GPU on different devices. CPU write bandwidth estimated by memset. GPU bandwidth estimated by glClear+glFinish. Desktop system has an Intel Core 2 Quad and NVIDIA 8800 GTS. The desktop system has significantly more bandwidth available to the GPU than to the CPU, and CPU and GPU memory accesses do not interfere with each other.

1.3.1 Relative Display Sizes

Despite the tight power and cost constraints for mobile devices, the display sizes of modern mobile devices are a considerable fraction of the size of desktop displays.

With the limited fragment shading throughput and memory bandwidth of mobile devices, these comparatively large display sizes mean that fragment shading and full-screen (or large-quad) operations can easily become a bottleneck, since these requirements scale proportionally with the number

Device	Resolution	% of 1280x1024 (20 in)	% of 1920x1080 (24 in)
Motorola XOOM	1280x800	%78.13	%49.38
Apple iPad 2	1024x768	%58.63	%37.06
Apple iPhone 4S	960x640	%46.89	%29.63
Samsung Galaxy S2	800x480	%30.00	%18.52

Table 1.2. Resolution comparison of desktop and mobile displays

of output pixels. Memory bandwidth is also a major power drain, making limiting bandwidth doubly important.

Within mobile devices, there is also a large spread of resolution sizes within mobile devices, especially between tablets and phone form factors, so testing on multiple devices is important, for performance testing as well as application useabilty concerns.

1.3.2 Framebuffer Bandwidth

Basic rendering can consume surprisingly significant amounts of memory bandwidth. Assume the common configuration of 16-bit color with 16-bit depth [Android 11], with a reasonable 1024x768 display. Accessing every pixel in the framebuffer 60 times a second takes 94MB/s of bandwidth. So simply to write all the pixels in a scene every frame, at 60 frames a second, with 0% overdraw, takes 94MB/s of bandwidth.

However, assuming an IMR architecture, to be able to render a scene, we also usually perform a depth-buffer read for each rendered pixel. Both the depth-buffer and the color buffer are also usually cleared each frame. And when applications write to the color buffer while rendering the scene, they also generally write the fragment depth to the depth-buffer.

The memory bandwidth consumption of the final framebuffer doesn't end when the application is done writing it either. After `eglSwapBuffers`, it may need to be composited by the platform-specific windowing system, and then scanned out to the display. Unlike on desktop systems which often have dedicated graphics or framebuffer memory, this will also consume system memory bandwidth. This will consume an additional 96MB/s of bandwidth just for scanout, or at least an additional 288MB/s with composition(read, write to composited framebuffer, and scan-out).

Thus basic clear-fill-and-post operation consumes 564-752mb/s of bandwidth, so even basic operations consume a significant amount of memory bandwidth; anything interesting the application is doing only costs more bandwidth. If a 32-bit framebuffer is used, this number will be even greater. To put this in context, the ipad2 has been benchmarked at 2.3GB/s for stdlib writes [Anand Lal Shimpi 11] 1.1 for more comparisons.

Applications using a 32-bit framebuffer that may be bandwidth-bound should experiment with a lower-precision format. Since the output framebuffer is not often used in subsequent calculations, the loss of numerical precision is not propagated and magnified. One valid concern is banding or quantization of smooth gradients [Guy 10]. However, this may be more of an issue in photography and media applications rather than games and 3d-applications, just because of the nature of the produced content.

1.3.3 Texture Bandwidth

Since texture accesses are often performed at least once per-pixel, these can be another large source of bandwidth consumption.

One simple way to reduce bandwidth is to lower the texture resolution. Fewer texels, besides a smaller memory footprint, means better texture cache utilization and more efficient filtering. The framebuffer resolution usually can't be lowered, since native resolution is expected. Texture sizes are more flexible, particularly if they represent low-frequency signals like illumination. Low-frequency textures could even be demoted to vertex attributes, and interpolated. If assets have been ported from desktop, there may be room for optimization here.

For static textures (as opposed to textures resulting from off-screen render targets) texture compression is another great way to save bandwidth, loading-time, memory footprint, and disk space. Even though work must be done to decompress the texture data when it is used, the smaller size of compressed textures makes them friendlier to texture caching and memory bandwidth, increasing run-time performance.

One complication is that there are multiple incompatible formats for texture compression supported via OpenGL ES2 extensions. Example formats would be S3TC, available on NVIDIA Tegra, and PVRTC, available on ImaginationTech SGX [Motorola 11].

To support texture compression on multiple devices, an application must either package multiple versions of its assets and dynamically choose the correct ones, or perform the compression at run-time/install-time. Performing the compression at run or install-time must be done carefully to not slow down the application, and gives up many of the benefits of improved loading-time and disk-space, as well as the internet bandwidth required for installation/download of the application on mobile devices. S3TC has compression ratios between 4:1 and 8:1 [Domine 00].

1.4 Reducing Fragment Workload

Due to the limited compute and bandwidth available on mobile devices with respect to the large number of pixels and the complexity of modern rendering, fragment shading is often a bottleneck for mobile GPUs. However, fragment shading can be improved in other ways than just simplifying shading.

1.4.1 Overdraw and Blending

Overdraw is when pixels that have previously been shaded are overwritten by later fragments in a scene. On IMRs, overdraw wastes fragment shading work, since the previous computed pixel value is over-written and lost. The additional framebuffer writes for over-written pixels also waste bandwidth.

On IMR GPUs, this extra bandwidth consumption and fragment work can be limited by sorting and rendering geometry from front to back. This is especially practical for static geometry which can be processed into a spatial data structure during an asset export step. An additional heuristic for games is to render the player character first and the sky-box last. [Pranckevicius and Zioma 11]. For batches where front-to-back object sorting is not practical, for example with complicated, interlocking geometry or heavy use of alpha-testing, a depth pre-pass can be used to eliminate redundant pixel calculations, at the cost of repeated vertex shading work, primitive assembly, and depth-buffer access.

The idea of a depth-pre-pass is to bind a trivial fragment shader and render the scene with color writes disabled. Depth testing proceeds as normal and fragment ordering is resolved. The normal fragment shader is then bound and the scene is re-rendered. In this manner, only the final fragments that affect the scene color are rendered. Clearly, this only works for opaque objects.

Even without overdraw, heavy amounts of overlapping geometry can still be expensive because of the depth-buffer reads needed to reject pixels. Primitive assembly, rasterization, and the pixel reject rate can also become limiting for large areas like sky-boxes [Pranckevicius and Zioma 11].

One type of effect that can be particular expensive in terms of fragment shading and read and write bandwidth is particle effects rendered via multiple overlapping quads with blending. Each layer of overlap requires a read and write of the existing framebuffer value, and an additional fragment computation and blending operation. This generates large amounts of framebuffer traffic and computation for the number of final pixels calculated.

1.4.2 Full-Screen Effects

Full-screen post-processing effects are a major tool for visual effects in modern games and graphics applications, and have been an area of innovation in recent years. Common applications of full-screen post-processing in games are motion-blur, depth-of-field, screen-space ambient occlusion, light bloom, color filtering, and tone-mapping. Other applications such as photo-editing tools may use full-screen or large-area effects for composition, blending, distortion and filtering.

Full-screen post-processing is a powerful tool to create effects but it is an easy way to consume large amounts of bandwidth and fragment processing. Such effects should be carefully weighed for their worth, and are prime candidates for optimization.

A full-screen pass implies at least a read and write of the framebuffer at full resolution, which at 16-bit color and a 1024x768 resolution means 188MB/s bandwidth. Even with tiling architectures, a post-processing pass means a round-trip to external memory. One way to optimize these effects is to remove the extra full-screen pass. Some post-processing effects such as color-filtering or tone-mapping that don't require knowledge of neighboring pixels or feedback from rendering may be merged into the fragment shaders for the objects themselves. This may require the use of *uber-shaders* or shader generation, to allow for natural editing of object fragment shaders while appending post-processing effects.

If the additional pass cannot be eliminated, then all layered full-screen post-processing effects can be coalesced into a single additional pass. This saves redundant round-trips to framebuffer memory.

Device	color_clear	vertex_lighting	one_tap_post	five_tap_post
Motorola Droid X	3.67GPps	234MPps	5.36MPps[1]	5.7MPps[1]
LG Thunderbolt	305MPps	48.7MPps	30MPps	20.36MPps
Dell Inspiron 520	1.92GPps	231MPps	139MPps	120MPps
Desktop System*	13.8GPps	2.95GPps	1.73GPps	1.29GPps

Table 1.3. Performance for different shading and pass configurations. All tests used 1024x1024 16-bit offscreen depth and color buffers as the main framebuffer, with a 32-bit RGBA intermediate color buffer and 16-bit depth buffer where applicable. vertex_lighting renders a synthetic scene with vertex lighting, a per-pixel texture lookup, and 39200 triangles with 0% overdraw. Five_tap_post and one_tap_post draw the same scene with five- and one- sample full-screen post-processing passes, respectively. All units are pixels per second. Desktop system has an Intel Core 2 Quad and NVIDIA 8800 GTS.

One limitation of OpenGL ES 2.0 is the poor support for *Multiple Render Targets (MRT)*, which allow multiple output buffers from a fragment

shader. This makes deferred shading impractical, it relies on separate *geometry buffers* to store different geometry attributes, but without MRTs, this requires rendering rendering a full pass of the scene for each. Even if MRTs were available however, the additional bandwidth cost of reading and writing multiple full-screen intermediate buffers might rule out deferred shading .

1.4.3 Off-screen Passes

Similar to full-screen effects are effects requiring off-screen render targets like environmental reflections, depth-map shadows, and light bloom.

Many of these effects require multiple samples to the off-screen image for a soft effect. Since these textures are rendering targets, they probably don't have full mipmap levels or optimal internal texture layouts for coherent read access, so eliminating the cost of multiple samples into a large texture is particularly important One way to optimize off-screen effects require a blurred image is to take advantage of texture filtering hardware. Rather than rendering a large offscreen image, then taking multiple samples in a fragment shader, the scene can be rendered into a low-resolution offscreen target and blurred via texture filtering.

The main fragment shader for the scene can then bind that target as a texture and read from it with an appropriate texture-filtering mode such as GL_LINEAR. The smaller size of the offscreen target makes this strategy particularly cache-friendly. This may work well for light bloom and environmental reflection, for example. Depending on the effect, an additional Gaussian blurring pass on the off-screen target may be needed, but these can also be accelerated with texture filtering and separable kernels as well [Rideout].

Even when the blurring due to texture filtering is not beneficial, reducing off-screen target resolution may be an easy way to reduce the fragment workload and memory bandwidth without a serious visual impact for low-frequency signals like environmental lighting.

Whenever moving additional computations from a separate full-screen pass into the fragment shader of objects in the scene, it is important on non-tiled architecture to minimize over-draw to avoid wasted work. One advantage of full-screen post-processing in a separate pass is that each pixel is computed exactly once.

1.4.4 Shaving Fragment Work

One area of optimization with a significant amount of leverage is optimizing fragment shaders. Shaders tend to be fairly small and simple, but the sheer number of pixels and amount of computation makes non-trivial fragment

shading a major bottleneck on both TBDR and IMR GPUs. Optimizations here will probably have some effect on visual quality, but it may well be worth the gain in performance.

For static geometry, baking most of the illumination into light-maps will save computation at run-time, and allow the use of more advanced lighting techniques than would otherwise be affordable [Miller 99] [Unity 11]. This does require a well-developed asset pipeline however.

Another classic trick to avoid floating-point work and special functions in fragment shaders is to approximate a complicated function with a look-up texture [Pranckevicius 11]. This allows the use of much more elaborate BDRF's. This also allows for effects that would be difficult to achieve purely procedurally [Jason Mitchell 07]. 1-D look-up textures may be particular cache-friendly, and with a smooth input parameter should have good locality of reference.

However, fragment shaders with multiple texture fetches may already be bound by texture fetch. Large amounts of state per fragment may also limit the number of in-flight fragments, which affects the ability of the GPU to hide texture fetch latency.

1.4.5 Vertex vs. Fragment Work

For lighting and shading the primary objects in our scene, traditional IMR wisdom states that moving computations like lighting, specularity, and normalization from per-fragment to per-vertex and then interpolating can save performance at the cost of image quality, and this is still very true for IMR.

However, for TBDRs, this performance wisdom is more dubious. This is because mobile devices do have smaller screens with fewer pixels to be shaded that desktop, and because TBDRs must perform all vertex computations for each tile [Apple 11]. TBDRs are more likely to be vertex-bound, and Unity recommends 40k or less vertices on recent iOS devices, which use Imagination Tech SGX GPUs [Unity 11].

This means that heavy vertex shaders, even if they save fragment work, may be a performance drag on TBDR. This is particularly true since TBDRs since they perform little-to-no redundant fragment work. When working with IMRs, lifting computation from the fragment shader to the vertex shader is likely a performance win, and becoming vertex-bound is somewhat less of a concern.

Another consideration to relationship between vertex and fragment shaders is that adding too many additional varyings can be a drag on performance, since they must all be interpolated, and a large amount of per-fragment memory may limit the number of fragments that can be in-flight at once. A large number of varyings may also thrash the post-transform

cache, which stores the results of vertex shading, making vertex processing more expensive. So thinning the interface between vertex and fragment shading can be valuable.

Vertex processing is more of a bandwidth drain on tiling architectures, since the attributes probably are pulled again for each tile (unless they hit in a pre- or post- transform cache) To lower this bandwidth, a lower-precision buffer format may be used such as *OES_vertex_half_float*. Interleaved vertex data, which interleaves the attributes for each vertex in the same buffer, is also much more efficient to fetch, since an entire vertex can be fetched in one linear read [Apple 11]. If there is a pre-transform vertex attribute cache, which stores fetched vertex attributes (with some locality), this will make more efficient use of it.

1.5 Conclusion

OpenGL ES is a fundamental component of the modern mobile experience (for UI rendering and composition) [Guy and Haase 11] and presents a huge market and potential impact for OpenGL developers. However, driven by explicit consumer demand for long battery life and slender devices on one hand, and large, brilliant displays with perfectly smooth rendering on the other, performance must be a dominant consideration during development. The wide range of devices in the market, with differing in age, resolution, and capability, only make this more difficult.

One important question is if the significant difference in performance between mobile and desktop GPUs will continue to be a dominating consideration, or if this is something something that the steady march of semiconductor process and architectural improvements will soon make irrelevant. Looking at the projected roadmaps for mobile GPU vendors, the compute power of mobile GPUs should indeed increase significantly over the next few years. However, other limits, including bandwidth and power consumption, are more fundamental and cannot be conquered as easily.

The expected workloads of mobile devices are also changing. Sprite-based games and 2-d workloads are still very important, but several publishers have produced mobile ports of desktop game engines, and games with console or desktop levels of rich game worlds and visual quality. These games raise the bar for what is considered possible and now expected on mobile systems, and present challenges in terms of the amount of geometry, assets, and visual effects they require. The main strategy to deliver on these promises is a measured assessment of a platform's capabilities and limitations paired with an understanding and quantification of the costs of different effects and rendering techniques.

In short, while developing a fast and efficient application for mobile

devies takes thought, careful measurement and budgeting, and creative corner-cutting, with a consciousness to the costs and limitations involved, developers can deliver beautiful and compelling graphics and an experience users can barely believe is possible.

Bibliography

[Anand Lal Shimpi 11] Vivek Gowri Anand Lal Shimpi, Brian Klug. "iPad2 Preview." http://www.anandtech.com/show/4215/apple-ipad-2-benchmarked-dualcore-cortex-a9-powervr-sgx-543mp2/2, 2011.

[Android 11] Google Android. "GLSurfaceView." http://developer.android.com/reference/android/opengl/GLSurfaceView.html, 2011.

[Antochi 04] Iosef Antochi. "Memory Bandwidth Requirements of Tile-Based Rendering." p. 10. SAMOS, 2004.

[Apple 11] Apple. "Best Practices for Working with Vertex Data." http://developer.apple.com/library/ios/#documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/TechniquesforWorkingwithVertexData/TechniquesforWorkingwithVertexData.html#//apple_ref/doc/uid/TP40008793-CH107-SW1, 2011.

[Domine 00] Sebastian Domine. "Using Texture Compression in OpenGL." http://www.oldunreal.com/editing/s3tc/ARB_texture_compression.pdf, 2000.

[Elhasson 05] Ikrima Elhasson. "Fast Texture Downloads and Readbacks using Pixel Buffer Objects in OpenGL." http://developer.download.nvidia.com/assets/gamedev/docs/Fast_Texture_Transfers.pdf?display=style-table, 2005.

[Guy and Haase 11] Romain Guy and Chet Haase. "Android 4.0 Graphics and Animations." http://android-developers.blogspot.com/2011/11/android-40-graphics-and-animations.html?, 2011.

[Guy 10] Romain Guy. "Bitmap quality, banding and dithering." http://www.curious-creature.org/2010/12/08/bitmap-quality-banding-and-dithering/, 2010.

[Jason Mitchell 07] Dhabih Eng Jason Mitchell, Moby Francke. "Illustrative Rendering in Team Fortress 2." International Symposium on Non-Photorealistic Animation and Rendering, 2007.

[Klug and Shimpi 11a] Brian Klug and Anand Lal Shimpi. "LG Optimus 2X & NVIDIA Tegra 2 Review: The First Dual-Core Smartphone." http://www.anandtech.com/show/4144/lg-optimus-2x-nvidia-tegra-2-review-the-first-dual-core-smartphone/5, 2011.

[Klug and Shimpi 11b] Brian Klug and Anand Lal Shimpi. "Samsung Galaxy S 2 (Internation) Review - The Best, Redefined." http://www.anandtech.com/Show/Index/4686?cPage=13\&all=False\&sort=0\&page=15\&slug=samsung-galaxy-s-2-international-review-the-best-redefined, 2011.

[Miller 99] Kurt Miller. "Lightmaps (Static Shadowmaps)." http://www.flipcode.com/archives/Lightmaps_Static_Shadowmaps.shtml, 1999.

[Motorola 11] Motorola. "Understanding Texture Compression." http://developer.motorola.com/docstools/library/understanding-texture-compression/, 2011.

[Pranckevicius and Zioma 11] Aras Pranckevicius and Renaldas Zioma. "Fast Mobile Shaders." http://blogs.unity3d.com/2011/08/18/fast-mobile-shaders-talk-at-siggraph/, 2011.

[Pranckevicius 11] Aras Pranckevicius. "iOS Shader Tricks, or it's 2001 all over again." http://aras-p.info/blog/2011/02/01/ios-shader-tricks-or-its-2001-all-over-again/, 2011.

[Rideout] Philip Rideout. "OpenGL Bloom Tutorial." http://prideout.net/archive/bloom/.

[Unity 11] Unity. "Optimizing Graphics Performance." http://unity3d.com/support/documentation/Manual/Optimizing\%20Graphics\%20Performance.html, 2011.

[Walton 10] Steven Walton. "Nvidia GeForce GTX 480 Review: Fermi Arrives." http://www.techspot.com/review/263-nvidia-geforce-gtx-480/page2.html, 2010.