Exploring Mobile vs. Desktop OpenGL Performance

24

Jon McCaffrey

24.1 Introduction

The stunning rise of mobile platforms has opened a new 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 making the transition from desktop to mobile need to be aware of the limits and capabilities of mobile devices to create the best experience possible for the given hardware resources. This chapter surveys mobile GPU design decisions and constraints, and then explores how these affect classic rendering paradigms.

First, we examine how mobile and desktop GPUs differ in design goals, scale and architecture. We then look at memory bandwidth, which greatly affects performance and device power consumption, and break down the contributions of display, rendering, composition, blending, texture access, and antialiasing. After that, our focus shifts towards optimizing fragment shading for limited compute power. We will look at ways to eliminate shading work entirely if we can or perform operations more efficiently if we can't

Finally, we will discuss the relationship between vertex and fragment shaders and how it is affected by different mobile GPU architectures and will end with some tips for optimizing vertex data for efficient reads and updates.

24.2 Important Differences and Constraints

24.2.1 Differences in Scale

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

Power consumption is a major concern for mobile platforms that is much less pressing on the 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 radio take a great deal of power, but OpenGL applications contribute to power consumption, especially through computation 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, 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 handheld 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 three billion transistors on recent models [Walton 10]. 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 systems, however, the GPU is usually one component on an integrated *system on a chip* (SoC) designed for mobile and embedded applications, which means that a mobile GPU is a fraction of the cost and area of a desktop GPU.

24.2.2 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 tile-based 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 (see Figure 24.1). If a fragment passes depth testing (assuming the platform has early-z), it will be shaded and its output color will be written to the framebuffer. However, a later fragment may overwrite this pixel, nullifying the earlier work done and writing the framebuffer again. This behavior is known as overdraw. If the frame-

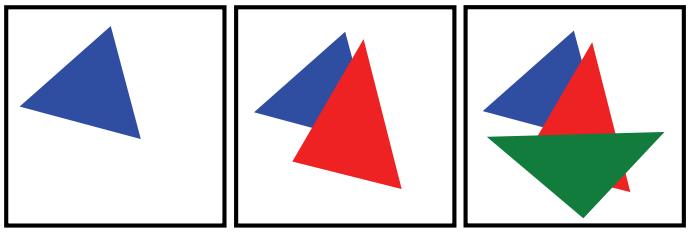

Figure 24.1. IMRs render each primitive only once and render the entire framebuffer in a single pass.

buffer is stored in DRAM external to the GPU, this means a relatively costly and slow memory access was wasted. 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 (see Figure 24.2). All draw commands are buffered. At the end of the frame, for each tile, all geometry overlapping that tile is transformed, clipped, and rasterized into a framebuffer cache. Once the final values for all pixels have been resolved, the entire tile is written out to memory from the framebuffer cache. This saves redundant framebuffer writes and allows for fast depth-buffer access since depth testing and depth and color 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.

There is an additional group of tilers which use *tile-based deferred rendering* (TBDR), for example, the Imagination Tech SGX family. The idea is to rasterize all primitives 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 written to the framebuffer exactly once.

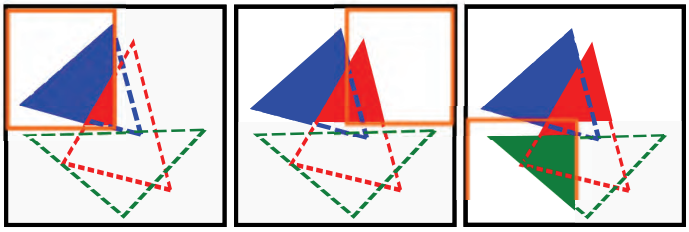

Figure 24.2. Tiling architectures divide the scene into tiles and render all primitives into each tile using a fast framebuffer cache.

Tiling doesn't come for free. The scene geometry must be retransformed and clipped for each tile, so, to maintain a balanced pipeline, additional vertex processing power is needed. Bandwidth will also be spent rereading vertex data for each tile. The digital logic for tiling, repeated vertex shading, and a fast framebuffer cache also takes transistors from raw fragment-shading horsepower. Tiling also requires buffering commands deeply, leading to a more complicated hardware and driver implementation, and the fact that the rendering of each primitive cannot be neatly placed in a single interval of time makes performance analysis more difficult. For more tips on performance-tuning specific to tiling architectures, see Chapter 23.

Architecturally, 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-platform releases.

24.2.3 Differences in Memory Architecture

On desktop systems, middle to high-end GPUs are discrete devices that communicate with the rest of the system via a peripheral bus like PCI-e, although some desktop CPUs are now shipping with capable integrated GPUs, for example, the AMD Llano processors. 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 a wide memory interface. There are no other components competing for this bandwidth 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. To save cost, power, die size, and package complexity, the GPU shares the RAM and memory interface with the other components. This is known as a *unified memory architecture* (UMA). A common memory type is 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 GPUs the peripheral bus could become a bottleneck for transfers, especially for non–PCI-e 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-side data with `glMapBufferOES`, there are fewer performance cliffs lurking in transfers between

OpenGL client and server data, and using client-side data for dynamic vertices or indices may not have as great a performance penalty. Data transfer and command latencies from the GPU to the CPU are also likely to be lessened.

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 runtime more difficult.

24.3 Reducing Memory Bandwidth

As shown in Table 24.1, memory bandwidth pressure is one of the major performance pressures on mobile devices, especially on games and other applications that also perform heavy amounts of CPU-side work during the frame, or in multimedia applications which have additional bandwidth clients besides the GPU and display.

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

Device	CPU Write Bandwidth (GB/s)	GPU Write Bandwidth (GB/s)
Motorola Xoom	2.6	1.252
Motorola Droid X	1.4	6.8
LG Thunderbolt	0.866	0.518
Dell Inspiron 520	4.8	3.8
Desktop System	14.2	25.7

Table 24.1. Write bandwidth for CPU and GPU on different devices. CPU write bandwidth estimated by `memset`. GPU bandwidth estimated by `glClear` followed by `glFinish`. Desktop system has an Intel Core 2 Quad and NVIDIA GeForce 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. The Droid X GPU write bandwidth score is high enough that it may not actually be writing the framebuffer each time (i.e., coalescing redundant clears or setting a cleared flag).

24.3.1 Relative Display Sizes

Despite the tight power and cost constraints for mobile devices, the display resolutions of modern mobile devices are a considerable fraction of the resolutions of desktop displays. Even though the display sizes are smaller, mobile devices often have a higher pixel density to be viewable at a close distance (see Table 24.2).

With the limited fragment shading throughput and memory bandwidth of mobile devices, these comparatively large display sizes mean that fragment shading and

Device	Resolution	% of 1280 × 1024	% of 1920 × 1080
Motorola Xoom	1280 × 800	78.13	49.38
Apple iPad 2	1024 × 768	58.63	37.06
Apple iPhone 4S	960 × 640	46.89	29.63
Samsung Galaxy S2	800 × 480	30.00	18.52

Table 24.2. Resolution comparison of desktop and mobile panels.

full-screen or large-quad operations can easily become a bottleneck since these requirements scale proportionally with the number of output pixels. Memory bandwidth is also a major power drain, making limiting bandwidth doubly important. Common large-quad operations include postprocessing effects and user-interface composition.

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

24.3.2 Framebuffer Bandwidth

Basic rendering can consume surprisingly significant amounts of memory bandwidth. Assume the framebuffer has 16-bit color with 16-bit depth [Android 11] and a 1024 × 768 resolution. Accessing every pixel in the framebuffer 60 times a second takes 94MB/s of bandwidth, so to write all the pixels' colors 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 desktop systems which often have dedicated graphics or framebuffer memory, this will also consume system memory bandwidth. This will consume 94MB/s of bandwidth just for scanout, or at least 288MB/s with composition (read, write to composited framebuffer, and scanout).

Thus, with a depth and color clear, one depth-buffer read, a depth- and color-buffer write, and display scanout, a basic clear-fill-and-display operation on an IMR consumes 564–752MB/s of bandwidth, so even simple-use cases consume a significant amount of memory bandwidth; anything interesting the application does only costs more bandwidth. If a 32-bit framebuffer is used, this number will be even

greater. This can be a significant portion of the bandwidth available on a mobile device (see Table 24.1 for bandwidth measurements for some devices).

Tile-based architectures can consume less bandwidth for this basic operation since they ideally handle the depth and color clears and the depth buffer reads within the framebuffer cache. Use of the `EXT_discard_framebuffer` [Bowman 09] extension saves additional bandwidth because it means the calculated depth buffer never needs to be written back to external memory from the framebuffer cache once the frame is complete. So a tile-based architecture will consume at least 188–377MB/s for basic clear-fill-and-display operation.

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 because of the nature of the produced content.

24.3.3 Antialiasing

Antialiasing improves image quality by refining edges that are jagged when rendered. *Supersampling antialiasing* (SSAA) consumes a large amount of extra bandwidth and fragment-shading load since it must render the scene to a larger, high-resolution buffer and then downsample to the final image. *Multisample antialiasing* (MSAA), on the other hand, rasterizes multiple samples per pixel and stores a depth and color for each sample. If all samples in a pixel are covered by the same primitive, the fragment shader will only be run once for that pixel, and the same color value will be written for all samples in that pixel. These samples are then blended to compute the final image [aths 03].

Though using MSAA creates little if any additional fragment-shading work or texture-read bandwidth consumption, it does use a significant amount of bandwidth to read and write the multiple samples for pixels. Tiling architectures may be able to store the samples in the framebuffer cache and perform this blending before write-back to system memory [Technologies 11]. Vendors may perform other optimizations like only storing multiple samples when there is nontrivial coverage information.

24.3.4 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 drawn by frequent offscreen rendering, 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 they are used, the smaller size of compressed textures makes them friendlier to texture caching and memory bandwidth, increasing runtime performance.

One complication is that there are multiple incompatible formats for texture compression supported via OpenGL ES 2 extensions. Example formats are ETC, available on most Android 2.2 devices, S3TC, available on NVIDIA Tegra, and PVRTC, available on ImaginationTech SGX [Motorola 11].

To support texture compression formats on multiple devices, an application must either package multiple versions of its assets and dynamically choose the correct ones or perform the compression at runtime, loadtime, or install-time. Performing the compression at runtime or install-time must be done carefully to avoid slowing down the application and gives up the benefits of improved loading time and disk space, as well as reduced network bandwidth required to download the application. S3TC has compression ratios between 4:1 and 8:1, so the space and download savings lost are substantial [Domine 00].

As for framebuffer bandwidth, using a texture format with lower precision like RGB565 saves read bandwidth. Unlike texture compression, this applies to textures used as render targets as well.

24.3.5 Texture Filtering and Bandwidth

The texture filtering mode used can also have a significant impact on the memory bandwidth consumed, though texture caching can greatly mitigate these costs. `GL_NEAREST` only needs a single value from the texture. `GL_LINEAR` requires four values for bilinear filtering, but it is unlikely that all four samples will have to be read from external memory since, due to the locality of texture coordinates of neighboring fragments, those samples are possibly already in the texture cache. Trilinear filtering with mipmaps via `GL_LINEAR_MIPMAP_LINEAR` requires eight values per sample, but it can actually increase performance since, for faraway pixels, samples from the smaller mip levels are very likely to hit in texture cache.

Anisotropic filtering via `EXT_texture_filter_anisotropic` prevents surfaces oblique to the viewer from appearing blurry. However, it requires between 2 and 16 taps into a mipmapped texture for each sample. Even if the majority of these taps hit in texture cache, high levels of anistropic filtering stress memory bandwidth and texture filtering hardware.

24.4 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.

24.4.1 Overdraw and Blending

Overdraw is when pixels that have previously been shaded are overwritten by later fragments in a scene (see Figure 24.3). On IMRs and tiling immediate-mode renderers, overdraw wastes completed fragment shading since the previous computed pixel value is overwritten and lost. On IMRs, this also results in an additional framebuffer write, when only one final pixel color needs to be written.

On IMR GPUs, this extra bandwidth consumption and fragment work can be limited by sorting and rendering geometry from front to back (see Figure 24.4). 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 prepass can be used to eliminate redundant pixel calculations, at the cost of repeated vertex shading work, primitive assembly, and depth-buffer access (see Figure 24.5).

The idea of a depth prepass is to bind a trivial fragment shader and render the scene with color writes disabled. Depth calculation, testing, and writes proceed as normal, and the final pixel depth is resolved. The normal fragment shader is then bound, and the scene is rerendered. In this manner, only the final fragments that affect the scene color are rendered. This only works for opaque objects.

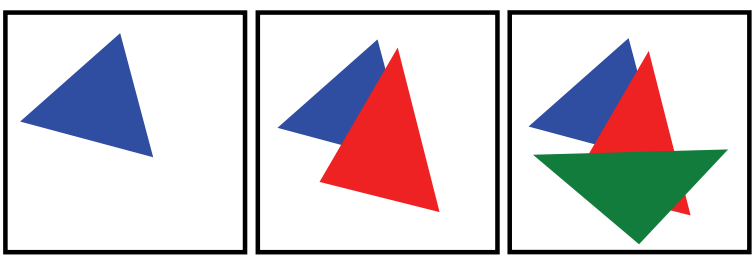

Figure 24.3. In this overdraw case, pixels that are covered by later primitives are shaded more than once

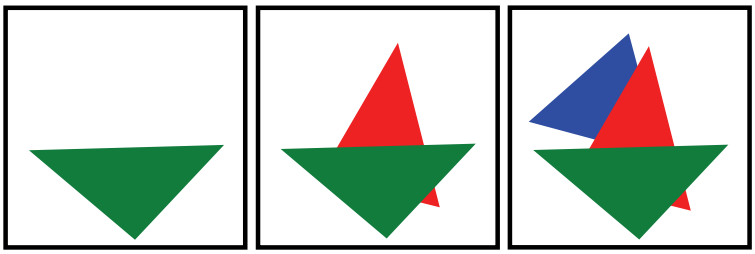

Figure 24.4. By ordering front to back, we no longer shade the covered pixels more than once

Even without overdraw, on IMRs, 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 primitives like sky-boxes [Pranckevicius and Zioma 11].

One type of effect that can be particular expensive in terms of fragment shading and framebuffer bandwidth is particle effects rendered via multiple overlapping quads with blending. On IMRs, each layer of overlap requires a read and write of the existing framebuffer value. For all mobile GPUs, each layers adds additional fragment computation and blending. Some simple effects like a torch flame can be converted into an animated shader, for example, by changing a texture offset each frame. Other effects, like a candle flame, can be done by rendering a mesh of dynamic data instead of many overlapping billboarded quads. This reduces overlap and the resultant blending. When applicable, using opaque, alpha-tested sprites also eliminates the cost of blending.

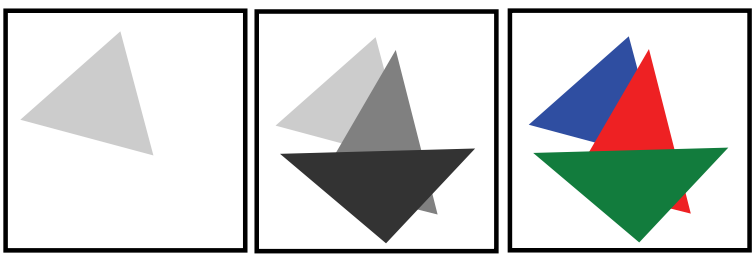

Figure 24.5. A depth prepass resolves ordering in the depth buffer before performing nontrivial fragment shading.

24.4.2 Full-Screen Effects

Full-screen postprocessing 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 postprocessing 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, warping, and filtering.

Full-screen postprocessing 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 1024×768 resolution means 188MB/s bandwidth. Even with tiling architectures, a post-processing pass means a roundtrip to external memory. One way to optimize these effects is to remove the extra full-screen pass. Some postprocessing 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 programmatically appending postprocessing effects.

If the additional pass cannot be eliminated, then all layered full-screen postprocessing effects can be coalesced into a single additional pass. Instead of multiple passes that each read the previous result from a texture and write out a new filtered value, each effect can pass its computed value to the next effect in the same shader. This saves redundant roundtrips to framebuffer memory.

One limitation of OpenGL ES 2.0 is 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 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 make deferred shading prohibitively expensive.

24.4.3 Offscreen Passes

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

Many of these effects require multiple samples of the offscreen 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 of a large texture is particularly important. One way to optimize offscreen effects that require a blurred image is to take advantage of

Device	GPU Arch	MP/s				
		clear	vtx_lgt	frg_lgt	one_tap	five_tap
Motorola Xoom	IMR	626	52.4	24.08	26.13	3.17
Motorola Droid X	TBDR	3670	234	62.9	5.36	5.7
LG Thunderbolt	TB IMR	305	48.7	–	30	20.36
Dell Inspiron 520	IMR	1920	231	204	139	120
Desktop System*	IMR	1380	2950	1920	1730	1290

*Desktop system has an Intel Core 2 Quad and NVIDIA 8800 GTS.

Table 24.3. Performance for different shading and pass configurations. All tests used 1024 × 1024 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. `clear` performs color buffer clear operations. `vtx_lgt` renders a synthetic scene with lighting computed per-vertex, a per-pixel texture lookup, and 39,200 triangles with 0% overdraw. `frg_lgt` uses the same scene and calculates the diffuse illumination in the fragment shader. `five_tap` and `one_tap` draw the vertex lighting scene with five- and one- sample full-screen postprocessing passes, respectively. All units are pixels per second. The Droid X `clear` scores are high enough that it may not actually be writing the framebuffer each time (i.e., coalescing redundant clears, setting a cleared flag, or color compression).

texture-filtering hardware. Rather than rendering a large offscreen image, and then taking multiple samples of 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 offscreen target may be needed, but these can also be accelerated with texture filtering and separable kernels as well [Rideout 11].

Even when blurring due to texture filtering is not beneficial, reducing offscreen target resolution is an easy way to reduce the fragment workload and memory bandwidth without a serious visual impact for effects that only need low-frequency signals like environmental reflections.

Whenever moving additional computations from a separate full-screen pass into the fragment shader of objects in the scene, it is important on non-TBDR architectures to minimize overdraw to avoid wasted work. One advantage of full-screen postprocessing in a separate pass is that each pixel is computed exactly once. Performance of various configurations can be seen in Table 24.3.

24.4.4 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 fragments and amount of floating-point computation makes nontrivial fragment shading a major bottleneck on both tiling 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 and lighting, baking most of the illumination into light maps saves computation at runtime and allows the use of more advanced lighting techniques than would otherwise be affordable [Miller 99, Unity 11]. Light-map generation and export does require a well-developed asset pipeline.

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

Fragment shaders with multiple texture fetches, however, may already be bound by texture fetch. Large amounts of state for each fragment shader may also limit the maximum number of in-flight fragments due to register pressure, which affects the ability of the GPU to hide the latency of texture lookups.

24.5 Vertex Shading

24.5.1 Vertex vs. Fragment Work

Traditional IMR wisdom states that lifting computations like lighting, specularity, and normalization from per-fragment to per-vertex and then interpolating the results can save performance at the cost of image quality, and this is still true for IMRs.

However, for tiling architectures, this performance wisdom is more dubious because tilers must perform all vertex computations for each tile [Apple 11]. Tilers are more likely to be vertex-bound, and Unity recommends 40K or fewer 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 tiling architectures. This is particularly true for 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 less of a concern.

Another consideration to the 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 to store varyings 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 are probably pulled again for each tile unless they hit in a pre- or post-transform cache. To lower this bandwidth, use a lower-precision buffer format such as `OES_vertex_half_float`.

Interleaved vertex data, which interleaves the attributes for each vertex in the same buffer, is also more efficient for attribute fetch since an entire vertex can be fetched in one linear read [Apple 11]. Since memory reads have some granularity, interleaving all the data for a vertex means less unnecessary data will be transferred because it was adjacent to a fetched attribute. If there is a pretransform vertex attribute cache, which stores fetched vertex attributes and the surrounding data, this will make more efficient use of it.

One caveat to interleaving vertex data is if the vertex data is partially dynamic. The most common case is when only positions are updated. A solution is to separate the vertex data into "hot" attributes that are frequently updated and "cold" ones which are mostly static, and store them in separate buffers. This avoids inefficient updates to the "hot" attributes because of a large stride between vertices.

24.6 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 the 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, 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 dominant consideration in application development or if it is 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 climb over the next few years. However, other limits, including bandwidth and power consumption, are more fundamental and cannot be conquered as easily. Desktop and even laptop systems are less tightly constrained on those dimensions.

The expected workloads of mobile devices are also changing. Sprite-based games and 2D 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.

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

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

[Antochi et al. 04] Iosif Antochi, Ben H. H. Juurlink, Stamatis Vassiliadis, and Petri Liuha. "Memory Bandwidth Requirements of Tile-Based Rendering." In *SAMOS*, *Lecture Notes in Computer Science*, edited by Andy D. Pimentel and Stamatis Vassiliadis, pp. 323–332. Springer, 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.

[aths 03] aths. "Multisampling Anti-Aliasing: A Closeup View." http://alt.3dcenter.org/artikel/multisampling_anti-aliasing/index_e.php, 2003.

[Bowman 09] Benji Bowman. "EXT_discard_framebuffer." http://www.khronos.org/registry/gles/extensions/EXT/EXT_discard_framebuffer.txt, 2009.

[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 (International) 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 11] Philip Rideout. "OpenGL Bloom Tutorial." http://prideout.net/archive/bloom/, 2011.

[Technologies 11] Imagination Technologies. "POWERVR Series5 Graphics SGX Architecture Guide for Developers." http://www.imgtec.com/powervr/insider/docs/POWERVR%20Series5%20Graphics.SGX%20architecture%20guide%20for%20developers.1.0.8.External.pdf, 2011.

[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.