OpenGL Insights

Edited by

Patrick Cozzi and Christophe Riccio

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an informa business
AN A K PETERS BOOK

Performance Tuning for Tile-Based Architectures

23

Bruce Merry

23.1 Introduction

The OpenGL and OpenGL ES specifications describe a virtual pipeline in which triangles are processed *in order*: the vertices of a triangle are transformed, the triangle is set up and rasterized to produce fragments, the fragments are shaded and then written to the framebuffer. Once this has been done, the next triangle is processed, and so on. However, this is not the most efficient way for a GPU to work; GPUs will usually reorder and parallelize things under the hood for better performance.

In this chapter, we will examine *tile-based rendering*, a particular way to arrange a graphics pipeline that is used in several popular mobile GPUs. We will look at what tile-based rendering is and why it is used and then look at what needs to be done differently to achieve optimal performance. I assume that the reader already has experience with optimizing OpenGL applications and is familiar with the standard techniques, such as reducing state changes, reducing the number of draw calls, reducing shader complexity and texture compression, and is looking for advice that is specific to tile-based GPUs.

Keep in mind that every GPU, every driver, and every application is different and will have different performance characteristics [Qua 10]. Ultimately, performance-tuning is a process of profiling and experimentation. Thus, this chapter contains very few hard-and-fast rules but instead tries to illustrate how to estimate the costs associated with different approaches.

This chapter is about maximizing performance, but since tile-based GPUs are currently popular in mobile devices, we will also briefly mention power consumption. Many desktop applications will simply render as many frames per second as possible,

always consuming 100% of the available processing power. Deliberately throttling the frame rate to a more modest level and thus consuming less power can significantly extend battery life while having relatively little impact on user experience. Of course, this does not mean that one should stop optimizing after achieving the target frame rate: further optimizations will then allow the system to spend more time idle and hence improve power consumption.

The main focus of this chapter will be on OpenGL ES since that is the primary market for tile-based GPUs, but occasionally I will touch on desktop OpenGL features and how they might perform.

23.2 Background

While performance is the main goal for desktop GPUs, mobile GPUs must balance performance against power consumption, i.e., battery life. One of the biggest consumers of power in a device is memory bandwidth: computations are relatively cheap, but the further data has to be moved, the more power it takes.

The OpenGL virtual pipeline requires a large amount of bandwidth. For a fairly typical use-case, each pixel will require a read from the depth/stencil buffer, a write back to the depth/stencil buffer, and a write to the color buffer, say 12 bytes of traffic, assuming no overdraw, no blending, no multipass algorithms, and no multisampling. With all the bells and whistles, one can easily generate over 100 bytes of memory traffic for each displayed pixel. Since at most 4 bytes of data are needed per displayed pixel, this is an excessive use of bandwidth and hence power. In reality, desktop GPUs use compression techniques to reduce the bandwidth, but it is still significant.

To reduce this enormous bandwidth demand, many mobile GPUs use *tile-based rendering*. At the most basic level, these GPUs move the framebuffer, including the depth buffer, multisample buffers, etc., out of main memory and into high-speed on-chip memory. Since this memory is on-chip, and close to where the computations occur, far less power is required to access it. If it were possible to place a large framebuffer in on-chip memory, that would be the end of the story; but unfortunately, that would take far too much silicon. The size of the on-chip framebuffer, or *tile buffer*, varies between GPUs but can be as small as 16×16 pixels.

This poses some new challenges: how can a high-resolution image be produced using such a small tile buffer? The solution is to break up the OpenGL framebuffer into 16×16 *tiles* (hence the name "tile-based rendering") and render one at a time. For each tile, all the primitives that affect it are rendered into the tile buffer, and once the tile is complete, it is copied back to the more power-hungry main memory, as shown in Figure 23.1. The bandwidth advantage comes from only having to write back a minimum set of results: no depth/stencil values, no overdrawn pixels, and no multisample buffer data. Additionally, depth/stencil testing and blending are done entirely on-chip.

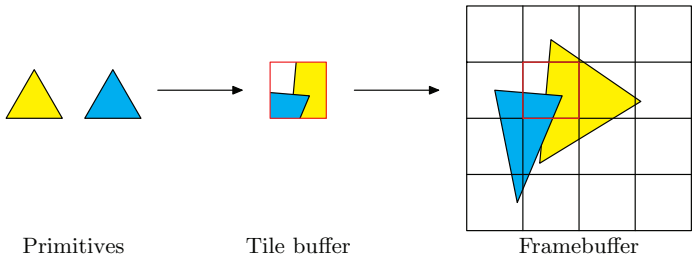

Figure 23.1. Operation of the tile buffer. All the transformed primitives for the frame are stored in memory (left). A tile is processed by rendering the primitives to the tile buffer (held on-chip, center). Once a tile has been rendered, it is copied back to the framebuffer held in main memory (right).

We now come back to the OpenGL API, which was not designed with tile-based architectures in mind. The OpenGL API is *immediate-mode*: it specifies triangles to be drawn in a current state, rather than providing a scene structure containing all the triangles and their states. Thus, an OpenGL implementation on a tile-based architecture needs to collect all the triangles submitted during a frame and store them for later use. While early fixed-function GPUs did this in software, more recent programmable mobile GPUs have specialized hardware units to do this. For each triangle, they will use the `gl_Position` outputs from the vertex shader to determine which tiles are potentially affected by the triangle and enter the triangle into a spatial data structure. Additionally, each triangle needs to be packaged with its current fragment state: fragment shader, uniforms, depth function, etc. When a tile is rendered, the spatial data structure is consulted to find the triangles relevant to that tile together with their fragment states.

At first glance, we seem to have traded one bandwidth problem for another: instead of vertex attributes being used immediately by a rasterizer and fragment shading core, triangles are being saved away for later use in a data structure. Indeed, storage is required for vertex positions, vertex shader outputs, triangle indices, fragment state, and some overhead for the spatial data structure. We will refer to these collective data as the *frame data* (ARM documentation calls them *polygon lists* [ARM 11], while Imagination Technologies documentation calls them the *parameter buffer* [Ima 11]). Tile-based GPUs are successful because the extra bandwidth required to read and write these data is usually less than the bandwidth saved by keeping intermediate shading results on-chip. This will be true as long as the number of post-clipping triangles is kept to a reasonable level. Excessive tessellation into micropolygons will bloat the frame data and negate the advantages of a tile-based GPU.

Figure 23.2(a) shows the flow of data. The highest bandwidth data transfers are those between the fragment processor and the tile buffer, which stay on-chip. Contrast this to Figure 23.2(b) for an immediate-mode GPU, where multisample color, depth, and stencil data are sent across the memory bus.

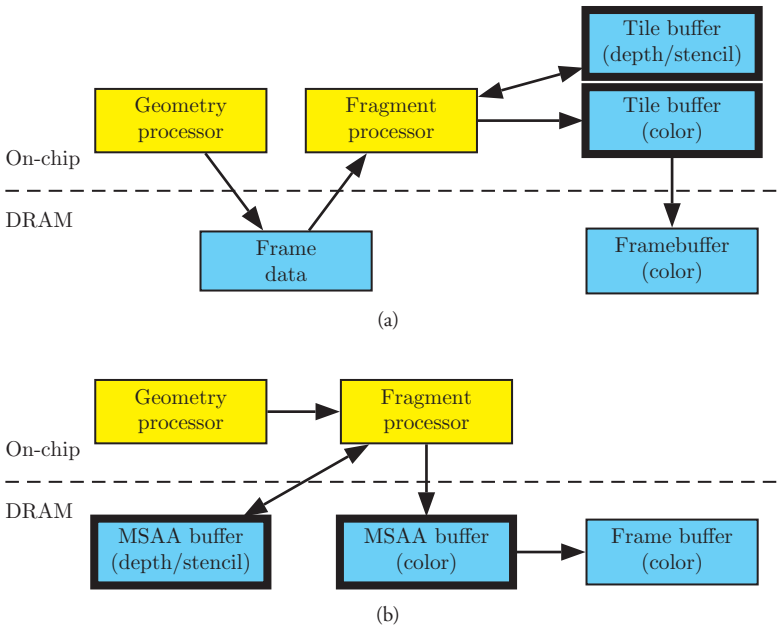

Figure 23.2. Data flow in (a) tiled-based and (b) immediate-mode GPUs for multisampled rendering. Yellow boxes are computational units, blue boxes are memory, and thick borders indicate multisampled buffers. The immediate-mode GPU moves multisampled pixel data on and off the chip, consuming a lot of bandwidth.

23.3 Clearing and Discarding the Framebuffer

When it comes to performance-tuning, the most important thing to remember about a tile-based GPU is that the representation of a frame that is currently being constructed is not a framebuffer but the frame data: lists of transformed vertices, polygons, and state necessary to produce the framebuffer. Unlike a framebuffer, these data grow as more draw calls are issued in a frame. It is thus important to ensure that frames are properly terminated so that the frame data do not grow indefinitely.

When swapping a double-buffered window, the effect on the back buffer depends on the window system bindings used. Both EGL and GLX allow the implementation to invalidate the contents of the back buffer. Thus, the driver can throw away the frame data after each swap and start with a blank slate (with EGL, applications can opt to have the back buffer preserved on a swap; see Section 23.4 for more details).

Things become more difficult when using framebuffer objects, which do not have a swap operation. Specifically, consider the use of `glClear`. Typical desktop GPUs are immediate-mode architectures, meaning that they draw fragments as soon as all the data for a triangle are available. On an immediate-mode GPU, a call to `glClear`

```
glDisable(GL_SCISSOR_TEST);
glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
glDepthMask(GL_TRUE);
glStencilMask(0xFFFFFFFF);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
```

Listing 23.1. Clearing the screen correctly for a tile-based GPU.

actually writes values into the framebuffer and thus can be expensive. Programmers use assorted tricks to avoid this, such as not clearing the color buffer if they know it will be completely overwritten and using half the depth range on alternate frames to avoid clearing the depth buffer. While these tricks were useful in the past, they have been surpassed by hardware-level optimizations, and can even reduce performance by working against these hardware optimizations.

On a tile-based architecture, avoiding clears can be disasterous for performance: since the frame is built up in frame data, clearing all buffers will simply free up the existing frame data. In other words, not only is `glClear` very cheap, it actually improves performance by allowing unneeded frame data to be discarded.

To get the full benefit of this effect, it is necessary to clear everything: using a scissor or a mask or only clearing a subset of color, depth, and stencil will prevent the frame data from being freed. While drivers may detect more cases where clearing can free the frame data, the safest and most portable approach is shown in Listing 23.1.

This should be done at the start of each frame,[1] unless the window system already takes care of discarding the framebuffer contents. Of course, the masks and scissor enable don't need to be set explicitly if they are already in the correct state.

The above discussion about clearing highlights a limitation of the API: `glClear` is a low-level command that specifies buffer contents rather than a high-level hint about how the application is using a buffer. OpenGL ES developers wanting portable performance should consider the `EXT_discard_framebuffer` extension, which provides this hint. The command `glDiscardFramebufferEXT` indicates to the driver that the application no longer cares about the contents of some buffers, allowing the driver to set the pixel values to whatever it wishes. Tiled-based architectures can use this hint to free up the frame data, while immediate-mode architectures can choose to ignore the hint. Listing 23.2 shows an example that can be used in place of Listing 23.1.

```
const GLenum attachments[3] = { COLOR_EXT, DEPTH_EXT, STENCIL_EXT };
glDiscardFramebufferEXT(GL_FRAMEBUFFER, 3, attachments);
```

Listing 23.2. Discarding framebuffer contents with `EXT_framebuffer_discard`.

[1] OpenGL does not define what a "frame" is, and it is a surprisingly slippery concept. In this context, we consider each framebuffer that is generated to constitute a separate frame, even if multiple framebuffer objects are combined to create a single onscreen image.

Discards have another use with framebuffer objects, i.e., render-to-texture. When rendering 3D geometry to a texture such as an environment map, a depth buffer is needed during the rendering but does not need to be preserved afterwards. The application can inform the driver of this by calling `glDiscardFramebufferEXT` after doing the rendering but before unbinding the framebuffer object, and a tile-based GPU may use this as a hint that depth values need not be copied back from the tile buffer to main memory. Although not yet available on desktop OpenGL, `EXT_discard_framebuffer` is likely to be useful for multisampled framebuffer objects as well, where the multisampled buffer may be discarded as soon as it has been resolved into a single-sampled target. At the time of this writing, `EXT_discard_framebuffer` is still relatively new, and some experimentation will be required to determine how effectively the hints are used by any particular implementation.

23.4 Incremental Frame Updates

For a 3D view with a moving camera, such as in a first-person shooter game, it is reasonable to expect every pixel to change from frame to frame, and so clearing the framebuffer will not destroy any useful information. For more GUI-like applications, there may be assorted controls or information views that do not change from frame to frame and which do not need to be regenerated. Application developers using EGL on a tile-based GPU are often surprised to find that the color buffer does not persist from frame to frame. EGL 1.4 allows this to be explicitly requested by setting `EGL_SWAP_BEHAVIOR` on the surface, but it is not the default on a tile-based GPU since it reduces performance.

To understand why back-buffer preservation reduces performance, consider again how a tile-based GPU composes fragments for a single tile. If the framebuffer is cleared at the start of a frame, the tile buffer need only be initialized to the clear color before fragments are drawn, but if the framebuffer is preserved from the previous frame, then the tile buffer needs to be initialized with the corresponding section of the framebuffer before any new fragments are rendered, and this requires bandwidth. The bandwidth cost is comparable to treating the previous framebuffer as a texture and drawing it into the current frame. Although it will depend on the complexity of the scene, it can be faster just to redraw the entire frame than to try to preserve regions from the previous frame.

Qualcomm provides a vendor extension (`QCOM_tiled_rendering`) that addresses this use-case. The application explicitly indicates which region it is going to update, and all rendering is clipped to this region. The GPU then only needs to process the tiles intersecting this region, and the rest of the framebuffer can remain untouched. This extension also includes features similar to `EXT_discard_framebuffer` to allow the user to indicate whether the existing contents of the target region need to be preserved. For example, suppose an application contains a 3D view in the region with offset x, y and dimensions $w \times h$, which is going to be com-

```
glStartTilingQCOM(x, y, width, height, GL_NONE);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
glViewport(x, y, width, height);
// Draw the scene
glEndTilingQCOM(GL_COLOR_BUFFER_BIT0_QCOM);
eglSwapBuffers(dpy, surface);
```

Listing 23.3. Replacing a portion of the framebuffer using `QCOM_tiled_rendering`. `GL_NONE` indicates that the previous contents of the framebuffer for the affected region may be discarded. `GL_COLOR_BUFFER_BIT0_QCOM` indicates that the rendered color data must be written back to the framebuffer. Depth and stencil may be discarded.

pletely replaced, while the rest of the window is static and does not need to be updated. This can be achieved using the code in Listing 23.3 in addition to setting `EGL_SWAP_BEHAVIOR` to `EGL_BUFFER_PRESERVED`.

23.5 Flushing

Tile-based GPUs are sometimes referred to as *deferred* because the driver will try to avoid performing fragment shading until it is required. Eventually, of course, the pixel values will be needed. The following operations will all force the framebuffer contents to be brought up to date:

- `eglSwapBuffers` and its equivalents in other window systems;
- `glFlush` and `glFinish`;
- `glReadPixels`, `glCopyTexImage`, and `glBlitFramebuffer`;
- querying an occlusion query result from an occlusion query in the current frame;
- using the result of render-to-texture for texturing.

Changing framebuffer attachments using either `glFramebufferRenderbuffer` or `glRenderbufferStorage`, or the texture equivalents is also likely to cause a flush, as the frame data will only be applicable to the old attachment.

The following pattern will have very poor performance:

1. Draw some triangles.
2. Use the framebuffer contents.
3. Draw another triangle.
4. Use the framebuffer contents.
5. Draw another triangle. . . .

Each time the framebuffer contents are needed, there will be another fragment-shading pass, which in the worst case could involve a read and a write for every framebuffer pixel just to draw one triangle. Because the cost of each pass is so high, the goal is to have only one pass per frame.

This is true even if the accesses to the framebuffer contents are done entirely on the GPU, such as by accessing the results of render-to-texture or by calling `glReadPixels` with a pixel pack buffer, because each draw-then-access requires the fragment shading to be rerun. Compare this to an immediate-mode GPU, where the cost of `glReadPixels` with a pixel pack buffer will be essentially the same regardless of when it is performed.

In some drivers, `glBindFramebuffer` also starts fragment shading for the framebuffer that has just been unbound. Thus, it is best to bind each framebuffer only once per frame. As an example, consider a scene in which some objects are made shiny by using generated environment maps. A naïve walk of the scene graph might cause each environment map to be generated immediately before drawing the object itself, but it would be better to first generate all the environment maps before binding the window system framebuffer to render the final scene.

Apart from the commands above, there is another situation in which flushing happens. Because the memory usage for the frame data scales with the amount of geometry in the frame, an application that just keeps drawing more geometry without swapping or clearing would eventually run out of memory. To prevent this from happening, the driver will eventually force a flush. This is very expensive because unlike a swap operation, all the buffers, including multisample buffers, are written out to memory and then reloaded for rendering to continue, which can easily consume 16 times the bandwidth of a regular flush.

This means that performance does not scale linearly with the number of vertices. Once an application has sufficiently simple geometry to run at interactive rates, it should be well clear of this performance cliff, but when starting optimization, it is worth checking for this case before estimating a target vertex count from current throughput.

23.6 Latency

Since vertex and fragment processing for a frame happen in separate phases, an application that has balanced demands on the CPU, vertex processor, and fragment processor will have three frames in flight at any time, as shown in Figure 23.3. The exact latency between command submission and completion will depend on which resources are most limited and will also vary over time.

Apart from impacting responsiveness to user input, latency is a concern when results of rendering are read back to the CPU. Synchronous queries such as `glReadPixels` without a pixel pack buffer will stall the CPU until results are available and should almost never be used. But even with asynchronous queries such as occlu-

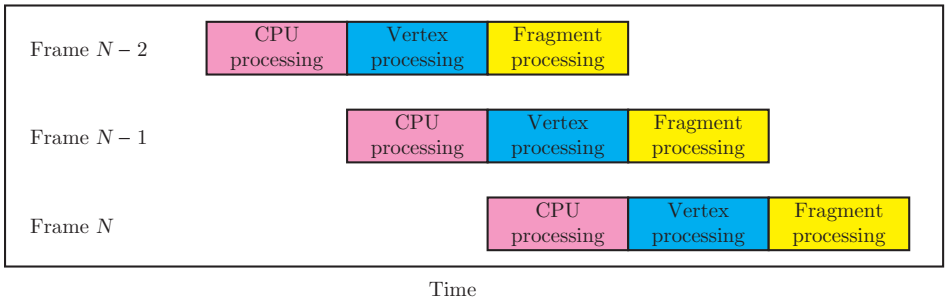

Figure 23.3. Processing pipeline in a tile-based GPU. At any point in time, there can be three frames in different stages of processing. This shows an idealized case with no pipeline bubbles.

sion queries, the result must eventually be read, and doing so too soon will stall the pipeline. If it is acceptable to just wait until the query result is available, then a periodic check of `GL_QUERY_RESULT_AVAILABLE` is sufficient. Code that is written for an immediate-mode GPU that assumes the query result will be available within a fixed number of frames may need to be retuned, either to wait for a larger number of intervening frames or to poll for the result becoming available. Similarly, if `glReadPixels` must be used, performance can be greatly improved at the cost of some latency by rotating between multiple framebuffer objects and reading not the just-rendered frame but a previous one, which is more likely to have completed rendering.

Latency also plays a role when objects are modified since commands bind their resources at the time they are issued. A common example is an animated mesh, where the vertex positions are updated every frame. The previous vertex positions may still be in use for vertex shading in the previous frame, so when the application updates the vertex buffer, the memory the GPU is reading from cannot be touched until the previous frame is complete. In most cases, drivers handle this by making an extra copy of the resource "under the hood" to avoid stalling the pipeline, but on a memory- and bandwidth-constrained mobile device, it is still worth being aware that this *copy-on-write* is happening. The problem becomes worse if a single resource is used multiple times during a frame, interspersed with partial updates, leading to multiple copy-on-writes. If possible, all the updates should be done as a block before the resource is used.

Be particularly careful when using extensions such as `EGL_KHR_image_pixmap` or `GLX_EXT_texture_from_pixmap` to modify operating system pixmaps. Drivers usually have less freedom to move these resources around in memory and may need to stall the pipeline or even flush partial results to the framebuffer and reload them.

The three-phase processing shown in Figure 23.3 means that tile-based GPUs will typically have a higher latency than immediate-mode GPUs, and thus, code that has been tuned for an immediate-mode GPU may need retuning. For some tile-based GPUs, vertex shading for a frame finishes much earlier than fragment shading

(in fact, before fragment shading starts), so the latency for vertex shading will be lower. However, in some cases, parts of vertex shading will be delayed until needed during fragment shading.

23.7 Hidden Surface Removal

When objects overlap in an immediate-mode GPU, causing one pixel to be overwritten by another, there are two costs associated with this: the cost of shading the hidden pixels and the extra bandwidth consumed by the associated framebuffer accesses. In a tile-based GPU, the latter cost is eliminated because only fully rendered tiles are emitted to memory, but the shading cost remains. It is thus still important to do high-level culling and to submit opaque objects in front-to-back order to take advantage of hardware early depth tests. Because the costs are different, however, the optimal balance of CPU load for sorting and GPU load for shading may be different compared to an immediate-mode GPU.

An exception to the above is the PowerVR family of GPUs, which feature per-pixel hidden surface removal during fragment shading [Ima 11]. Before running any fragment shaders, the polygons are preprocessed to determine which fragments potentially contribute to the final result, and only those are shaded. This removes the need to sort opaque geometry. To take full advantage of this, the fragment shader must be guaranteed to replace occluded pixels. The presence of the GLSL `discard` keyword as well as sample masking, alpha testing, alpha-to-coverage, and blending will all disable the optimization as the occluded pixel may potentially impact the final image. Thus, these features should only be enabled for the objects that require them, even at the cost of extra state changes.

Where PowerVR-style hardware hidden-surface removal is not available, another option is to use an initial *depth-only pass*: submit all the geometry once with an empty fragment shader and color writes disabled to populate the depth buffer, and then draw everything again with the real fragment shader. The depth pass will determine the depth of the visible surfaces, and the color pass will then perform fragment shading only for the those surfaces (assuming early depth culling).

This depth-only pass technique can be effective on either an immediate-mode GPU or a tile-based GPU when it eliminates expensive shading computations, but the trade-offs are different. In both cases, the depth pass incurs all the vertex processing and rasterization costs of the color pass (however, the Adreno 200 and possibly others have higher fragment throughput in a depth-only pass [Qua 10]). On an immediate-mode GPU, a depth-only pass incurs a bandwidth penalty since the depth buffer is accessed during both passes. On a tile-based GPU, the depth buffer accesses have no main memory bandwidth penalty, but there is the smaller penalty of duplicating all the geometry in the frame data. Thus, for bandwidth-limited applications a depth-only pass may be effective on a tile-based GPU even when it is not effective on an immediate-mode GPU.

23.8 Blending

On immediate-mode GPUs, blending is usually expensive because it requires a read-modify-write cycle to the framebuffer, which is held in relatively slow memory. On a tile-based CPU, this read-modify-write cycle occurs entirely on-chip and so is very cheap. Some GPUs have dedicated blending hardware which makes the blending operation essentially free, while others use shader instructions to implement blending; hence, blending will reduce fragment shading throughput.

Note that this only addresses the direct cost of the blending operation compared to other transparency or translucency techniques such as alpha tests or alpha-to-coverage. Making an object partially transparent or translucent has indirect costs, as the object can no longer be treated as an occluder for hidden-surface removal. The fragments behind the translucent object must now be processed, whereas previously they could be eliminated by optimizations such as hardware hidden-surface removal or early depth testing.

23.9 Multisampling

Multisampling is an effective technique to improve visual quality without sacrificing as much performance as supersampling. Each framebuffer pixel stores multiple samples, which are averaged together to produce an antialiased image, but fragments generated by rasterization need only be shaded once per pixel. While this keeps fragment shading costs largely the same, it has an enormous bandwidth impact on immediate-mode GPUs: with 4 times multisampling (a common choice), the bandwidth of all framebuffer accesses increases by a factor of 4. Various hardware optimizations reduce this bandwidth overhead to the point where multisampling is practical, but it is still expensive.

In contrast, multisampling in a tile-based GPU can be very cheap, as the multiple samples need only be retained in the on-chip tile buffer, with only the averaged color value written out to framebuffer memory. Thus, multisampling has no impact on framebuffer bandwidth.

There are, nevertheless, two costs: firstly, 4 times multisampling will require four times as much tile buffer memory. Since tile buffer memory is expensive in terms of silicon area, some GPUs compensate for this by reducing the tile size when multisampling is in effect. A reduction in tile size has some impact on performance, but halving the tile size will not halve the performance, and applications limited by fragment shading throughput will see only a minor impact.

The second cost for multisampling (which also affects immediate-mode GPUs) is that more fragments will be generated along object silhouettes. Each polygon will hit more pixels as shown in Figure 23.4. Furthermore, where both the foreground and background geometry contribute to a single pixel, both fragments must be shaded, and so hardware hidden surface removal will cull fewer fragments. The cost of these

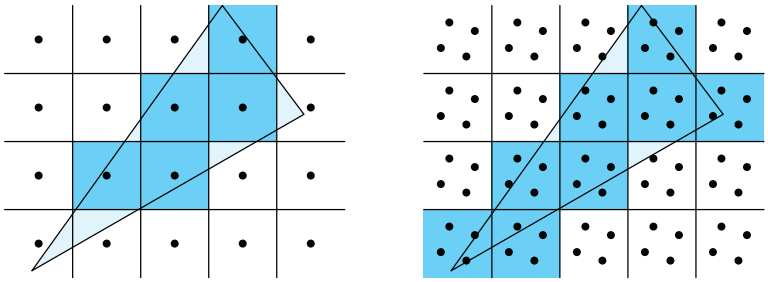

Figure 23.4. The effect of multisampling on fragment shader load. Without multisampling, only pixels whose centers are covered generate fragments (left; blue). If any of the sample points are covered, a fragment is generated, thus leading to slightly more fragments along edges (right).

extra fragments will depend on how much of the scene is made up of silhouettes, but 10% is a good first guess.

23.10 Performance Profiling

On an immediate-mode GPU, the `ARB_timer_query` extension can be used to gauge the cost of rendering some part of the scene: a range of commands to profile is bracketed by `glBeginQuery` and `glEndQuery`, and the time elapsed between these commands in the command stream is measured. This functionality is described in more detail in Chapter 34.

While it is possible to implement this extension on a tile-based GPU, the results will not be useful for anything less than frame granularity. This is because commands are not processed in the order they are submitted: the vertex processing is all done in a first pass, and then fragment processing is ordered by tiles. Thus, profiling will need to rely on more intrusive techniques, such as turning parts of the scene on or off to determine the performance impact. Vendor-specific tools for accessing internal performance counters can also be a great help in identifying which parts of the pipeline are causing bottlenecks.

Apart from post hoc profiling, it is often a good idea to start development with microbenchmarks that measure the performance of specific aspects of the system to determine a budget for triangles, textures, shader complexity, etc. When doing so, keep in mind that submitting too much geometry without swapping will lead to a performance cliff as described in Section 23.5. It is also important to ensure that the commands do actually get executed—placing a `glClear` after draw calls may well cancel those draw calls before they reach the GPU.

23.11 Summary

Every GPU and every driver is different, and different choices in optimizations and heuristics mean that the only way to truly determine the performance impact of design choices on a specific system is to test it. Nevertheless, the following rules of thumb are a good starting point to obtain high performance from a tile-based GPU:

- Clear or discard the entire contents of the color, depth, and stencil buffers at the start of each frame.
- For each framebuffer, bind it once during the frame, and submit all the commands for the frame before unbinding it or using the results.
- Keep latency in mind when using occlusion queries or other mechanisms to retrieve the results of commands, and if an application was previously tuned for the latencies of an immediate-mode GPU, it may need to be retuned.
- Keep polygon counts to a reasonable level, and in particular avoid micropolygons.
- On hardware with built-in hidden surface removal (PowerVR), there is no need to sort opaque objects front to back; on other hardware, this should be done. Also consider using a depth-only pass.
- Take advantage of cheap multisampling.
- Remember that on mobile devices, performance must be balanced against power consumption.

Bibliography

[ARM 11] ARM. *Mali GPU Application Optimization Guide*, 2011. Version 1.0.

[Ima 11] Imagination Technologies Ltd. *POWERVR Series5 Graphics SGX Architecture Guide for Developers*, 2011. Version 1.0.8.

[Qua 10] Qualcomm Incorporated. *Adreno™ 200 Performance Optimization: OpenGL ES Tips and Tricks*, 2010.