8

Bug-Free and Fast Mobile WebGL

Olli Etuaho

8.1 Introduction

WebGL availability on mobile browsers has recently taken significant leaps forward. On leading mobile platforms, WebGL has also reached a performance and feature level where it is a feasible target for porting desktop apps. Often, content written on desktop will simply work across platforms. Still, special care needs to be taken to make a WebGL application run well on mobile devices. There are some nuances in the WebGL specification that may go unnoticed when testing only on desktop platforms, and limited CPU performance means that JavaScript and API usage optimizations play a much larger role than on desktops and high-end notebooks. Mobile GPU architectures are also more varied than desktop ones, so there are more performance pitfalls to deal with (Figure 8.1).

This chapter focuses on shader precision, which is not widely understood in the community and thus has been a frequent source of bugs even in professionally developed WebGL libraries and applications. I will also touch upon some other common sources of bugs, and provide an overview of optimization techniques that are particularly useful when writing applications with mobile in mind. As ARM SoCs make inroads into notebooks, such as the recent Tegra K1 Chromebooks, this material will become relevant even for applications that are only targeting a mouse-and-keyboard form factor.

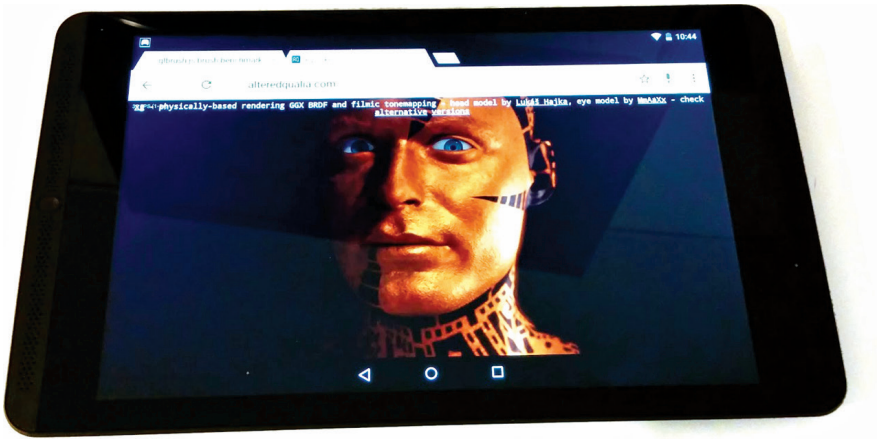

Figure 8.1

An example of what is possible with WebGL on mobile platforms: a physically based rendering demo running at 30 FPS in Chrome on an NVIDIA Shield Tablet (Tegra K1 32-bit).

8.1.1 Developer Tools

Modern browsers include many developer tools that help with improving a WebGL app's mobile compatibility and performance. In Chrome 42, *JavaScript CPU Profile*, *Device mode,* and *Capture Canvas Frame* tools are particularly useful for mobile WebGL development. They also have equivalent alternatives in Firefox, where we can use the *Profiler* tab of dev tools, the *Responsive Design Mode*, and the *Canvas* tab or the *WebGL Inspector* extension. Some equivalent tools are also found in Internet Explorer's F12 developer tools and Safari's Web Inspector. Familiarize yourself with these tools in your preferred development environment, as I'll be referring to them throughout the rest of the chapter. I'm going to use the tools found in Chrome as an example, but often enough the basic processes can be generalized to tools provided by other browsers.

The Device mode tool mostly helps with getting viewport settings right and making sure the web application's UI can be used with touch. It does not emulate how the GPU behaves on the selected device, so it does not completely eliminate the need for testing on actual hardware, but it can help when getting started with mobile development. Firefox's Responsive Design Mode does not have the touch features of Chrome's Device mode, but can still also help.

The JavaScript CPU Profile and Capture Canvas Frame tools help with finding application bottlenecks and optimization opportunities. In Chrome 39, Capture Canvas Frame is still experimental and needs to be enabled from the flags page* with `#enable-devtools-experiments` and from developer tools settings. It's possible to use these inside the Inspect Devices tool† to perform the profiling on an Android device connected to your development workstation using USB.

* chrome://flags
† chrome://inspect

Firefox also has some tools that don't have exact counterparts in Chrome. In the about:config settings page in Firefox, there are two WebGL flags that are particularly useful. One is `webgl.min_capability_mode`. If that flag is enabled, all WebGL parameters such as maximum texture size and maximum amount of uniforms are set to their minimum values across platforms where WebGL is exposed. This is very good for testing if very wide compatibility with older devices is required. More on capabilities can be found from the "Capabilities Exposed by getParameter" section of this chapter.

The other interesting Firefox flag is `webgl.disable-extensions`. This is self-explanatory; it is good for testing that the application has working fallbacks in case some WebGL extension it uses is not supported.

8.2 Feature Compatibility

8.2.1 Shader Precision

Shader precision is the most common cause of issues when WebGL content developed on desktop is ported to mobile. Consider the GLSL example in Listing 8.1, which is designed to render a grid of randomly blinking circles as in Figure 8.2.

It looks like the developer is aware that some mobile devices do not support highp precision in fragment shaders [ESSL100 §4.5.2], so the first three lines intended to set mediump precision when running on mobile devices have been included. But there is already the first small error: including `#ifdef GL_ES` is unnecessary in WebGL, where the `GL_ES` macro is always defined. It is only useful if the shader source is directly used in both OpenGL and WebGL environments.

However, the critical error is much more subtle, and it might come as a surprise that the pseudorandom number generator is completely broken on many mobile devices. See Figure 8.3 for the results the shader produces on one mobile chip.

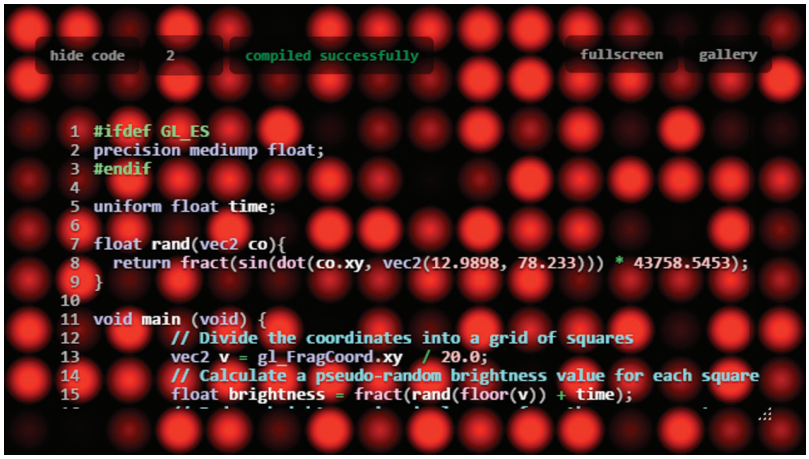

Figure 8.2

Intended effect of Listing 8.1, captured on a notebook with an NVIDIA GeForce GPU.

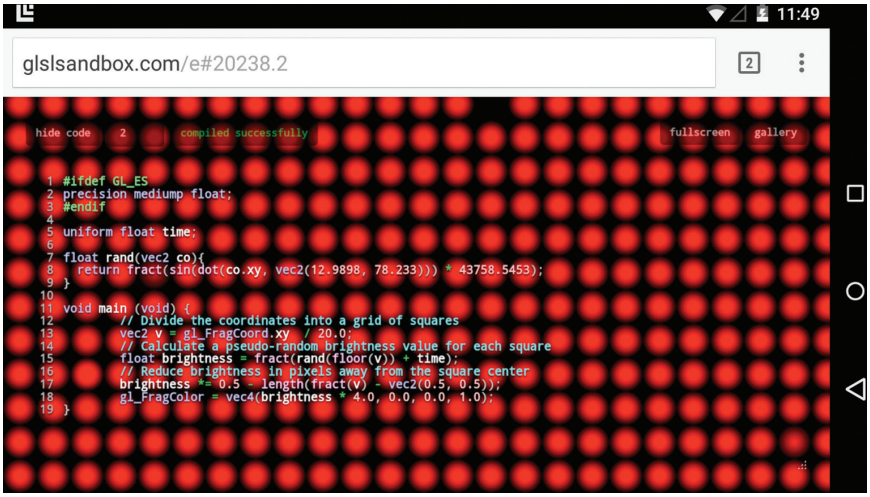

Figure 8.3

The example shader running on a device which includes 16-bit floating-point hardware.

The reason for the different rendering is that the lowest floating-point precision most desktop GPUs implement in hardware is 32 bits, and all floating-point calculations regardless of GLSL precision run on that same 32-bit hardware. In contrast, the mobile device represented in Figure 8.3 includes both 32-bit and 16-bit floating-point hardware, and calculations specified as lowp or mediump get performed on the more power-efficient 16-bit hardware.

The differences between mobile GPUs get a lot more varied than that, so on other devices we can see other kinds of flawed rendering. The GPUs use different bit counts in their floating-point representation, use different rounding modes, and may handle subnormal numbers differently [Olson 13]. This is all according to the specification, so the mobile devices are not doing anything wrong by performing the calculations in a more optimized way. The WebGL specification is just very loose when it comes to floating-point precision.

The calculation that goes wrong in this case is on line 8. The result from `sin(x)` is in the range [–1, 1], so `sin(x) * 43758.5453` will be in the range [–43758.5453, 43758.5453]. The float value range in the minimum requirements for mediump precision is [–16384, 16384], and values outside this range might be clamped, which is the first possible cause of rendering issues. Even more importantly, the relative precision in the minimum requirements for mediump floats is 2^{-10}, so only the first four decimals will count. For example, the closest possible representation of 1024.5 in minimum mediump precision is 1024.0, so `fract(1024.5)` computed in mediump precision returns 0.0, and not 0.5. To get a better feel for how lower precision floating-point numbers behave, see this low-precision floating point simulator.*

* http://oletus.github.io/float16-simulator.js/

There are two ways to fix Listing 8.1. The simplest way is to replace the first three lines with:

```
precision highp float;
```

and be done with it. As a result, the shader either works exactly as on common desktop GPUs, or does not compile at all if the platform does not support highp in fragment shaders. This might raise worries about abandoning potential users with older devices, but the majority of mobile devices that have reasonable WebGL support also fully support highp. This includes recent SOCs from Qualcomm, NVIDIA, and Apple. In particular, all OpenGL ES 3.0 supporting devices fully support highp [ESSL300 §4.5.3], but many older devices also include support for it. In data collected by WebGL Stats in the beginning of December 2014, 85% of mobile clients reported full support for highp in fragment shaders.

Listing 8.1 Example of a shader with precision issues.

```
#ifdef GL_ES
precision mediump float;
#endif

uniform float time;

float rand(vec2 co) {
  return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) * 43758.5453);
}

void main (void) {
  //Divide the coordinates into a grid of squares
  vec2 v = gl_FragCoord.xy /20.0;
  //Calculate a pseudo-random brightness value for each square
  float brightness = fract(rand(floor(v)) + time);
  // Reduce brightness in pixels away from the square center
  brightness * = 0.5 - length(fract(v) - vec2(0.5, 0.5));
  gl_FragColor = vec4(brightness * 4.0, 0.0, 0.0, 1.0);
}
```

In vertex shaders, it is recommended to use highp exclusively, since it is universally supported and vertex shaders are the bottleneck less often than fragment shaders. The better performance that can be attained on a minority of mobile GPUs in particularly vertex-heavy applications usually isn't worth the risk of bugs from using mediump in vertex shaders.

Going the highp route does sacrifice some performance and power usage along with device compatibility. A better solution would be to change the pseudorandom function. In this case, the fix needs to be verified on a platform that supports performing floating-point calculations in a lower precision. One option is using suitable mobile hardware. To find out which kind of hardware you have, see the *shader precision* section of the khronos webgl Information page.*

Another convenient option is to use software emulation of lower precision floats. The ANGLE library implements such emulation by mutating GLSL shaders that are passed to WebGL. When the mutated shaders are run, they produce results that are very close to

* https://www.khronos.org/registry/webgl/sdk/tests/extra/webgl-info.html

what one would see on most mobile devices. The emulation carries a large performance cost and may prevent very complex shaders from being compiled successfully, but is typically fast enough to run content targeting mobile platforms at interactive rates on the desktop. Chrome 41 was the first browser to expose this emulation through a command line flag--`emulate-shader-precision`. You'll find more information on precision emulation on the GitHub repository of WebGL Insights.*

In the case of the example code, simply reducing the last coefficient in the pseudorandom function will make the shader work adequately in mediump precision. The code with the fixes in place is found in Listing 8.2.

Listing 8.2 The example shader with precision-related issues fixed.

```
precision mediump float;

uniform float time;

float rand(vec2 co) {
  return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) * 137.5453);
}

void main (void) {
  //Divide the coordinates into a grid of squares
  vec2 v = gl_FragCoord.xy /20.0;
  //Calculate a pseudo-random brightness value for each square
  float brightness = fract(rand(floor(v)) + time);
  // Reduce brightness in pixels away from the square center
  brightness * = 0.5 - length(fract(v) - vec2(0.5, 0.5));
  gl_FragColor = vec4(brightness * 4.0, 0.0, 0.0, 1.0);
}
```

The OpenGL ES Shading Language spec mandates that if both a vertex shader and a fragment shader reference the same uniform, the precision of the uniform declarations must match. This can sometimes complicate making precision-related fixes, especially if the WebGL shader code is not written directly but rather generated by some tool.

If a project is hit by a compiler error because of uniform precision mismatch, it is best to fix this by making the precision of the two uniform declarations the same by any means available, rather than using two different uniforms with different precision settings to work around the issue. Using two uniforms hinders the maintainability of the code and adds CPU overhead, which is always to be avoided on mobile platforms. Keep this issue in mind when making decisions about shader development tools. Code written in ESSL is much easier to convert to other languages than vice versa, so ESSL is a good choice for a primary shading language.

Tip: Using mediump precision in fragment shaders provides the widest device compatibility, but risks corrupted rendering if the shaders are not properly tested.

Tip: Using only highp precision prevents corrupted rendering at the cost of losing some efficiency and device compatibility. Prefer highp especially in vertex shaders.

* https://github.com/WebGLInsights/WebGLInsights-1

Tip: To test device compatibility of shaders that use mediump or lowp precision, it is possible to use software emulation of lower precision.

Tip: `fract()` is an especially risky function when being run at low precision.

Tip: Remember to define sampler precision when sampling from float textures.

8.2.2 Render Target Support

Attempting to render to an unsupported color buffer format is another common source of bugs when running WebGL content on mobile platforms. Desktop platforms typically support many formats as framebuffer color attachments, but 8-bit RGBA is the only format for which render target support is strictly guaranteed in WebGL [WebGL]. In the mobile world, one may find GLES 2.0 devices that only implement this bare minimum required by the spec. With devices that support GLES 3.0 the situation is improved, and both 8-bit RGBA and RGB are guaranteed to be supported in WebGL [GLES3.0 §3.8.3].

However, one needs to tread even more carefully when using floating-point render targets, which are available in WebGL through `OES_texture_float` and `WEBGL_color_buffer_float`. The `OES_texture_float` extension, which includes the possibility of implicit render target support, is somewhat open to interpretation, and the level of support varies. Some mobile devices do not support rendering to floating-point textures at all, but others are better: some OpenGL ES 3.0 devices can expose rendering to 16-bit float RGBA, 16-bit float RGB, and 32-bit float RGBA, just like browsers on desktop. Note that support for rendering to 32-bit float RGB textures was recently removed from the WebGL specifications altogether to unify the platform, but some browsers on desktop may still include this legacy feature.

The level of render target support on different platforms is summarized in the following table for the most interesting formats. It is organized by the native APIs that WebGL implementations use as backends on different platforms. On older mobile devices made before 2013, the API supported is typically OpenGL ES 2.0. On mobile devices starting from late 2013, the API supported is typically OpenGL ES 3.0 or newer. On Apple computers, starting from some 2007 models, and recent Linux PCs, the API is OpenGL 3.3 or newer, and on Windows, browsers typically implement WebGL using a DirectX 9 or DirectX 11 backend. The Windows render target support data here are from ANGLE, which is the most common backend library for WebGL on Windows.

Format/platform	OpenGL ES 2.0	OpenGL ES 3.0/3.1	OpenGL 3.3/newer	ANGLE DirectX
8-bit RGB	Maybe	Yes	Yes	Yes
8-bit RGBA (1)	Yes	Yes	Yes	Yes
16-bit float RGB	Maybe (3)	Maybe (3)	Maybe	Yes (5)
16-bit float RGBA	Maybe (3)	Maybe (3, 4)	Yes	Yes
32-bit float RGB (2)	No	No (6)	Maybe	Maybe (5)
32-bit float RGBA	No	Maybe (4)	Yes	Yes

Notes:

(1) Guaranteed by WebGL specification, even if it's not guaranteed by GLES2.

(2) Support was removed from `WEBGL_color_buffer_float` recently.

(3) Possible to implement using `EXT_color_buffer_half_float`.

(4) Possible to implement using `EXT_color_buffer_float`.

(5) Implemented using the corresponding RGBA format under the hood.

(6) Possible to implement by using RGBA under the hood, but browsers don't do this.

To find out which texture formats your device is able to use as framebuffer color attachments, use the Khronos WebGL information page.*

Tip: When using an RGB framebuffer, always implement a fallback to RGBA for when RGB is not supported. Use `checkFramebufferStatus`.

8.2.3 Capabilities Exposed by getParameter

The WebGL function `getParameter` can be used to determine capabilities of the underlying graphics stack. It will reveal the maximum texture sizes; how many uniform, varying, and attribute slots are available; and how many texture units can be used in fragment or vertex shaders. The most important thing is that sampling textures in vertex shaders may not be supported at all—the value of `MAX_VERTEX_TEXTURE_IMAGE_UNITS` is 0 on many mobile devices.

The `webgl.min_capability_mode` flag in Firefox can be used to test whether an application is compatible with the minimum capabilities of WebGL. Note that this mode does not use the minimums mandated by the specification, as some of them are set to such low values that they are never encountered in the real world. For example, all devices where WebGL is available support texture sizes at least up to 1024×1024, even if the minimum value in the spec is only 64×64. OpenGL ES 3.0 devices must support textures at least up to 2048×2048 pixels in size.

8.3 Performance

When it comes to performance, one thing in particular sets mobile GPUs apart from their desktop counterparts: the limited main memory bandwidth that is also shared with the SoC's CPU cores [Pranckevičius 11; Merry 12]. This has led Imagination, Qualcomm, and ARM to implement a tiled rendering solution in hardware, which reduces the main memory bandwidth usage by caching a part of the framebuffer on-chip [Merry 12]. However, this approach also has drawbacks: It makes changing the framebuffer or blending state particularly expensive. The NVIDIA Tegra family of processors has a more desktop-like GPU architecture, so there are fewer surprises in store, but as a result the Tegra GPUs may be more sensitive to limited bandwidth in some cases.

In the case of WebGL, the CPU also often becomes a bottleneck on ARM-based mobile devices. Running application logic in JavaScript costs more than if the same thing was implemented in native code, and the validation of WebGL call parameters and data done by the browser also uses up CPU cycles. The situation is made worse by the fact that WebGL work can typically be spread to at most a few CPU cores, and single-core performance is seeing slower growth than parallel-processing performance. The rate of single-core performance gains started slowing down on desktop processors around 10 years ago [Sutter 09], and recently mobile CPUs have started to show this same trend.

Even before the past few years, the performance of mobile GPUs has grown faster than performance of mobile CPUs. To use performance figures Apple has used in marketing the iPhone as an example, their graphics performance has increased nearly 90% per year on average, whereas their CPU performance has grown only by 60% per year on average.

* https://www.khronos.org/registry/webgl/sdk/tests/extra/webgl-info.html

This means that optimizations targeting the CPU have become more important as new hardware generations have arrived.

8.3.1 Getting Started with Performance Improvements

Performance improvements should always start with profiling where the bottlenecks are. For this, we combine multiple tools and approaches. A good starting point is obtaining a JavaScript CPU profile from a target device using browser developer tools. Doing the profiling on a few different devices is also useful, particularly if it is suspected that an application could be CPU limited on some devices and GPU limited on others. A profile measured on desktop doesn't help as much: The CPU/GPU balance is usually very different, and the CPU bottlenecks can be different as well.

If idle time shown in the profile is near zero, the bottleneck is likely to be in JavaScript execution, and it is worthwhile to start optimizing application logic and API usage and seeing whether this has an effect on performance. Application logic is unique to each application, but the same API usage patterns are common among many WebGL applications, so see what to look for in the next section. This is going to be especially beneficial if the CPU profile clearly shows WebGL functions as bottlenecks. See Figure 8.4 for an example. Each of the optimizations detailed in the following sections can sometimes yield more than 10% performance improvements in a CPU-bound application.

The GitHub repository for WebGL Insights contains a CPU-bound drawing test to demonstrate the effect of some of these optimizations. The test artificially stresses the CPU to simulate CPU load from application logic and then issues WebGL draw calls with

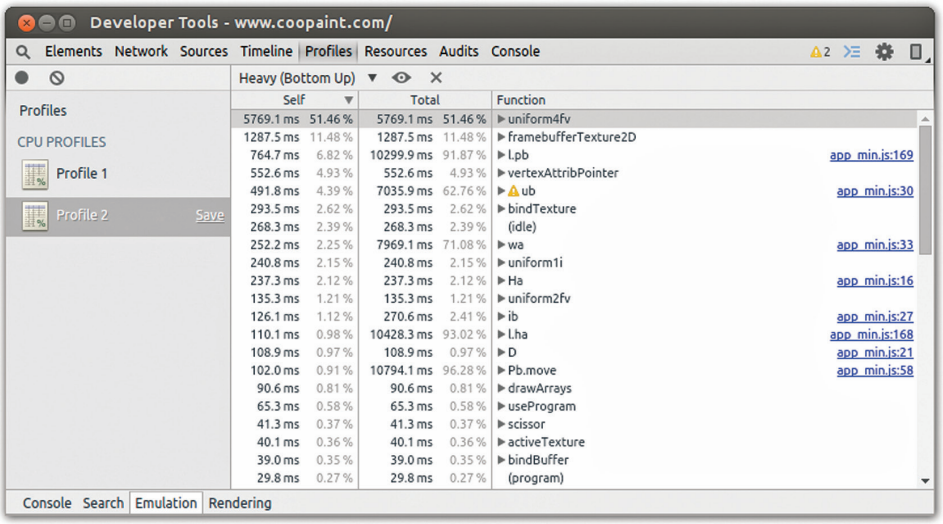

Figure 8.4

Example of a JavaScript CPU profile measured while doing heavy operations in the WebGL painting application CooPaint on a Nexus 5 (2013 phone). Some WebGL functions consume a large portion of CPU time. This application would very likely benefit from API usage optimizations such as trimming the number of `uniform4fv` calls.

options for different optimizations. The test results for each optimization given later in this chapter were measured on a Nexus 5 (2013 phone with Qualcomm Snapdragon 800) and a Shield Tablet (2014 tablet with NVIDIA Tegra K1 32-bit).

If we see plenty of idle time on the CPU executing JavaScript, our application may be GPU bound or main memory bandwidth bound. In this case, we should look for ways to optimize shaders or think of different ways to achieve the desired rendering result. Full-screen effects are usually a good place to start optimizing in this case [McCaffrey 12].

8.3.2 Optimizing API Usage

Every WebGL call has some CPU overhead associated with it. The underlying graphics driver stack needs to validate every command for errors, manage resources, and possibly synchronize data between threads [Hillaire 12]. On top of that, WebGL duplicates a lot of the validation to eliminate incompatibility between different drivers and to ensure stricter security guarantees. On some platforms, there is also overhead from translating WebGL calls to a completely different API, but on mobile platforms this is less of a concern since the underlying API is typically OpenGL ES, which WebGL is derived from. For more details, see Chapter 2.

Since every API call has overhead, performance can often be improved by reducing the number of API calls. The basics are simple enough: An application should not have unnecessary "get" calls of any kind, especially `getError`, or frequent calls requiring synchronization like `readPixels`, `flush`, or `finish`. Drawing a chunk of geometry with WebGL always has two steps: Setting the GL state required for drawing and then calling `drawElements` or `drawArrays`.

Setting the GL state can be broken down further into choosing a shader program by calling `useProgram`, setting flags and uniforms, and binding resources like textures and vertex buffer objects. An optimal system will change the GL state with the minimal number of API calls, rather than redundantly setting parts of the state that don't change for every single draw call. To start with a simple example, if an application needs blending to be disabled for all draw calls, it makes a lot more sense to call `gl.disable(gl.BLEND)` once at the beginning of drawing rather than repeatedly before every single draw call.

For binding resources, there are a few distinct ways to reduce API calls. Often the simplest way is to use vertex array objects, or VAOs, that can significantly improve performance on mobile devices. They are an API construct developed specifically to enable optimization, and are not to be confused with vertex buffer objects, or VBOs. In our CPU-bound drawing test, FPS increase from using VAOs was 10%–13%. For more information, see [Sellers 13].

A VAO encapsulates the state related to bound vertex arrays—that is, the state that's set by `bindBuffer`, `enable/disableVertexAttribArray`, and `vertexAttrib-Pointer`. By using VAOs we can replace calls to the aforementioned functions with a single `bindVertexArrayOES` call. VAOs are available in WebGL 1.0 with the `OES_vertex_array_object` extension, which is widely supported on mobile devices. As of early 2015, more than 80% of smartphone and tablet clients recorded by WebGL Stats have it.*

* http://webglstats.com/

Adding a fallback for devices without VAO support is also straightforward. Let's call the code that binds buffers and sets vertex attrib pointers related to a specific mesh the *binding block*. If VAOs are supported, the code should initialize the VAO of each mesh using the binding block. Then, when the mesh is drawn, the code either binds the VAO if VAOs are supported, or executes the binding block if VAOs are not supported. The only case where this becomes more complicated is when there's a different number of active vertex attribute arrays for different meshes—then the code should add `disableVertexAttribArray` calls where appropriate. For a complete code example, see an explanation of VAOs* or an implementation of a fallback path in SceneJS.†

Lowering the number of vertex buffers helps to reduce CPU usage if VAOs are not a good fit for the code for some reason. This can be done by interleaving different types of vertex data for the same object: If we have, for example, positions, texture coordinates, and normals for each vertex, they can all be stored in the same vertex buffer in an interleaved fashion. In our CPU-bound drawing test that uses four vertex attributes, interleaving the attributes increased the FPS around 4%. The downside is that interleaving the data needs to be either handled by the content creation pipeline or done at load time; the latter may marginally slow down loading. Interleaving three attributes for a million vertices in a tight JS loop had a cost of around 200 ms on a Nexus 5 (2013 phone).

In some special cases, geometry instancing can also be used, but that is typically much harder to integrate into a general-purpose engine than using VAOs, for example. It is only feasible if drawing objects is decoupled enough from the application logic for each object. Yet another way to reduce resource binding overhead is to combine textures of different objects into large texture atlases, so that less texture binding changes are needed. This requires support from the content authoring pipeline and the engine, so implementing it is also quite involved.

Uniform values are a part of the shader program state. If we are using the same shader program to draw multiple objects, it is possible that some of the objects also share some of the same uniform values. In this case, it makes sense to avoid setting the same value to a uniform twice. Caching every single uniform value separately in JavaScript may not be a performance win, but if we can group, for example, lighting-related uniform values together, and only update them when the shader used to draw or the lighting changes, this can yield a large performance improvement. In our CPU-bound drawing test, updating one `vec3` uniform had a cost on the order of 1 microsecond on the mobile devices tested. This is enough to make extra uniform updates add up to a significant performance decrease if they are done in large quantities.

Reducing the number of `useProgram` calls is often also possible by sorting draw calls so that the ones using the same program are grouped together. This also helps to reduce redundant uniform updates further. We may even benefit from using parts of the uniform state as a part of the sort key.

Software-driven deferred rendering is still a long shot on most mobile devices, but using `WEBGL_draw_buffers` brings it a bit closer to within reach. The performance improvement from using `WEBGL_draw_buffers` varies, but it reduces CPU usage, since draw

* http://blog.tojicode.com/2012/10/oesvertexarrayobject-extension.html
† https://github.com/xeolabs/scenejs/blob/v4.0/src/core/display/chunks/geometryChunk.js

calls only need to be issued once, and also reduces vertex shader load and depth buffer accesses. There should be clear benefits at least in complex scenes [Tian 14].

To find out which of these optimizations we might be able to make, it's useful to get a complete picture of our application's API usage by using Chrome's Capture Canvas Frame tool. The tool will list the sequence of WebGL calls done by our application to render a single frame. It helps to see which commands to set GL state are common between multiple draw calls and how many calls are spent doing vertex array setup that might be better accomplished by using VAOs.

Tip: Sort draw calls according to shader program state, and reap the benefits by removing redundant `useProgram` calls and uniform updates.

Tip: Use vertex array objects (VAOs) and interleave static vertex data. This will save a lot of API calls.

8.3.3 Optimizing Shader Execution

When we suspect that our application is shader-bound, we can always perform a simple test to see if this really is the case: Replace all of the shaders with trivial ones that only render a single recognizable color and measure the performance. If the performance is significantly changed, the application is likely shader-bound—either by GPU computation or by texture fetches performed by the shaders.

There are a few different ways to optimize a shader-bound application. Many of the optimizations presented here are discussed in Pranckevičius's excellent SIGGRAPH talk [Pranckevičius 11]. The talk has some more details about micro-optimization, but here we will concentrate on more general guidelines.

If geometry is drawn in a random order, fragment shaders might be run unnecessarily for fragments that eventually get obscured by something else in the scene. This is commonly known as overdraw, and it can be reduced with the help of early z-test hardware built into the GPU. This requires sorting opaque geometry front-to-back, so that the fragments that are obscured by something else get processed last [McCaffrey 12]. The sorting should be done on a relatively coarse level in order to avoid spending too much time on it and moving the bottleneck to the CPU. A too fine-grained depth sort can also increase the number of shader state changes needed to render the scene, so it's often a balancing act between sorting by shader state and sorting by depth. Also see Chapter 10.

The front-to-back sorting helps to improve performance on all but Imagination's PowerVR series of GPUs, which implement deferred fragment shading in hardware [Merry 12]. Also note that if shaders modify the fragment depth value or contain discard statements, the early z-test hardware won't help. Another possible technique is to use a z-prepass, it's typically not worth the overhead [McGuire 13].

Another optimization strategy is to switch to a simpler lighting model. This may mean sacrificing some quality, but there are often alternative ways to do things, such as using look-up textures instead of shader computation. However, since mobile GPUs are typically even more bandwidth-bound than they are computation-bound, this kind of change may not always be a performance win. Recent trends are also toward higher computational performance in mobile GPUs, with bandwidth being the more fundamental limit [McCaffrey 12].

Recent Tegra processors perform especially well in shader computation, so they become texture-bandwidth-bound more easily than other mobile GPUs.

Finally, if we want to squeeze out the last bits of performance, we can try lowering the precision of our shader computations. Just remember to test any shaders that use mediump or lowp carefully for correctness. The performance benefits that can be gained from this vary greatly between platforms: on NVIDIA Tegra K1, precision does not matter at all, and on the Qualcomm Adreno GPU line, precision only makes a small difference, but large differences have been reported on some other mobile GPUs.

8.3.4 Reducing Bandwidth Usage

There are a few different ways to reduce bandwidth usage in a WebGL app, and some of them are WebGL-specific. First of all, we should make sure that extra copies don't happen when the WebGL canvas is composited. Keep the `preserveDrawingBuffer` context creation attribute in its default value false. In some browsers, also having the `alpha` context creation attribute as false may enable more efficient occlusion culling; it communicates to the browser compositor that the canvas element is opaque and does not need to be blended with whatever is underneath it on the page. However, if we need a static background other than a flat color for our WebGL content, it may make sense to render the background using other HTML elements and only render the parts that change on every frame on the main WebGL canvas, keeping `alpha` as true.

The more obvious ways to reduce bandwidth are reducing texture or framebuffer resolution. Reducing texture resolution can sometimes be done without sacrificing too much visual quality, especially if the textures have mostly low-frequency content [McCaffrey 12]. Many mobile device displays have extreme pixel densities these days, so it's worthwhile to consider whether rendering at the native resolution is worth losing performance or reducing the visual quality in other ways. Also see Chapter 14.

Implementing full-screen effects in an efficient way or avoiding them altogether can also enable huge bandwidth savings [McCaffrey 12; Pranckevičius 11]. In particular, it is better to combine different postprocessing filters into a single shader or add simple postprocessing effects directly into the shaders used to render geometry, when possible.

Using lots of small polygons also costs bandwidth on tiler architectures, since they need to access the vertex data separately for each tile [Merry 12]. Optimizing models to minimize the vertex and triangle count helps on these GPUs.

8.3.5 Choosing a WebGL Engine for Mobile

Using a higher level library or engine instead of writing directly against the WebGL API makes development more efficient. This also applies to mobile applications. The downside is that there's some overhead from using the library, and if the library is not a good fit for the content, its effects on performance on mobile platforms may be disastrous. There are other use cases for WebGL besides rendering 3D scenes, but here I'll give a brief overview of libraries that are specifically targeting 3D rendering from the mobile perspective.

three.js is one of the most popular WebGL libraries. However, it's not usually the best fit for developing applications for mobile. three.js offers a very flexible API, but this comes at a cost: Lots of dynamic behavior inside the library adds a whole lot of CPU overhead. At the time of writing, it had only a few of the optimizations detailed in this chapter. It often does redundant or inefficient updates to the GL state, particularly to uniform

values, and due to how it's structured, this is unlikely to improve much without some very significant changes. Still, if we're planning to render only relatively simple scenes with few objects, it's possible to get good performance from three.js, and it allows us to easily use advanced effects using built-in and custom shaders. It does have depth sorting to help with avoiding overdraw. With three.js, it is recommended to use the `BufferGeometry` class instead of the Geometry class to improve performance and memory use.

Some slightly less widely used libraries may be a better fit for your application, depending on what kind of content you're planning to render. Babylon.js (Chapter 9) and Turbulenz (Chapter 10) have both been demonstrated to be able to run fairly complex 3D game content in an ARM SoC environment. Babylon.js has a robust framework for tracking GL state and making only the necessary updates, which saves a lot of CPU time. Turbulenz does some of this as well, and also uses VAOs to improve performance. Be aware of bugs though: Demos using older versions of Babylon.js do not do justice to the newer versions, where lots of issues appearing on mobile platforms have been fixed. Turbulenz still has some open issues that affect mobile specifically, so more work is required to ship a complete product with it.

SceneJS is another production-proven WebGL library, which was heavily optimized in version 4.0. It lacks some of the flexibility of its peers and may not be suited to all types of 3D scenes, but makes up for this by using VAOs and state tracking to optimize rendering. If the application content consists of large amounts of static geometry, SceneJS may be an excellent pick.

Native engines that have been Emscripten-compiled to JavaScript (Chapter 5) are sadly not yet a viable alternative when targeting mobile devices. The memory cost is simply too high on devices with less than 4 GB of memory, and the runtimes have a lot of overhead. Future hardware generations, improvements to the typed array specification, JavaScript engines, and the 3D engines themselves might still change this, but so far better results can almost always be had by using JavaScript directly.

8.4 Resources

See this chapter's materials in the WebGL Insights GitHub repo for additional resources, such as:

- A JavaScript calculator that demonstrates the behavior of low-precision floats
- Tools that emulate lowp and mediump computations on desktop
- A WebGL test that demonstrates CPU optimizations

Acknowledgments

Thanks to Shannon Woods, Florian Bösch, and Dean Jackson for providing useful data for this chapter.

Bibliography

[ESSL100] Robert J. Simpson. "The OpenGL ES Shading Language, Version 1.00.17." Khronos Group, 2009.

[ESSL300] Robert J. Simpson. "The OpenGL ES Shading Language, Version 3.00.4." Khronos Group, 2013.

[GLES3.0] Benj Lipchak. "OpenGL ES, Version 3.0.4." Khronos Group, 2014.
[Hillaire 12] Sébastien Hillaire. "Improving Performance by Reducing Calls to the Driver." In *OpenGL Insights*. Edited by Patrick Cozzi and Christophe Riccio. Boca Raton, FL: CRC Press, 2012.
[McCaffrey 12] Jon McCaffrey. "Exploring Mobile vs. Desktop OpenGL Performance." In *OpenGL Insights*. Edited by Patrick Cozzi and Christophe Riccio. Boca Raton, FL: CRC Press, 2012.
[McGuire 13] Morgan McGuire. "Z-Prepass Considered Irrelevant." Casual Effects Blog. http://casual-effects.blogspot.fi/2013/08/z-prepass-considered-irrelevant.html, 2013.
[Merry 12] Bruce Merry. "Performance Tuning for Tile-Based Architectures." In *OpenGL Insights*. Edited by Patrick Cozzi and Christophe Riccio. Boca Raton, FL: CRC Press, 2012.
[Olson 13] "Benchmarking Floating Point Precision in Mobile GPUs." http://community.arm.com/groups/arm-mali-graphics/blog/2013/05/29/benchmarking-floating-point-precision-in-mobile-gpus, 2013.
[Pranckevičius 11] Aras Pranckevičius. "Fast Mobile Shaders." http://aras-p.info/blog/2011/08/17/fast-mobile-shaders-or-i-did-a-talk-at-siggraph/, 2011.
[Resig 13] John Resig. "ASM.js: The JavaScript Compile Target." http://ejohn.org/blog/asmjs-javascript-compile-target/, 2013.
[Sellers 13] Graham Sellers. "Vertex Array Performance." OpenGL SuperBible Blog. http://www.openglsuperbible.com/2013/12/09/vertex-array-performance/, 2013.
[Sutter 09] Herb Sutter. "The Free Lunch Is Over: A Fundamental Turn toward Concurrency in Software." Originally appeared in *Dr. Dobb's Journal* 30(3), 2005. http://www.gotw.ca/publications/concurrency-ddj.htm, 2009.
[Tian 14] Sijie Tian, Yuqin Shao, and Patrick Cozzi. "WebGL Deferred Shading." Mozilla Hacks Blog. https://hacks.mozilla. org/2014/01/webgl-deferred-shading/, 2014.
[WebGL] Khronos WebGL Working Group. "WebGL Specification." https://www.khronos.org/r egistry/webgl/specs/latest/1.0/, 2014.