OpenGL Insights

Edited by
Patrick Cozzi and Christophe Riccio
36.1 Introduction

Understanding performance bottlenecks in games helps developers deliver the best gameplay experience. In games, performance bottlenecks are usually grouped in one of two categories: CPU or GPU. Focusing optimization efforts in the appropriate category saves development time and helps get our game running better faster. Optimizing CPU issues when the bottleneck is in the graphics pipeline will result in little to no performance gain and a good amount of frustration. Deploying our game in a web browser complicates the process of isolating bottlenecks. Using the techniques described here, we’ll be more successful in identifying the most profitable areas to optimize.

Postprocessing effects have become a standard in AAA games and very often are a performance bottleneck. In this article, we discuss an implementation of the bloom effect in WebGL and its performance characteristics in the browser. Because WebGL runs in a web browser, this poses some special challenges when doing graphics analysis in comparison to a native 3D graphics application. Just as a native application may choose a different code path when detecting a different operating system, the same is true of the browser’s implementation of the Canvas or WebGL APIs. Add to this the fact that we’ll likely be supporting multiple browsers, and there is the potential for many permutations and a challenge in understanding what happens from “script to pixel.” We’ll discuss the support for WebGL analysis in common 3D graphics tools and the various implementations of the standard in modern web browsers.
36.2 The Stages of Bloom

The familiarity of the bloom effect in games is one reason it was used for this article. The other major reason is that it is composed of several steps with parameters that can be tweaked to favor quality versus performance. As shown in Figure 36.1, this implementation starts with the original scene rendered to a texture and applies the bloom effect in four major steps:

1. Draw scene to a texture.
2. Identify fragments whose luminance exceeds a threshold.
3. Blur the results of the luminance test.
4. Combine the original rendered scene texture with the blurred highlights.

Each of these steps has parameters that can trade quality for performance. In the luminance step, we can set the luminance threshold to control the number of fragments of the original scene texture that are written to the luminance render target. In the blur step, we can set the number of blur passes and resolution of the render target.

![Figure 36.1. Visual representation of the results of each stage.](image)

---

**Figure 36.1.** Visual representation of the results of each stage.
target to increase/decrease blur quality. In the final step, we can control the weight of the blurred highlights that get combined with the original scene.

Fragments of the original scene that have a luminance value above the luminance threshold are written to the render target. Anything below the luminance threshold is written as black. The number of blur passes determines the number of times the highlights (luminance results) are blurred. The resolution determines the size of the render target used by the blur passes. The weight of the blurred highlights determines how much of the blurred highlights end up in the final frame. We expose some of these parameters as part of the HUD and others are set in code.

The source code that accompanies this chapter is laid out in a simple format to make it easy to follow and understand. The bloom implementation is composed of the following:

- **MainLoop** (in index.html) takes care of calling update/render loop with the appropriate request-animation frame method for each browser.
- **Init** (in bloom.js) defines all resources used in the sample, such as shaders, textures, scene objects geometry, and render targets.
- **Update** (in bloom.js) contains all nonrendering actions such as updating rotations.
- **Render** (in bloom.js) draws scene geometry, performs luminance test, blurs highlights, and combines results into the final frame.
- **bloom-utils.js** contains helper functions used to load shaders and textures, parse .obj files, and create geometry.

### 36.3 Overhead of Bloom

Now that we’ve described the general implementation of bloom as a postprocessing effect, we’ll describe the specifics about our implementation in WebGL. The first thing we measure is the actual overhead of applying bloom to the scene. With the Javascript code in Listing 36.1, we capture a good enough approximation of the frame time to measure overhead and update the scene.

```javascript
var MainLoop = function () {
    nCurrentTime = ( newDate ).getTime();
    fElapsedTime = nCurrentTime - nLastTime;
    nLastTime = nCurrentTime;
    // call Update & Render
    // call requestAnimationFrame( MainLoop );
};
```

**Listing 36.1.** JavaScript code to approximate frame time.
This will measure the time between `requestAnimationFrame` callbacks. Some web browsers may expose performance data at runtime when enabled. For example, running Google Chrome with the `--show-fps-counter` flag displays a frames-per-second counter. With this measurement code in place, introducing bloom approximately doubles our frame time (see Figure 36.2).

The measurements were taken on Google Chrome version 15.0.874.106 running on a prerelease second generation Intel Core processor (Intel microarchitecture code name Sandy Bridge, D1 stepping quad core 2.4 GHz CPU with 4GB DDR3 1333MHz RAM) with Intel HD Graphics 3000 running Windows 7 Ultimate with Service Pack 1. The frame time is composed of the amount of time it takes API calls to set state on the CPU, and the time it takes the GPU to process the draw calls. The JavaScript code above suffices to measure time spent on the CPU. To understand GPU frame time, we’ll refer to some offline tools discussed later in this article.

### 36.4 Analyzing WebGL Applications

Analyzing WebGL applications poses a few interesting challenges because there are many moving parts that have to work together: operating systems, graphics APIs, graphics drivers, browsers, and analysis tools.

#### 36.4.1 Almost Native Graphics Layer (ANGLE)

One of the main challenges when doing analysis on a WebGL application is to understand the difference between running on Windows, Mac OS X, or Linux. On Windows, OpenGL drivers can usually be downloaded from the graphics hardware vendor’s website when available. On Mac OS X, OpenGL drivers are part of the
system and are updated through the OS update mechanism. On Linux, OpenGL drivers might not be installed by default, but are generally provided through the distribution's package management system or the hardware vendor's website.

For the broadest compatibility on the Windows platform, Chrome and Firefox make use of the Almost Native Graphics Layer Engine [ANGLE 11]. This layer translates OpenGL ES 2.0 calls to DirectX 9 API calls, and translates GLSL shaders to equivalent HLSL shaders. As a user, this translation is completely hidden, but as a developer, this layer is as important as the WebGL application we wrote. ANGLE has a few quirks related to differences in the APIs specifically with buffers and texture fetches. For example, ANGLE does not create/update resources until a draw call is issued, as explained in Chapter 39.

36.4.2 JavaScript profiling

Most modern web browsers have a set of JavaScript developer tools that are prepackaged or can be installed from an extension (see, for example, Figure 36.3). Chrome, Firefox, Internet Explorer, Opera, and Safari have their own JavaScript debuggers

![Chrome developer tools and Firebug.](image)
and profilers. These help with debugging HTML DOM and network latency issues. JavaScript profilers are helpful in understanding where CPU time is spent. However, these tools don’t show contextual information for WebGL beyond the JavaScript API calls.

### 36.4.3 WebGL Inspector

The other major issue with analyzing a WebGL application is the limited support of tools. WebGL Inspector [Vanik 11] is currently the de facto tool for debugging API calls and understanding bound resources. This tool can capture a frame and show the API calls; state; and bound textures, buffers, and programs. It is available as a Google Chrome extension and as a JavaScript library that can be dropped into our WebGL application—useful when running on browsers other than Chrome. WebGL Inspector, shown in Figure 36.4, is free and available for download from http://benvanik.github.com/WebGL-Inspector/.

![WebGL Inspector captured frame](image)

**Figure 36.4.** WebGL Inspector showing a frame capture of our sample.

### 36.4.4 Intel Graphics Performance Analyzers (GPA)

A positive side effect of Chrome and Firefox using ANGLE on Windows is that DirectX analysis tools can be used to analyze WebGL applications. In this article, we use Intel GPA Frame Analyzer [Intel 11] to capture frames and analyze the post-translation DirectX draw calls and resources. This article shows frame captures from Intel HD Graphics 3000, but Intel GPA is not restricted to Intel graphics hardware.
Figure 36.5 shows a captured frame of the bloom application described above. You can download Intel GPA for free from http://www.intel.com/software/gpa. Refer to the documentation on the Intel GPA website and the documentation that installs this tool for detailed instructions on capturing frames.

36.5 Analysis Workflow on Windows

In this section, we will learn how to use WebGL Inspector and Intel GPA Frame Analyzer to identify problem areas and/or confirm that our program is doing what we think it is doing. On Windows, WebGL Inspector and Frame Analyzer together show the full graphics pipeline when the browser uses ANGLE. WebGL Inspector shows the WebGL side, and Frame Analyzer shows the post-translation DirectX equivalent. WebGL Inspector works well for tracking down incorrectly bound resources and debugging our graphics code.

Once the WebGL Inspector extension is installed and enabled, or we include the JavaScript library in our project, we should see a “capture” button on the top right. With that said, the first step is to capture a frame with WebGL Inspector and make sure we are binding the correct buffers, shaders, and textures. Figure 36.6 shows the “Programs” tab where all shaders used by the WebGL application are displayed as well as the status, uniform, and attribute information. This tab will also display shader
compilation and link errors. The other tabs in WebGL Inspector show detailed information about the other resources such as buffer contents for bound buffers and texture resolution for all bound textures. WebGL Inspector also shows previews for resources such as vertex buffers and textures. The previews can help as a sanity check to make sure the correct mesh or texture is bound when making a draw call.

Unlike WebGL Inspector, Intel GPA is not integrated into the web browser through an extension or JavaScript library. In addition, capturing a frame gets a bit more interesting because of the multiprocess architecture of some browsers like Google Chrome. Intel GPA can attach to the Chrome process on launch, but the process that handles the rendering calls is a child process of the main Chrome process. Fortunately, starting Chrome with a `--no-sandbox` flag allows GPA to attach to the correct rendering process and trigger frame captures. Note that running Chrome with a `--no-sandbox` flag will not change performance characteristics but will change the security characteristics of the browser. For this reason, this flag should never be used for general browsing.

### 36.5.1 Tracking Down API Calls

After capturing a frame and opening it with Frame Analyzer, we will see a visualization of all draw calls in the captured frame, as shown in Figure 36.5. Each Draw, Clear, and StretchRect call is shown as a bar whose height is by default set to GPU duration. At first glance, this visualization shows the order in which geometry is drawn as well as which calls are most expensive. Draw calls are blue bars, Clear calls are light blue bars, and StretchRect calls are dark red/magenta bars. The light gray bars are markers for render target changes. Draw/Clear/StretchRect calls in between two light gray bars affect the same render target. The labels in Figure 36.7 are not a feature of Frame Analyzer but were added for clarity.

Looking at Figure 36.7, we can see that the tall bars correspond to the blur passes, which is expected since that fragment shader is the bulk of the work in this
application. Looking closer at the frame, we can see where the scene is drawn, where
the luminance test happens, the blur passes, and the final composition of the frame.
It is also clear from Figure 36.7 that there are more draw calls in the visualization
than what the WebGL Inspector API log shows. If we look at the calls in between the
luminance test and the first blur pass, we will notice that they seem to be redrawing
the luminance results but using a lower-resolution render target. Comparing this
to the API log from WebGL Inspector, we notice that the only thing happening
between the \texttt{gl.drawArrays} call and the beginning of the blur pass marked by
\texttt{gl.bindFramebuffer} is this piece of code:

\begin{verbatim}
gl.bindTexture(gl.TEXTURE_2D, RenderTargets.HighPass.Texture);
gl.generateMipmap(gl.TEXTURE_2D);
\end{verbatim}

There aren't any noticeable draw calls in that piece of code. But in Windows,
\texttt{gl.generateMipmap(gl.TEXTURE_2D)} is translated to multiple draw calls by
ANGLE. A quick peek at the ANGLE source code (src/libGLESv2/Texture.cpp)
[ANGLE 11] that translates \texttt{generateMipmap} to DirectX 9 shows the following:

\begin{verbatim}
// ...snip
for (unsigned int i = 1; i <= q; i++) {
  IDirect3DSurface9 *upper = NULL;
  IDirect3DSurface9 *lower = NULL;
  mTexture->GetSurfaceLevel(i-1, &upper);
  mTexture->GetSurfaceLevel(i, &lower);
  if (upper != NULL && lower != NULL) {
    getBlitter()->boxFilter(upper, lower);
    if (upper != NULL) upper->Release();
    if (lower != NULL) lower->Release();
    mImageArray[i].dirty = false;
  }
// ...snip
\end{verbatim}
In short, `getBlitter()->boxFilter( upper, lower )` results in a draw call and because it’s in a loop, it’s called multiple times, creating all the extra draw calls we see in Figure 36.7 between the different phases. Since it’s creating all the mipmaps for the previous draw based on the resolution of the render target used, reducing the initial render target resolution will not only reduce the work that each pass needs to do, but it will also reduce the number of mipmaps created.

Looking at Figure 36.7, we can see that each labeled region begins with a Clear (light blue), followed by one or more Draw (blue) calls, and ends with a StretchRect (dark red). Like the name suggests, StretchRect will stretch the results to the bound render target to fit the viewport. In some cases, it might be an undesirable effect, but it mostly works well to fill the viewport with our scene. Unfortunately, this results in another hidden call that is unaccounted for compared to the API log in WebGL Inspector.

### 36.6 Optimized Bloom

Now that we understand how to analyze the graphics side of our sample with WebGL Inspector and Intel GPA, we can begin using that information to make changes to our code where it will have the most impact. As clearly shown in Figure 36.8, the blur passes are the bottleneck in our bloom implementation. Using Intel GPA Frame Analyzer, we see that these two calls make up approximately 63% of the frame time.

![Figure 36.8. The tallest and thickest bars are the blur calls.](image)

#### 36.6.1 Lower Render Target Resolution

In our implementation, we have exposed two parameters we can tweak for the blur: number of passes and render-target resolution. From Figure 36.8, we can see that there are only two blur passes, which is fairly low and gives us good quality. Lowering the resolution of the render target we use for the blur passes will have two effects: reducing the number of fragments processed and the number of extra draw...
calls caused by `gl.generateMipmap`, as discussed above. After lowering the resolution to one quarter of the original resolution, we notice that the two blur passes are now only approximately 11% of rendering. That is a significant performance improvement, as shown in Figure 36.9, with an easy code change.

Looking at Figure 36.10, it’s hard to tell the difference by just looking at the final frames, even in WebGL Inspector, since the quality was not noticeably degraded.

![Figure 36.9. Performance impact of lowering resolution of blur render target (frame time in this graph refers to GPU frame time as reported by Intel GPA Frame Analyzer).](image1)

![Figure 36.10. Original final frame with 1024 × 1024 blur render target and after lowering the resolution of the blur render target to 256 × 256.](image2)
However, we can confirm the improvement in Intel GPA by capturing a new frame, as shown in Figure 36.11.

We could go with an even lower resolution, but there is a point where the quality might be affected. In this case, lowering the resolution works well and produces acceptable results because we are working with a blur. For other applications, lowering the resolution might not be the solution.

### 36.6.2 Unnecessary Mipmap Generation

As discussed in Section 36.5.1, there was a call to `generateMipmap` after every bloom stage. In Windows, this resulted in several more API calls than we could account for in the WebGL Inspector log and the code. Originally, we were planning to map the render target textures to quads and display them all on screen to show the bloom stages. We discarded that idea and instead we map each bloom stage to a fullscreen quad. The bloom stages’ results can be displayed one at a time. This allowed us to remove the call to `generateMipmap` and thus remove all the extra API calls. This can be confirmed by comparing Figures 36.7 and 36.12.
36.6.3 Floating-Point Framebuffers

After removing the call to `generateMipmap` in between bloom stages, we looked into the `OES_texture_float` extension to enable floating-point buffers. Originally, we used `gl.UNSIGNED_BYTE` as the format for the framebuffer, which created A8R8G8B8 framebuffers. With the `OES_texture_float` extension enabled, we create floating-point buffers by passing `gl.FLOAT` as the texture format. This creates A32R32G32B32F framebuffers. After lowering the resolution and removing unnecessary mipmap generation, it allows us to create higher-quality blur for approximately the same cost. The code change happened in our `MakeRenderTarget` function (see Listing 36.2).

```javascript
var MakeRenderTarget = function ( gl, nWidth, nHeight ) {
    // create the new framebuffer
    // use floating point framebuffers if OES_texture_float extension exists
    var nTexFormat = ( glgetExtension( "OES_texture_float" ))? gl.FLOAT : gl.UNSIGNED_BYTE;

    // create the offscreen texture
    var pTexture = gl.createTexture();
    gl.bindTexture( gl.TEXTURE_2D, pTexture );
    gl.texParameteri( gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST );
    gl.texParameteri( gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST );
    gl.texParameteri( gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE );
    gl.texParameteri( gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE );
    gl.texImage2D( gl.TEXTURE_2D, 0, gl.RGBA, pFrameBuffer.width, pFrameBuffer.height, 0, gl.RGBA, nTexFormat, null );

    // create the offscreen depth buffer
    // attach texture and depth buffer to framebuffer
    // reset bindings to defaults
    return { "FrameBuffer" : pFrameBuffer, "Texture" : pTexture, "Depth" : pDepthBuffer, "Width" : nWidth, "Height" : nHeight }
};
```

Listing 36.2. Creating floating-point frame buffers with `OES_texture_float`.

According to [Lipchak 05], it requires `NEAREST` magnification filter and `NEAREST` and `NEAREST_MIPMAP_NEAREST` minification filters to be supported. For the bloom sample, we draw these textures in a way that does not need the minification filter, so we set both to `gl.NEAREST`. 
36.7 Conclusion

Support for WebGL is progressing at a steady pace and is helping the browser become a viable development and distribution platform for games with higher-quality graphics. Like any other platform, getting the best performance allows our games to shine and improve the gameplay experience. Tools play an important role in helping game and graphics developers deliver these experiences in the browser. In this article, we presented several tools that work well with WebGL applications and explained some of the areas where potential bottlenecks might appear in the current implementations on Windows. In short, developers should understand the differences between hardware platforms and operating systems to get the best performance. The web browser has always been a way to abstract away the hardware and OS, but with WebGL we’re getting closer to these layers and can now use that to our advantage.

Bibliography


