WebGL Insights

Edited by Patrick Cozzi

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

Section II
Moving to WebGL

Like many WebGL developers, I'm happy to say that WebGL brought me to the web. I developed with C++ and OpenGL for years, but the lure of being able write 3D apps that run without a plugin across desktop and mobile was too great, and I quickly moved to WebGL when the specification was ratified in 2011. In this section, developers, researchers, and educators share their stories on why and how to move to WebGL. We also see similar themes throughout this book.

When our team at AGI wanted to move from C++/OpenGL to JavaScript/WebGL, we weren't sure how well a large JavaScript codebase would scale. We were coming from a C++ codebase that is now seven million lines of code. Could we manage something even 1% of that size? Thankfully, the answer was a definite yes; today, our engine, Cesium, is more than 150,000 lines of JavaScript, HTML, and CSS. In Chapter 4, "Getting Serious with JavaScript," my collaborators, Matthew Amato and Kevin Ring, go into the details. They focus on modular design, performance, and testing. For modularity, they survey the asynchronous module definition (AMD) pattern, RequireJS, CommonJS, and ECMAScript 6. Performance topics include object creation, memory allocation, and efficiently passing data to and from Web Workers. Finally, they look at testing with Jasmine and Karma, including unit tests that call WebGL and aim to produce reliable results on a variety of browsers, platforms, and devices.

Many developers, myself included, have written new engines for WebGL. However, many companies with large, established graphics and game engines may not want to rewrite their runtime engine. They want to reuse their existing content pipeline and design tools, and simply target the web as another runtime. For this, Mozilla introduced Emscripten, which translates C/C++ to a fast subset of JavaScript called asm.js that Firefox can optimize. In Chapter 5, "Emscripten and WebGL," Nick Desaulniers from Mozilla explains how to use Emscripten, including a strategy for porting OpenGL ES 2.0 to WebGL, a discussion of handling third-party code, and a tour of the developer tools in Firefox.

When moving a codebase to WebGL, there are two extremes: Write a new codebase in JavaScript or translate the existing one. Between these extremes is a middle ground: hybrid client-server rendering, where the existing codebase generates commands or images on the server. In Chapter 6, "Data Visualization with WebGL: From Python to JavaScript," Cyrille Rossant and Almar Klein explain the design of VisPy, a Python data visualization

library for scatter plots, graphics, 3D surfaces, etc. It has a layered design from low-level, OpenGL-oriented interfaces to high-level, data-oriented ones, with a simple declarative programming language, GL Intermediate Representation (GLIR). GLIR allows for visualization in pure Python apps as well as JavaScript apps. A Python server generates GLIR commands to be rendered with WebGL in the browser, in a closed-loop or open-loop fashion.

WebGL adoption goes beyond practitioners, hobbyists, and researchers. Given its low barrier to entry and cross-platform support, WebGL is finding itself a prominent part of computer graphics education. Edward Angel and Dave Shreiner are at the forefront of this movement, moving both their introductory book, *Interactive Computer Graphics: A Top-Down Approach,* and the SIGGRAPH course from OpenGL to WebGL. Ed is the person who motivated me to use WebGL in my teaching at the University of Pennsylvania. In 2011, WebGL was a special topic in my course; now, it is *the* topic. In Chapter 7, "Teaching an Introductory Computer Graphics Course with WebGL," Ed and Dave explain the why and the how of moving a graphics course from desktop OpenGL to WebGL, including walking through the HTML and JavaScript for a simple WebGL app.

4

Getting Serious with JavaScript

Matthew Amato and Kevin Ring

4.1 Introduction

As we will see in Chapter 7, "Teaching an Introductory Computer Graphics Course with WebGL," the nature of JavaScript and WebGL makes it an excellent learning platform for computer graphics. Others have argued that the general accessibility and quality of the toolchain also make it great for graphics research [Cozzi 14]. In this chapter, we discuss what we feel is the most important use for JavaScript and WebGL: writing and maintaining real-world browser-based applications and libraries.

Most of our knowledge of JavaScript and WebGL comes from our experiences in helping to create and maintain Cesium,* an open-source WebGL-based engine for 3D globes and 2D maps (Figure 4.1). Before Cesium, we were traditional desktop software developers working in C++, C#, and Java. Like many others, the introduction of WebGL unexpectedly drew us into the world of web development.

Since its release in 2012, the Cesium code base has grown to over 150,000 lines of JavaScript, HTML, and GLSL; has enjoyed contributions from dozens of developers; and has been deployed to millions of end users. While maintaining any large code base presents challenges, maintaining a large code base in JavaScript is even harder.

* http://cesiumjs.org

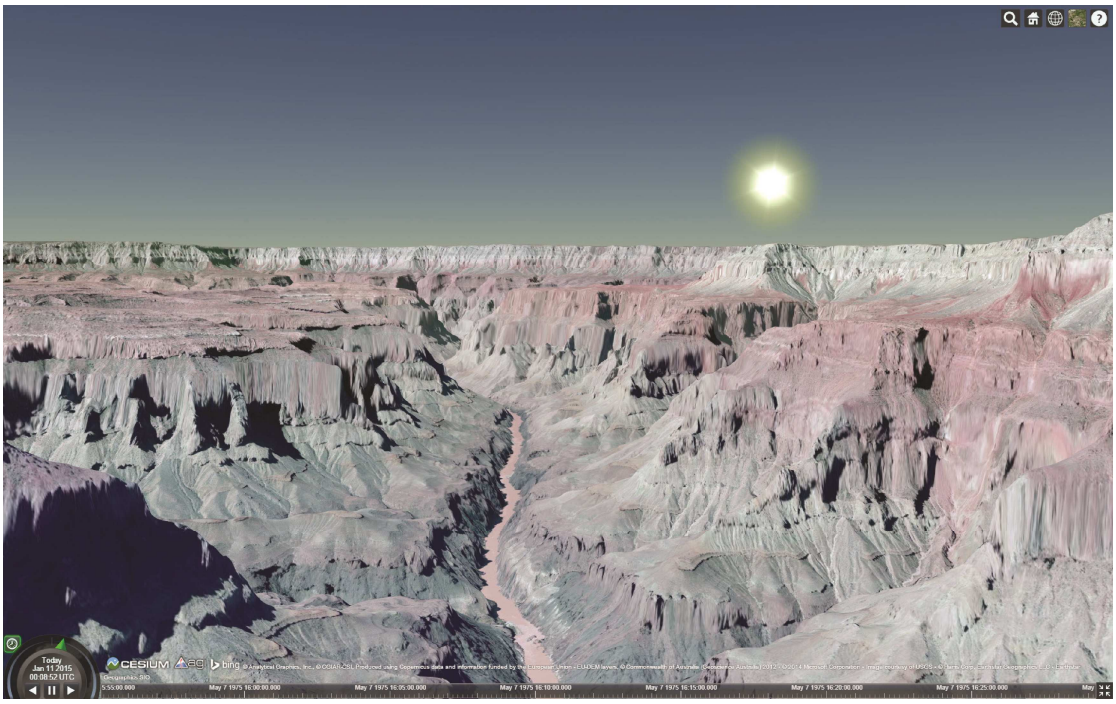

Figure 4.1

Watching the sun set over the Grand Canyon in Cesium.

In this chapter, we discuss our experiences with these challenges, and our strategies for solving or mitigating them. We hope to provide a good starting point for anyone developing a large-scale application for the browser in JavaScript, or in a closely related language like CoffeeScript.

First, the lack of a built-in module system means there's no one right way to organize our code. The common approaches used in smaller applications will become extremely painful as the application grows. We discuss solutions for modularization in Section 4.2.

Second, a lot of the features and flexibility that make JavaScript approachable and easy to use also make it easy to write nonperformant code. Different browser engines optimize for different use cases, so what is fast in one browser might not be fast in another. This is especially concerning for us as WebGL developers, because interactive, real-time graphics often have the highest performance requirements of any application on the web. We give some tips and techniques for writing performant JavaScript code in Section 4.3.

Finally, dynamically typed languages like JavaScript make automated testing even more important than usual. With JavaScript's lack of a compilation step and its dynamic, runtime-resolved symbol references, even basic refactorings are unnerving without a robust suite of tests that can be run quickly to ensure that the application still works. A good approach to testing is essential to building a large-scale application and enabling it to evolve over time. We discuss strategies for testing a large JavaScript application, especially one that uses WebGL, in Section 4.4.

4.2 Modularization

Small JavaScript applications often start their lives as a single JavaScript source file, included in the HTML page with a simple `<script>` tag. The source file defines the functions and types that the application needs in the global scope. As the application grows, maybe we'll add another source file, then another, until soon enough we find ourselves with hundreds of source files and script tags.

Of course, most developers recognize the problems with this approach well before their application gets to hundreds of source files. Some of the problems are

- **Order dependency:** The `<script>` tags for the source files must be included in the HTML page in the proper order. If one file uses a symbol defined by another file before the second file has actually been loaded, the first file will see an undefined reference and throw an exception. This is especially painful in an application consisting of many HTML files, because this properly ordered list of script tags must be maintained in many places.
- **Global scope pollution:** All functions and types are added to the global scope, and each file goes to the global scope to find its dependencies. If another library uses the same names for functions and types as our application code, one or the other will fail.
- **Lack of encapsulation:** There is no obvious place to keep private details of our functions and types, such as internal helper functions.
- **Poor performance:** Loading a large number of individual JavaScript files is slow. This might be OK during development, when we're loading our code from a local web server, but the performance is terrible when the client and web server are on opposite sides of the country or world.

There are various workarounds to these problems. For example, we can avoid the poor performance and order dependency of loading many JavaScript files by creating a build step to concatenate all of our source files together prior to deployment. Of course, that build step still needs to concatenate the source files in the right order!

Early on in Cesium's life, we decided to address all of these problems by using the asynchronous module definition (AMD) pattern and RequireJS.*

4.2.1 Asynchronous Module Definition (AMD)

AMD is a way of structuring JavaScript modules such that they

- Explicitly state what other modules they depend upon
- Are not loaded until all of their dependencies have been loaded first
- Don't touch the global scope

A module is a small unit of functionality in our application, such as a single function or a single class. As an example, here is a slightly modified version of the `Ray` AMD module

* http://requirejs.org/

from Cesium. A `Ray` consists of an origin and direction in 3D space, and it can compute the point a given distance along the ray:

Listing 4.1 A simple asynchronous module definition.

```
define([
        './Cartesian3'
    ], function(
        Cartesian3) {
    "use strict";

  var Ray = function(origin, direction) {
    this.origin = origin;
    this.direction = direction;
  };

  Ray.getPoint = function(ray, t) {
    var offset = Cartesian3.multiplyByScalar(ray.direction, t);
    return Cartesian3.add(ray.origin, offset);
  };

  return Ray;
});
```

In the AMD pattern, our code is placed inside a function that is passed as a parameter to the `define` function. This "module" function gives us a place to store implementation details if desired. JavaScript's function-level scoping guarantees that anything defined inside this function will not be visible outside it unless we explicitly allow it to escape.

Our module doesn't touch the global scope at all. Instead of pulling its dependencies, such as `Cartesian3`, from the global scope, our `Ray` module expects those dependencies to be passed as parameters to the module function. The array passed as the first parameter to `define` specifies which modules—just `Cartesian3` in this case—this module needs to have passed as parameters to its module function. Similarly, our module's export—the `Ray` constructor function—is never assigned anywhere in the global scope. Instead it is just returned to the caller.

But who is the caller? It's the AMD module loader.

The `define` function registers a module with a list of dependencies. Each of the dependencies is a module itself, usually contained in a single JavaScript source file with the same name as the module. Sometime later, when all the dependencies have been loaded, the module function is invoked. The module function returns the module back to the loader, and the loader can then load any other modules that depend on it. With knowledge of all modules and their dependencies, the loader can ensure that they are loaded in the correct order and that only the modules necessary for a given task are ever loaded.

This is where the "asynchronous" in asynchronous module definition comes from: Modules are not created immediately upon execution of the JavaScript files that contain them. Instead, they are created asynchronously as their dependencies are loaded.

With AMD, writing a web page that uses the modules is easy and doesn't require a build step. Typically, the HTML just references the main script using the RequireJS *data-main* attribute:

```
<script data-main = "scripts/main" src = "scripts/require.js"></script>
```

scripts/main.js is itself an AMD module that explicitly specifies its dependencies:

Listing 4.2 An entry point AMD module with three dependencies.

```
require(['a', 'b', 'c'], function(a, b, c) {
     a(b(), c());
});
```

When RequireJS sees the *data-main* attribute, it attempts to load the specified module. Loading that module requires all of its dependencies, *a*, *b*, and *c*, to be loaded first. Attempting to load those modules will, in turn, cause their dependencies to be loaded first. This process continues recursively. Once *a*, *b*, and *c* and all of their dependencies are loaded, *main*'s module function is called and the app is up and running.

With AMD, we get quick iteration, because no build is necessary; just reload the page! There's no need to manage an ordered list of `<script>` tags in each HTML page; we simply specify the entry point and RequireJS takes care of the rest. We also get ease of debugging during development, because the browser sees individual source files exactly as we've written them. This was especially important before browsers had good support for source maps.

What about deployment?

Loading all of those individual modules as separate JavaScript files can take a little while, especially over a high-latency network connection. Fortunately, the r.js optimizer* makes it easy to build and minify all of our modules, creating a single JavaScript source file with all the code our application requires, and no more. If our application uses libraries that are built with AMD, it's even possible to build the application and these libraries together, ensuring that only the parts of the libraries that we actually use are included in our application.

Assuming our application has a single script as its *data-main*, as shown before, we can build a combined and minified version of the application by running the following from the *scripts* directory:

```
r.js-o name=main out=../build/main.js
```

Then, we simply change the *data-main* attribute to point to the built version:

```
<script data-main = "build/main" src = "scripts/require.js"></script>
```

RequireJS has a dizzying array of options, allowing us to control how module names are resolved, to specify paths to third-party libraries, to use different minifiers, and much more. RequireJS also has a large assortment of loader plugins.† One that is especially useful

* https://github.com/jrburke/r.js
† https://github.com/jrburke/requirejs/wiki/Plugins

in WebGL applications is the *text* plugin, which makes it easy to load a GLSL file into a JavaScript string in order to pass it to the WebGL API. All the details can be found on the RequireJS website.

4.2.2 AMD Alternatives

The Cesium team has had great success with AMD, and has found RequireJS to be a robust and flexible tool. We don't hesitate to recommend its use in any serious application. However, there are some valid criticisms of AMD, most of which boil down to a distaste for its fundamental design goal: to create a module format that can be loaded in web browsers without a build step and without any preprocessing.

To that end, AMD adopts a syntax for defining dependencies that is considered by many to be ugly and cumbersome. In particular, it requires us to maintain two parallel lists at the top of each module definition and to keep them perfectly in sync: the list of required modules and the list of parameters to the module creation function. If we accidentally let these lists get out of sync, perhaps by deleting a dependency from one list but forgetting to do so from the other, our parameter named `Cartesian3` might actually be our `Matrix4` module, which would certainly lead to unexpected behavior when we try to use it.

If we accept a build step, perhaps because our code needs to be built for other reasons *anyway*, there are better options than AMD for defining modules that are easy to read and write. After all, today's web browsers support source maps, so debugging transformed code, or even combined and minified code, can look and feel just like debugging the code we actually wrote by hand. By working incrementally, a build process can often be fast enough that it will be finished before we can switch back to our browser window and hit refresh.

If we can use a simpler module pattern and make our development environment more like our production environment in the process, without sacrificing debuggability or iteration time, that's a big win. With that in mind, let's briefly survey some of the more promising alternatives to AMD.

4.2.3 CommonJS

The most popular direct alternative to AMD is the CommonJS* module format. CommonJS modules are not explicitly wrapped inside a function. The module-private scope is implied within each source file rather than expressed explicitly as a function. They also express their dependencies using a syntax that's a bit nicer and more difficult to get wrong:

Listing 4.3 Importing modules using CommonJS syntax.

```
var Cartesian3 = require('./Cartesian3');
var defaultValue = require('./defaultValue');
```

CommonJS is the module format used on the server for Node.js† modules. In Node.js, each of those calls to *require* loads a file off the local disk, so it is reasonable that it not

* http://wiki.commonjs.org/wiki/Modules/1.1
† http://nodejs.org/

return until the file is loaded and the module created. In the high-latency world of the browser, however, synchronous *require* calls would be much too slow.

So, to use CommonJS modules in the browser, we must convert these modules into a browser-friendly form. One approach is to transform them to AMD modules before loading them in the browser. The *r.js* tool we used earlier to create minified builds can also be used to transform CommonJS modules.

Another tool that is gaining traction, especially among the Node.js crowd, is Browserify.* Browserify takes Node.js-style CommonJS modules and combines them all together into a single browser-friendly JavaScript source file that can be loaded with a simple `<script>` tag. With Browserify, it's even possible to consume AMD modules. For example, when we built Australia's National Map† on top of Cesium, we pulled Cesium's AMD modules into our Browserify build by using the *deamdify* plugin.

One nice aspect of Browserify is that it works with the Node.js ecosystem, even enabling us to use npm for our in-browser package management. It's quite refreshing to download, install, and bundle a third-party library—and make it trivial for other developers to do the same—with little more than an `npm install`.

A very interesting approach for serious application development is to construct the application as a large number of separately developed and versioned npm packages. Each package should be useful in its own right and hosted in a separate git repo. npm elegantly manages the dependencies between these packages. While it can be challenging to see how to factor an application into these stand-alone packages, the reward is a library of packages that can be reused across applications. The stackgl‡ project is a great example of this approach (Chapter 13).

4.2.4 TypeScript

Another approach to building a serious modularized application is to use an entirely different language that compiles to JavaScript. Two of the better-known languages in this category are CoffeeScript and Dart. Our favorite such language, though, is TypeScript,§ in large part because of its compatibility with JavaScript. All JavaScript code is automatically valid TypeScript code, and the output of the TypeScript compiler is idiomatic JavaScript much like what we'd write by hand. TypeScript has nice syntax for defining and exporting modules, and it can be configured to produce its generated JavaScript modules using either AMD or CommonJS format. As of TypeScript 1.x, the syntax for importing modules is as follows:

Listing 4.4 Importing modules in TypeScript.

```
import Cartesian3 = require('./Cartesian3');
import defaultValue = require('./defaultValue');
```

* http://browserify.org/
† https://github.com/NICTA/nationalmap
‡ http://stack.gl/
§ http://www.typescriptlang.org/

This syntax is likely to change in TypeScript 2.0, because TypeScript is intended to closely track the upcoming ECMAScript 6 standard, discussed later.

In addition to good module support, TypeScript also supports optional type annotations, which are then enforced by the compiler. In Cesium, all public APIs and most private ones have their types explicitly documented, because doing so makes the code easier to read and the API easier to understand. We believe that having a compiler that enforces type compatibility is very beneficial to improving the documentation as well as to eliminating a certain class of bugs.

4.2.5 ECMAScript 6

The upcoming version of JavaScript, called ECMAScript 6 or ES6, will have built-in support for modules and should be an official standard by the time you read this.* ES6 modules avoid AMD's "synchronized lists" problem, but because they are part of the language, they can load asynchronously within web browsers. ES6 dependencies are specified as follows:

Listing 4.5 Importing modules using ES6 syntax.

```
import Cartesian3 from 'Cartesian3';
import defaultValue from 'defaultValue';
```

Even if your application targets older browsers, you can start using ES6 today by using a tool that compiles ES6 to the current version of JavaScript, ES5. A great list of such tools is maintained by Addy Osmani.†

4.2.6 Other Options

There are many other approaches for modularizing JavaScript code. The Google Closure compiler‡ has support for modularization and is a popular option for building large JavaScript applications. We've even heard of folks using *#include* and the C preprocessor as a simple build process. When evaluating a toolchain for your serious application, be sure to consider how it will interface with the larger JavaScript ecosystem. Even the most beautiful module system is not so appealing if it makes it hard to leverage third-party libraries, documentation generation tools, test frameworks, test runners, etc.

4.3 Performance

Writing about JavaScript performance is tricky because it is a constantly moving target. Browser implementations continue to improve on a regular basis and what is slow now may not be slow for long. Still, even with the continually evolving nature of self-updating browsers, there is a set of common dos and don'ts that are generally applicable and unlikely to change.

* The schedule has slipped before, however. See http://ecma-international.org/memento/TC39-M.htm for the latest status.
† https://github.com/addyosmani/es6-tools
‡ https://developers.google.com/closure/compiler/

It's impossible for us to write about JavaScript performance without also admitting that it varies from browser to browser. Every JavaScript engine has its strengths and weaknesses and performance characteristics can vary wildly depending on the feature being used. Websites, such as jsPerf.com, have sprung up to help compare performance across a wide variety of browsers, usually as microbenchmarks of a particular language or library feature. While jsPerf can be useful, we recommend a more direct approach. All modern browsers have amazing profiling tools built directly into their equally amazing debug environments. Whether we are using Chrome, Firefox, Internet Explorer, or Safari, we find the easiest way to determine the locations of the slow spots is to simply run the code with the profiler enabled. Unfortunately, not all performance issues show up in the profiler. Some language features or other architectural choices can have a hidden cost that is distributed throughout the entire codebase. The best way to combat these types of issues is following the best practices we lay out here, as well as staying up to date on the ever-changing state of JavaScript engines.

4.3.1 Defining and Constructing Objects

Some of the most fundamental optimizations a JavaScript engine can make depend on having type information available. Unfortunately, because JavaScript is a dynamically typed language, this information is not easily obtained. Most engines use a technique known as type inferencing to deduce types from our code at runtime [Hackett 12]. The more our JavaScript code acts as it would in a statically typed language, the easier it is for the engine to optimize it. For example, consider this `Cartesian3` constructor function:

Listing 4.6 A simple `Cartesian3` constructor.

```
var Cartesian3 = function(x, y, z) {
  this.x = x;
  this.y = y;
  this.z = z;
};
```

Later, we may need to add a `w` property to a particular `Cartesian3` instance. It's tempting to use the extensibility of JavaScript objects to add the property with a simple `instance.w = 1`. This is not recommended, however, because it can negatively impact performance [Clifford 12].

In a statically typed language, we couldn't dynamically add the `w` property to a single instance of the `Cartesian3` class at all. Instead, we'd have to define a new class and instance entirely, and copy the values of the `x`, `y`, and `z` properties to the new instance. While modern JavaScript runtime engines have many tricks, they fundamentally still generate instructions in the same machine code that statically typed languages compile to.

When we define a constructor function like the preceding `Cartesian3`, many JavaScript VMs create an internal representation of its type to allow for fast property access and method calls, and to make its in-memory representation as efficient as possible. Ideally, a `Cartesian3` would be represented in memory as simply three floating-point numbers, though there's likely to be a bit more overhead than that even in the best JavaScript engines. The more optimal the in-memory layout of the `Cartesian3`

instance is, the more costly it will be to add that `w` property. It's likely that the JavaScript engine will have to reallocate our instance and copy the existing properties, just as we would manually in a statically typed language. Ultimately, the engine may decide to use a much less efficient representation for this now-dynamic instance.

In addition, when the engine generates machine code for a function that takes a `Cartesian3` instance as a parameter, it may create an optimized implementation tied to that type. For example, many engines employ inline caching, storing the results of method and property lookups in the generated code. A structural change to an instance, such as adding the `w` property, invalidates the inline caches. The engine will have to generate new code or, perhaps more likely, fall back on an unoptimized code path using on-stack replacement [Pizlo 14].

A better solution is to define a `Cartesian4` constructor function, including a `w` property from the start, and use instances of that type as needed. This gives the JavaScript engine the most information about our types and the best chance of being able to generate fast code. If something would be difficult or slow to do in a statically typed language, it will almost certainly be slow in JavaScript, too.

There are multiple ways to define and construct objects in JavaScript, but constructor functions like we used before are the fastest way. In our benchmarks, `Object.create` is three to five times slower than simply calling `new`, but it used to be much worse [Jones 11]. Object literal notation is almost as fast as new, but only if we cache functions in a parent scope to avoid recreating them every time. Keep in mind that these are microbenchmarks, so they may not always match real application performance. Rather than fret over object creation, a better solution is to avoid allocation altogether, which we discuss in Section 4.3.2.

Much like our method for object construction, our method for defining properties on an object can have a significant effect on overall performance. While some JavaScript engines will inline simple getters and setters, function call overhead is still significant enough in some browsers that it needs to be taken into account. This means that if we want to expose a property on an object, it's faster to make it a public field rather than abstract it behind get and set methods. Also, while `Object.defineProperty` lets us create modern properties found in languages such as C#, on some browsers they tend to have the same overhead as a function call.

The general rule we follow in Cesium is simple. If a property would normally be a simple getter or setter function in other languages, then we expose it directly. If a property requires additional work to be done on get or set, only then do we use `Object.defineProperty`. By exposing properties everywhere, this makes the API consistent and minimizes runtime overhead.

4.3.2 Garbage Collection Overhead

One of the primary issues in many high-performance JavaScript applications is garbage collection, and the nature of 3D applications compounds this problem further. For example, imagine we need to multiply two vectors for every object in the scene. Assuming we have 10,000 objects and we are targeting 60 frames per second, we end up creating 120,000 vector result objects per second. In languages such as C++ or C#, we wouldn't even consider this to be a problem because our vectors are likely allocated on the stack. In JavaScript, it can be a major bottleneck.

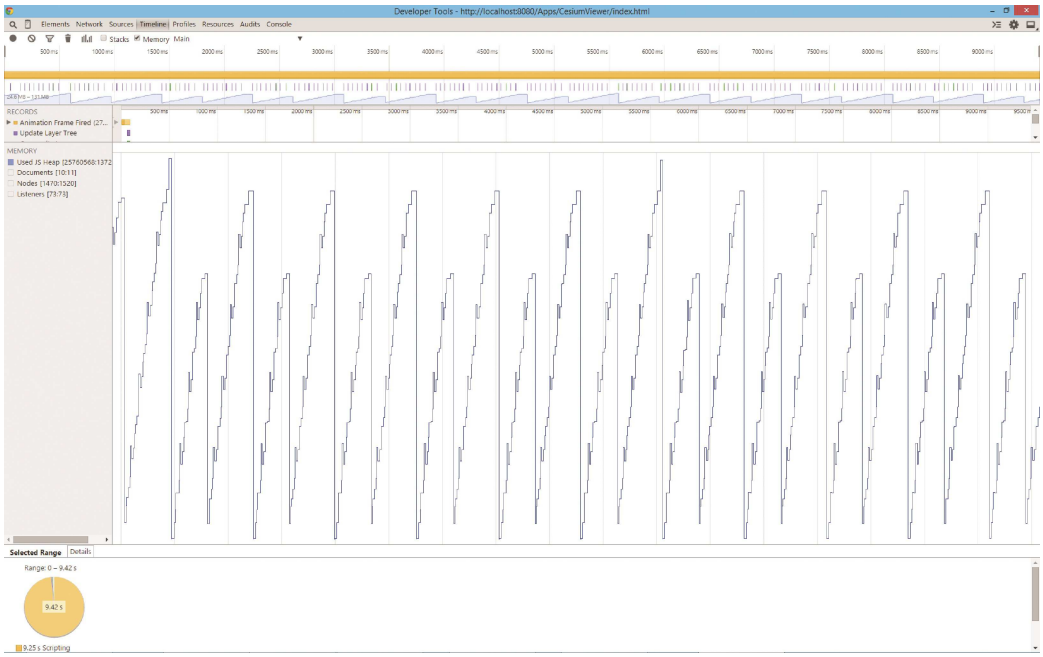

Figure 4.2

The sawtooth pattern, shown in Chrome Developer Tools, is a telltale sign of garbage collection issues.

Math operations, of the kind described previously, are unavoidable in WebGL applications. Early on in Cesium's development it was not uncommon to profile a particular use case only to discover that 50% of the time was being spent in garbage collection. A visual representation of this issue appears as a deep sawtooth pattern in browser profiling tools, as shown in Figure 4.2. The peaks are when the garbage collector kicks in, freeing memory but stealing valuable processing time from our own code. This kind of unwanted memory churn is usually created by algorithms that compute intermediate values that are quickly thrown away. For a more concrete example, see Listing 4.7, which is a simplified version of Cesium's `Cartesian3` linear interpolation implementation.

Listing 4.7 A memory-inefficient linear interpolation function.

```
Cartesian3.add = function(left, right) {
  var x = left.x + right.x;
  var y = left.y + right.y;
  var z = left.z + right.z;
  return new Cartesian3(x, y, z);
};

Cartesian3.multiplyByScalar = function(value, scalar) {
  var x = value.x * scalar;
  var y = value.y * scalar;
```

```
  var z = value.z * scalar;
  return new Cartesian3(x, y, z);
};

Cartesian3.lerp = function(start, end, t) {
  var tmp = Cartesian3.multiplyByScalar(end, t);
  var tmp2 = Cartesian3.multiplyByScalar(start, 1.0 - t);
  return Cartesian3.add(tmp, tmp2);
};
```

Every call to `lerp` allocates three objects: two intermediate `Cartesian3` instances and the result instance. While a microbenchmark of 100,000 calls takes about 9.0 milliseconds in Firefox, it doesn't expose a problem with garbage collection because the memory is not cleaned up until after our benchmark has already completed.

We can remove the extra memory allocation by using two simple techniques. First, we require users to pass in an already allocated result parameter to avoid having to create a new instance every time. Second, we use module-scoped scratch parameters in calls to add within `lerp`.

Listing 4.8 Memory-efficient linear interpolation using result parameters and scratch variables.

```
Cartesian3.add = function(left, right, result) {
  result.x = left.x + right.x;
  result.y = left.y + right.y;
  result.z = left.z + right.z;
  return result;
};

Cartesian3.multiplyByScalar = function(value, scalar) {
  result.x = value.x * scalar;
  result.y = value.y * scalar;
  result.z = value.z * scalar;
  return result;
};

var tmp = new Cartesian3(0, 0, 0);
var tmp2 = new Cartesian3(0, 0, 0);

Cartesian3.lerp = function(start, end, t, result) {
  Cartesian3.multiplyByScalar(end, t, tmp);
  Cartesian3.multiplyByScalar(start, 1.0 - t, tmp2);
  return Cartesian3.add(tmp, tmp2, result);
};
```

The modified implementation initializes two scratch variables during load but otherwise allocates no additional memory. While 100,000 calls to this version of lerp only took about 6 milliseconds in Firefox, most likely due to less object creation, it's not faster in all browsers. The important gain is only seen when profiling our application as a whole and

finding that our garbage collection time has dropped significantly in the profiler, resulting in higher frame rates.

It's not uncommon for Cesium to call over 100,000 functions like this per frame. Using result parameters and scratch variables saves us several milliseconds in our per-frame budget. While we hate that we have to clutter up our code and API like this, result parameters are an absolute necessity for anyone looking to write a performant, nontrivial WebGL application.

4.3.3 The Hidden Cost of Web Workers and How to Avoid It

In Cesium we allow user-defined geometric volumes such as ellipsoids, polygons, boxes, and cylinders to be computed synchronously on the main thread or asynchronously in a background thread via Web Workers. It was much to our surprise that our initial Web Worker implementation was several orders of magnitude slower than the single-threaded version. It turns out that Web Workers have a hidden cost when posting large amounts of data between threads.

To illustrate the problem, let's assume we were working only with polygons. Polygon triangulation is a CPU-intensive task and offloading it to a worker thread prevents us from locking up our application while it's processing. It's also not uncommon for a group of polygon definitions to contain over 500,000 vertices, such as the country borders in Figure 4.3.

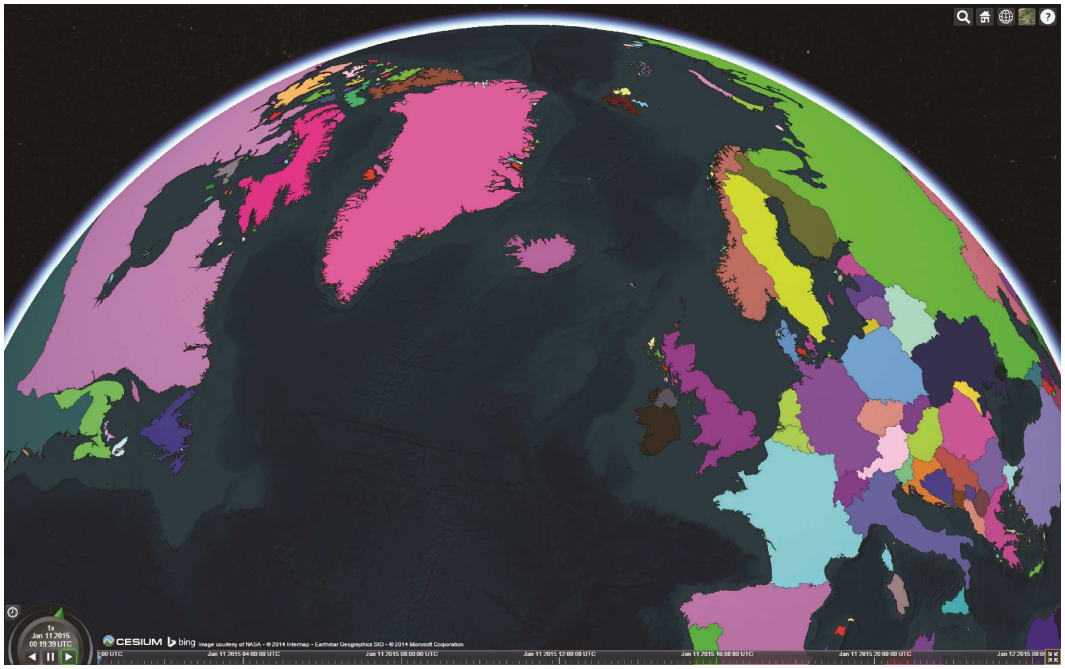

Figure 4.3

Highly detailed polygons exposed a performance issue in our Web Worker implementation.

Because JavaScript is historically a single-threaded language, working with Web Workers is very different from working with traditional APIs. A worker has no access to the DOM and executes in a different global context than the main thread.

Messages and data are always passed to workers by copy, because workers and the main thread do not share memory or any other mutable state. The HTML5 specification defines an algorithm, structured clone,* that is used to copy a message for a worker thread. In structured clone, the copied objects lose all prototype and function information, so it must be reconstructed by the receiving thread if needed. Structured clone is roughly equivalent to serializing an object to JSON and then deserializing it on the receiving end.

There is one way to avoid copying when passing data to a Web Worker. A Transferable† object, as the name implies, can optionally be transferred, instead of copied, to a worker thread. A transferred object becomes property of the receiving worker and is no longer accessible by the sender. Thus, we avoid both sharing and copying the data. While we unfortunately cannot mark our own objects as transferrable, there are two Transferable objects defined in the specification: `ArrayBuffer` and `MessagePort`.

For our use case, it turns out that the cloning operation is prohibitively slow in all browsers. This makes some sense since structured clone is a generic algorithm meant for copying almost anything. But exactly how slow is it? Suppose we had a Web Worker that simply posted the data it received back to the main thread. For an `ArrayBuffer`, it would transfer the buffer. All other data would be copied as normal. The time it would take to run this worker and receive the data back is almost entirely the overhead of the structured clone operation multiplied by two:

Listing 4.9 A simple Web Worker and timing function to measure the performance cost of structured clone.

```
//contents of worker.js
var onmessage = function(e) {
  postMessage(e.data, e.data.buffer ? [e.data.buffer] : undefined);
};

//code to spawn worker.js
function timeWorker(data) {
    var worker = new Worker("worker.js");
    var start = performance.now();

    worker.addEventListener("message", function(e) {
      console.log(performance.now() - start);
      worker.terminate();
    }, false);
    worker.postMessage(data);
}
```

In our tests, executing the preceding code with an array of 500,000 `Cartesian3` instances took an average of 3.8 to 6.2 seconds (not milliseconds!) to complete, depending

* http://www.w3.org/TR/html5/infrastructure.html#safe-passing-of-structured-data
† http://www.w3.org/TR/html5/infrastructure.html#transferable-objects

on the browser being used. Even worse is that because posting to and receiving from a worker is synchronous, half of this time is spent with the page locked up and unable to respond to user input. In hindsight, this shouldn't have surprised us, but it was still demoralizing when we first encountered it. In many cases, the overhead of sending data to a Web Worker was much worse than just doing the work synchronously and locking up the main thread. We felt that there had to be a better way.

As we mentioned previously, `ArrayBuffers` are one of the objects that can be transferred to a worker thread without copying. What if we manually packed our data into a typed array and transferred it to the worker? Could manually packing somehow be faster than what the browser is already doing in native code? The worker would itself have to unpack the data, and the packing and unpacking code would be specific to the arguments being sent to the worker, but we felt it was worth a try. Here's the modified code from before, along with two helper functions to pack and unpack arrays of `Cartesian3` instances:

Listing 4.10 Method of packing data into a typed array for near instantaneous transfer to worker threads.

```
function packCartesian3Array(data) {
    var j = 0;
    var packedData = new Float64Array(data.length * 3);
    for (var i = 0, len = data.length; i < len; i++) {
        var item = data[i];
        packedData[j++] = item.x;
        packedData[j++] = item.y;
        packedData[j++] = item.z;
    }
    return packedData;
}

function unpackCartesian3Array(packedData) {
    var j = 0;
    var data = new Array(packedData.length/3);
    for (var i = 0; i < packedData.length; i++) {
        var x = packedData[j++];
        var y = packedData[j++];
        var z = packedData[j++];
        data[i] = new Cartesian3(x, y, z);
    }
    return data;
}

function timeWorker(data) {
    var packedWorker = new Worker("worker.js");
    var start = performance.now();
    var packedData = packCartesian3Array(data);

    packedWorker.addEventListener("message", function(e) {
        var receivedData = unpackCartesian3Array(e.data);
        console.log(performance.now() - start);
        packedWorker.terminate();
    }, false);
    packedWorker.postMessage(packedData, [packedData.buffer]);
}
```

The end result was pleasantly surprising. The manually packed version is tremendously faster than relying on default cloning, taking an average of only 60 to 600 milliseconds to complete. While it has to be manually maintained, this technique allows us to pack all objects and their properties, including strings, into a single typed array for efficient transfer.

4.3.4 Making Optimal Use of Multiple Cores

A common technique in multithreaded programming is to use as many threads as cores available to achieve maximum parallelism. If we use too many threads, performance will suffer due to excessive context switching and, with too few threads, spare cores go to waste. Unfortunately, there is no official standard for accessing the number of cores on a system via JavaScript, which we feel is a major oversight that greatly reduces the usefulness of Web Workers in general. Thankfully, a new nonstandard property has recently been added to some browsers which exposes the number of logical processors available on the client system, `navigator.hardwareConcurrency`.* Even though it is only supported in Chrome, Opera, and Safari, it is too useful not to mention here. While Firefox† and IE‡ have both decided not to implement the property at this time, a shim§ is available.

4.4 Automated Testing of WebGL Applications

We believe that automated tests are critical for any serious application. A good suite of automated tests gives us confidence that our code is working at a deep level, because we're able to test edge cases and uncommon paths. It also greatly improves our confidence when refactoring, which is critical for an application with a development plan that spans years.

While this is true in an application written in any language, there are some additional concerns in JavaScript. Web browsers are frustratingly tolerant of broken JavaScript code. We can write a JavaScript function containing code that is not even syntactically valid and the browser will not complain until that function is actually executed. Similarly, a simple typo in a code path—one that handles errors, perhaps—will likely go undetected unless that code path is executed. Automated tests with excellent code coverage are our best available tool for making sure that all of our code is executed and has the correct behavior.

There is an astounding number of JavaScript test frameworks and runners available today, each with a chart comparing itself to a subset of the others that clearly shows that it is the best of the lot. For Cesium, we settled on using the Jasmine test framework and the Karma test runner.

4.4.1 Jasmine

In some ways, Jasmine is "old school." It doesn't use a module system; instead, we include it with a script tag and it adds its functions to the global scope. It also doesn't have a ton of features. What it does have, however, is clean and elegant syntax for writing tests, and its

* https://wiki.whatwg.org/wiki/Navigator_HW_Concurrency
† https://groups.google.com/forum/#!topic/mozilla.dev.platform/QnhfUVw9jCI
‡ https://status.modern.ie/hardwareconcurrency
§ https://github.com/oftn/core-estimator

simplicity makes it possible to integrate into a wide variety of applications. For example, we've successfully used Jasmine in both AMD-based applications and CommonJS/Browserify-based applications.

Jasmine is a behavior-driven development (BDD) framework, which essentially means that we write our tests in a style that feels a bit like describing in English what our code is supposed to do:

Listing 4.11 A Jasmine unit test for `Cartesian3` normalization.

```
describe('Cartesian3', function() {
  it('normalizes to a vector with magnitude 1', function() {
    var original = new Cartesian3(1.0, 2.0, 3.0);
    var normalized = Cartesian3.normalize(original);
    var magnitude = Cartesian3.magnitude(normalized);
    expect(magnitude).toEqual(1.0);
  });
});
```

Running Jasmine tests, referred to as specs in Jasmine, in a browser requires us to set up a `SpecRunner.html` file, using the one included in the Jasmine distribution as a template (Figure 4.4). The specifics of the SpecRunner vary depending on how our application is structured. In all cases, we include the Jasmine scripts using standard `<script>` tags. Actually running the specs, however, is dependent on our project architecture.

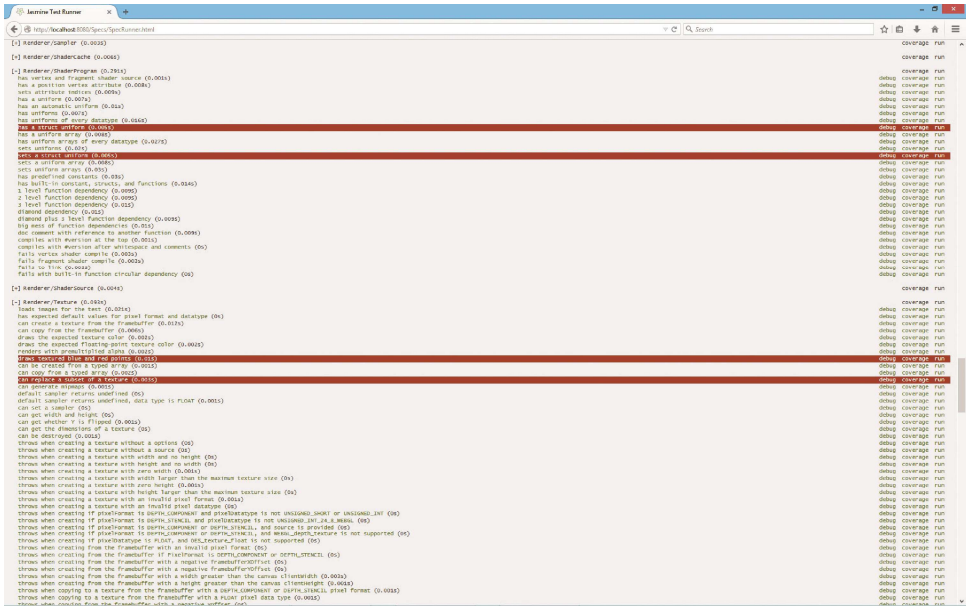

Figure 4.4

Cesium's customized SpecRunner.html with four failing tests.

If we forgo a module system, this is straightforward, though painful for a large application: Just include the script tags for all of our source files and spec files, in the right order, in `SpecRunner.html`.

If we have just a single combined source file with all of our specs and their dependencies, as we get with CommonJS modules built using Browserify, it's easy: We simply add the single script tag for the built JavaScript file to `SpecRunner.html`.

The asynchronous module definition (AMD) case is more complicated than the other two, simply because AMD is asynchronous. By default, Jasmine runs all of the specs it knows about in the `window.onload` event. When we're using AMD, the spec modules are not yet loaded by the time `window.onload` fires. We need to load all of our spec modules, and only launch Jasmine once they are all loaded. First, we add a `<script>` tag for RequireJS and give it a `data-main` attribute that is the entry point module for our specs.

```
//SpecRunner.html
<script data-main = "specs/spec-main" src = "../requirejs-2.1.9/
  require.js"></script>
```

Our `spec-main` module requires-in all the spec modules and then executes the Jasmine environment:

Listing 4.12 An entry point module for executing Jasmine specs.

```
//spec-main.js
define([
  './Cartesian3Spec',
  './Matrix4Spec',
  './RaySpec'
], function() {
  var env = jasmine.getEnv();
  env.execute();
});
```

We don't need to actually have a parameter for each module in our `spec-main` function because we don't need to use it; we only need to ensure that the modules are loaded.

Of course, it's unfortunate that we need to list every spec module in this way. The complete list of spec files has to be specified *somehow*, though, because the web browser can't inspect the local filesystem to determine the list of specs. In Cesium we use a simple build step to automatically generate the complete list of spec modules so that we don't have to maintain this list manually.

With our `SpecRunner.html` set up properly, we only need to serve it up through any web server and visit it from any browser to run our tests in that browser.

4.4.2 Karma

Running tests in Jasmine is a manual process. We open a web browser, navigate to our `SpecRunner.html` file, wait for the tests to run, and look for any failures. Karma lets us automate this.

With Karma, a single command can launch all the browsers on the system, run the tests in each of them, and report the results on the command line. This is critical for working with continuous integration (CI) because it gives us a way to turn test failures generated via a web browser into build process failures. Karma can also watch the tests for changes and automatically rerun them in all browsers, which is handy when practicing test-driven development or when otherwise focused on building out the test suite.

Compared to Jasmine, setting up Karma is pretty easy, even for use with AMD.* Install it in your Node.js environment using npm, and then interactively build a config file for your application by running the following:

```
karma init
```

Karma has out-of-the-box support for Jasmine and several other test frameworks, and more can be added via plugins. Once Karma is configured, we can run the tests in all configured browsers by running

```
karma start
```

4.4.3 Testing JavaScript Code That Uses WebGL

Most of what we've discussed so far is applicable to testing just about any JavaScript application. What unique challenges does WebGL present?

We can test most of our graphics code without ever actually rendering anything. For example, we can validate our triangulation, subdivision, batching, and level-of-detail selection algorithms with standard unit tests that invoke these algorithms and assert that they produce the data structures and numbers that we expect. Inevitably, however, some portion of our rendering code—however small—is intimately tied to the WebGL API.

Purists might argue that our unit tests should never make calls into the WebGL API directly, instead calling into a testable abstraction layer of mocks and stubs. In this perfect world, unit testing our WebGL application would be no different from unit testing any other application. We'd write tests that drive our code and then assert that the correct pattern of WebGL functions was invoked, without ever actually invoking those functions.

While we do appreciate that there is a place for this sort of testing, we also believe that every serious WebGL application will eventually have to step outside it. For one thing, mocking and stubbing the entire WebGL API—or at least a large enough subset of it to test a sophisticated piece of application logic—would be a significant undertaking.

The bigger problem is that WebGL is a complicated API. If we only ever tested against a mocked version of it, we wouldn't have great confidence that our code would work against the real one. Or, perhaps we should say real *ones*, because in some sense, each browser and GPU combination may have unique capabilities and bugs. We might choose to call these tests using the real API *integration tests* rather than *unit tests*, but the fact remains that they're an important part of our testing picture.

With that in mind, we've purposely avoided discussing cloud-based JavaScript testing solutions such as Sauce Labs† in previous sections. This is because none of these solutions, as of this writing, have reliable support for WebGL. It's unfortunate, because we'd love to be able to run our tests across a wide variety of operating systems and web browsers without

* http://karma-runner.github.io/0.12/plus/requirejs.html
† https://saucelabs.com/

maintaining any test infrastructure ourselves. But it's also understandable, because these solutions necessarily use virtualization, and GPU hardware acceleration in a virtualized environment is still in its infancy. Thus, our current approach is to use Karma to run tests on physical machines that we maintain, all driven by our CI process.

When we're writing a WebGL application, we may have hundreds or thousands of lines of code that all conspire to put a certain pattern of pixels on the screen. How can we write automated tests to confirm that the pattern of pixels is correct?

There is no easy answer to this question. On previous projects, we wrote tests to draw a scene, take a screenshot, and compare it to a "known good" screenshot. This was extremely error prone. Differences between GPUs and even driver versions inevitably caused our test images to be different from the master images. Anytime a test failed, we immediately wondered what was wrong with the driver, operating system, or test, rather than asking ourselves what might be wrong with our code. We used "fuzzy" image comparison to make our tests assert that an image "mostly" matched the master image, but even then it was a constantly frustrating balancing act between reporting failure on a new GPU where the code was actually working perfectly well, and reporting success even though something actually went wrong. We would not recommend this approach.

Others have reported better success with comparing screenshots by maintaining and manually verifying a set of "known good" images for every combination of platform, GPU, and driver [Pranckevičius 2011]. While it's easy to see how this would be effective, it also strikes us as an extraordinarily costly approach to testing.

Instead, most Cesium tests that do actual rendering render a single pixel, and then assert that the pixel is correct. For example, here's a simplified test that asserts a polygon is drawn:

Listing 4.13 Single pixel sanity checking in Cesium unit tests.

```
it('renders', function() {
  var gl = createContext();
  setupCamera(gl);
  drawPolygon(gl);

  var pixels = new Uint8Array(4);
  gl.readPixels(0, 0, 1, 1, gl.RGBA, gl.UNSIGNED_BYTE, pixels);
  expect(pixels).not.toEqual([0, 0, 0, 0]);

  destroyContext(context);
});
```

This test only asserts that the pixel is not black, which is typical of the single-pixel rendering tests in Cesium. Sometimes we may check for something a bit more specific, like nonzero in the red component or "full white." We generally don't check for a precise color, though, because differences between browsers and GPUs can make even that test unreliable.

While the preceding example test creates a unique WebGL context for the test, we try to avoid this in the Cesium tests. One reason is that context creation and setup take time, and we want our tests to run as quickly as possible. A more serious problem, though, is that web browsers don't expect applications to create and destroy thousands of contexts. We've seen bugs in multiple browsers where context creation would start failing midway through

our tests. On the other hand, using a single context for all tests risks a test corrupting the context's state and causing later tests to fail. In Cesium, we've found a good balance by creating a context for each test suite. A test suite is a single source file that tests a closely related group of functionality, such as a single class, so it's usually easy to reason about the context state changes occurring in the suite.

A single-pixel rendering test like this is far from exhaustive, of course. Because it really just asserts that the polygon put *something* on the screen, there are plenty of things that could go wrong and still allow this test to pass. The opposite is not true. We should never see this test fail when the polygon, WebGL stack, and driver are working correctly.

4.4.4 Testing Shaders

Cesium has a library of reusable GLSL functions for use in vertex and fragment shaders. Some of these are fairly sophisticated, such as computing the intersections of a ray with an ellipsoid or transforming geodetic latitude to the Web Mercator coordinates commonly used in web mapping. We find it very beneficial to unit test these shader functions in much the same way we would unit test similar functions written in JavaScript. We can't run Jasmine on the GPU, however, so how do we test them?

Our technique is straightforward. We write a fragment shader that invokes the function we wish to test, checks whatever condition we're testing, and outputs white to `gl_FragColor` if the condition is true. For example, the test shader for the `czm_transpose` function, which transposes a 2×2 matrix, looks like this:

Listing 4.14 Testing reusable functions in GLSL.

```
void main() {
      mat2 m = mat2(1.0, 2.0, 3.0, 4.0);
      mat2 mt = mat2(1.0, 3.0, 2.0, 4.0);
      gl_FragColor=vec4(czm_transpose(m)==mt);
}
```

When `czm_transpose` computes the correct transpose, this shader sets `gl_FragColor` to white. If the transpose is incorrect, `gl_FragColor` is transparent black.

We then invoke this test shader from a Jasmine spec. Our spec draws a single point with a trivial vertex shader and the fragment shader above. It then reads the pixel that the shader wrote, using `gl.readPixels`, and asserts that it is white.

We've found this to be an easy, lightweight, and effective way to test shader functions. Unfortunately, it is not possible to test the entire vertex or fragment shaders in this way, nor is it straightforward to assert more than one condition per test. If either of those features is required, consider using a more full-featured GLSL testing solution such as GLSL Unit.*

For Cesium, however, we've found this to be unnecessary. By testing the building blocks of our shaders—the individual functions—and keeping the `main()` functions as simple as possible, we are able to have high confidence in our shaders without a complicated GLSL testing process.

* https://code.google.com/p/glsl-unit/

4.4.5 Testing Is Hard

In Cesium, we have a few types of tests:

- Tests of underlying algorithms that verify the data structures and numbers that the algorithms produce. These tests don't do any rendering.
- Rendering smoke tests, as described in Section 4.4.3. These usually render a single pixel and verify that it is not wildly wrong. We also sometimes render a full scene and verify only that no exceptions were thrown during the process.
- Shader function tests, as described in Section 4.4.4. These test the reusable functions that compose our shaders by invoking them in a test fragment shader and asserting that it produces white.

We find these types of tests to be relatively easy to write, robust, and well worth the time investment it takes to write them.

Unfortunately, we've found no way around having an actual human run the application once in a while, on a variety of systems and in a variety of browsers, to confirm that the rendered output is what we expect.

Acknowledgments

We'd like to thank everyone who reviewed this chapter and gave us valuable feedback: Jacob Benoit, Patrick Cozzi, Eric Haines, Briely Marum, Tarek Sherif, Ishaan Singh, and Traian Stanev. We'd also like to thank our families for their patience and understanding as we wrote this chapter, and our employers, Analytical Graphics, Inc. (AGI) and National ICT Australia (NICTA), for their flexibility. Finally, we'd like to thank Scott Hunter, who taught us virtually all of what we've written here, except for the parts that are wrong. NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

Bibliography

[Cozzi 14] Patrick Cozzi. "Why Use WebGL for Graphics Research?" http://www.realtimerendering.com/blog/why-use-webgl-for-graphics-research/, 2014.

[Hackett 12] Brian Hackett and Shu-yu Guo. "Fast and Precise Hybrid Type Inference for JavaScript." http://rfrn.org/~shu/drafts/ti.pdf, 2012.

[Jones 11] Brandon Jones. "The somewhat depressing state of Object.create performance." http://blog.tojicode.com/2011/08/somewhat-depressing-state-of.html, 2011.

[Pizlo 14] Filip Pizlo. "Introducing the WebKit FTL JIT." https://www.webkit.org/blog/3362/introducing-the-webkit-ftl-jit/, 2014.

[Pranckevičius 2011] Aras Pranckevičius. "Testing Graphics Code, 4 Years Later." http://aras-p.info/blog/2011/06/17/testing-graphics-code-4-years-later/, 2011.

[Clifford 12] Daniel Clifford. "Breaking the JavaScript Speed Limit with V8." Google I/O https://www.youtube.com/watch?v=UJPdhx5zTaw 2012.