November 26th, 2015
Joseph Cameron
http://jfcameron.github.io/

An introduction to WebGL

What is WebGL

WebGL is a specification of the OpenGL graphics API designed for browsers. It is currently supported by all major browsers, both on desktop and mobile devices. WebGL allows you to program JavaScript applications that interact directly with the user's graphics device. This means you can create realtime 3D applications that are embedded directly in a web page. As long as the application isn't excessively complicated, it will be able to run on any modern device with an internet connection. The standard is closely tied to the OpenGL ES (Embedded Systems) standard, which is used for mobile devices. If you learn WebGL, it goes a long way to understanding how to write a 3D renderer on cellphones, tablets, set-top boxes etc. and vice versa.

How is WebGL being used

WebGL has been available for a number of years now, but the technology seems to be pretty underused in my opinion. There are a few applications out there that are very interesting and go a long way to demonstrate the power of WebGL and why it's worth learning.

Three.js

- free 3D game engine for desktop browsers. http://threejs.org/
- Example Three game "Gorescript" https://timeinvariant.github.io/gorescript/play/

Three is a very impressive piece of software. It features a deferred renderer, which allows you to implement all sorts of modern graphical effects. Because the OpenGL feature that deferred rendering requires is not part of the current WebGL standard, Three relies on browser specific API extensions to get access to it. This introduces compatibility issues among devices and it means that devices that do support WebGL but do not support this feature will be unable to use the deferred renderer. Fortunately the next version of WebGL does include this feature, called "MRT" (Multiple Render Targets), and once the standard becomes ubiquitous, these extensions will no longer be required.

Shadertoy.com

- Website for shader programmers to show off their fragment shaders (also called pixel shaders) https://www.shadertoy.com/.
- Check out the site to see various graphical effects and accompanying source code. You see a huge amount of raytracing demos, which seems to be the effect the author of the site is most interested in
- Written by Inigo Quilez, lots of interesting information and demos on raytracing can be found on his personal website http://www.iquilezles.org/

Ray Tracing is a cool technique that allows for some very impressive visuals. I really suggest taking a look at shadertoy and Inigo Quilez's website to see what it's all about. Shadertoy is especially good for this because you can see the demo, the code and you can make updates to it in real time. Essentially, instead of working with rasters of polygonal meshes, you have defined your geometry mathematically and are using those mathematical shapes to render the geometry directly in the fragment shader. This lets you have perfectly smooth shapes etc., it lets you describe interactions between shapes that would otherwise be extremely complicated to mimic with meshes. Unfortunately ray tracing is pretty slow. This is because you're not taking advantage of the rest of the graphics pipeline and instead forcing a single part of the pipeline to do everything. In other words, a lot of your graphics hardware just isn't being used. Maybe in the future there will be hardware accelerated ray tracing graphics cards, I haven't looked into whether or not this is being considered by card producers.

Emscripten + WebGL

- Used as a means to port C and C++ 3D programs onto browsers
- Emscripten is a "transpiler," it converts C++ code to JavaScript https://github.com/kripken/emscripten
- WebGL is then used to allow your C++ code to reach the graphics device within the context of the browser so that it can still make OpenGL calls and therefore render images
- Notable ports include Bullet physics, a well regarded open source 3D rigid body physics engine and SDL, a very popular library for developing games (SDL is used in the OSX and Linux ports of Valve's Source engine)
- Funny ports include js.js, a javascript interpreter. https://github.com/jterrace/js.js. This does actually have some relevant security applications, but the idea of having a JS interpreter interpreting a JS interpreter interpreting JS... you better have a good reason to do this in production.

Emscripten is another really interesting development. Converting native code to JS is interesting in itself, as it gives you (as a C++ developer) another platform to build for. Having access to OpenGL via WebGL opens up a lot of options in terms of porting desktop videogames to the web. The JavaScript code that Emscripten generates is also a lot faster than JS you'll see written by humans. It generates a strict form of the language that makes it easier for the interpreter to read the code. It also fills the code with little hints about underlying types etc. that the interpreter searches for and uses to run even faster. It also completely avoids the garbage collector by creating and using a virtual heap, instead of creating and orphaning objects ad hoc in the more usual JS way.

If you are familiar with the game engine Unity, you may have noticed the move from the Web Player option to Web build. Basically NAPI, a browser API that Unity's Web Player depended on to work has been dropped by chrome over security concerns. The way Unity is able to still build for web is by transpiling the engine and your game code to JS via Emscripten. The build option is still in beta, but it will presumably become faster and more stable in the future. Here's an article by Unity about this:
http://blogs.unity3d.com/2014/04/29/on-the-future-of-web-publishing-in-unity/

Finally, I'll show you what I've done using WebGL. I created a little game engine and embedded it on the landing page of my portfolio as a Game Dev version of the "Hero Image."

http://jfcameron.github.io/

- Features a 3D renderer, an Entity Component System and a rigid body physics system
- Decently commented, but I still haven't written any documentation, so I wouldn't suggest using it
- May find some use as a reference
- Thinking of replacing this project with a transpiled C++ version, but the C++ version is missing some features compared to the current JS version

***

How does OpenGL work

WebGL and OpenGL generally can seem like a fairly difficult thing to learn. If you don't have any experience with graphics APIs - like with DirectX for example - you're going to immediately run into a wall of new and unintuitive concepts. If you take the time to read, study and (most importantly) practice and really confront the things you find confusing, you'll figure it out. Once you understand the basics, you'll get a sense of the system at large. The more you understand a given individual piece, the easier the other pieces will become to learn.

The first thing to learn about is the pipeline. The pipeline is a way to think about the sequence of operations done by both the graphics hardware and by the graphics programmer to turn your 3D or 2D assets into a picture on the screen. The pipeline is what happens after you call draw but before you see an image on the screen.

https://www.opengl.org/wiki/Rendering_Pipeline_Overview

From the diagram you can see it operates on vertexes, produces fragments and outputs them to the display. There are also two steps called "Shaders," a word you may know as a synonym with graphical effect. More generally, they're points in the pipeline where graphics programmers can add custom operations. The names of the shaders, Vertex Shader and Fragment shader, tell you what they operate on. So what are vertexes and fragments? Verts are just the 2d or 3d points that make up the mesh that is currently being drawn. Fragments are more colloquially called "pixels," but this isn't really the correct term. My graphics teachers always insisted on calling them "potential pixels," since fragments can be destroyed by the fragment shader or during culling operations, but yes, you're essentially looking at a big array of fragments when you look at your computer screen.

Generally the data coming into the pipeline from verts alone will not be enough to create the graphical effects you want. This is where something called "uniforms" come in. Uniform data is data that is passed to the shader that always looks the same, no matter what vertex is being operated on, hence the name "uniform." A typical example of a uniform would be a texture. Say the pipeline has reached your fragment shader, you've got this fragment, which represents a single piece of color somewhere along the surface of your mesh, if you want to add color from a texture, you'd perform a lookup on the texture and adjust the fragment's color accordingly.

At a glance, that's pretty much it. Verts, frags, uniforms and shaders. Your shaders decide how you render your scene, whether or not there is lighting or shadow casting or diffuse coloring or whatever you want. Verts, frags, uniforms supply your shaders with the information they need to create these effects. It all comes down to shader programming and providing the necessary data for those shaders to work.

Next I'll go over a very basic example of rendering a textured quad and show you how it all comes together in practice.

QA

***

Links

Three.js
http://threejs.org/
https://timeinvariant.github.io/gorescript/play/

ShaderToy
https://www.shadertoy.com/
http://www.iquilezles.org/

Emscripten
https://github.com/kripken/emscripten
http://blogs.unity3d.com/2014/04/29/on-the-future-of-web-publishing-in-unity/

Js.js
https://github.com/jterrace/js.js

Portfolio
http://jfcameron.github.io/

OpenGL
https://www.opengl.org/wiki/Rendering_Pipeline_Overview - The pipeline
https://www.khronos.org/registry/webgl/specs/latest/ - WebGL official specification
https://www.khronos.org/files/webgl/webgl-reference-card-1_0.pdf - WebGL reference card