

Introduction to WebGL



CS 418: Interactive Computer Graphics
Professor Eric Shaffer

The WebGL Rasterization Engine

- WebGL relatively new (2011) 3D graphics support for web
- WebGL advantages
 - runs in browser
 - naturally cross-platform
 - don't need to obtain/build other libraries
 - gives you "windowing" for free
 - easy to publish/share your stuff
- Disadvantages
 - Depends on how you feel about JavaScript
 - Performance can be tricky
 - Pretty low-level

Programming Language for CS 418

- HTML
- JavaScript
- WebGL
- WebGL version of the GLSL shading language (**runs on GPU**)


- Chrome as default browser
- Chrome DevTools to debug code
- Some WebGL examples:
<https://www.chromeexperiments.com/webgl>

Firefox is fine too...and probably Edge.
We test in Chrome...

JavaScript

- We will provide example code
- You are responsible for learning what you need to complete the assignments
- Mozilla reference/tutorials are quite good

← → ↻ Secure | <https://developer.mozilla.org/en-US/docs/Web/JavaScript>

 Technologies ▾ References & Guides ▾ Feedback ▾

JavaScript

Web technology for developers ▸ JavaScript

Related Topics

JavaScript

Tutorials:

- ▶ Complete beginners
- ▶ JavaScript Guide
- ▶ Intermediate
- ▶ Advanced

References:

- ▶ Built-in objects
- ▶ Expressions & operators
- ▶ Statements & declarations
- ▶ Functions
- ▶ Classes
- ▶ Errors
- ▶ Misc
- ▶ New in JavaScript

Documentation:

- ▶ Useful lists
- ▶ Contribute

JavaScript (JS) is a lightweight interpreted or JIT-compiled programming language with first-class functions. While it is most well-known as the scripting language for Web pages, [many non-browser environments](#) also use it, such as [Node.js](#), [Apache CouchDB](#) and [Adobe Acrobat](#). JavaScript is a prototype-based, multi-paradigm, dynamic language, supporting object-oriented, imperative, and declarative (e.g. functional programming) styles. Read more [about JavaScript](#).

This section of the site is dedicated to the JavaScript language itself, and not the parts that are specific to Web pages or other host environments. For information about APIs specific to Web pages, please see [Web APIs](#) and [DOM](#).

The standard for JavaScript is [ECMAScript](#). As of 2012, all [modern browsers](#) fully support ECMAScript 5.1. Older browsers support at least ECMAScript 3. On June 17, 2015, [ECMA International](#) published the sixth major version of ECMAScript, which is officially called ECMAScript 2015, and was initially referred to as ECMAScript 6 or ES6. Since then, ECMAScript standards are on yearly release cycles. This documentation refers to the latest draft version, which is currently [ECMAScript 2018](#).

Do not confuse JavaScript with the [Java programming language](#). Both "Java" and "JavaScript" are trademarks or registered trademarks of Oracle in the U.S. and other countries. However, the two programming languages have very different syntax, semantics, and uses.

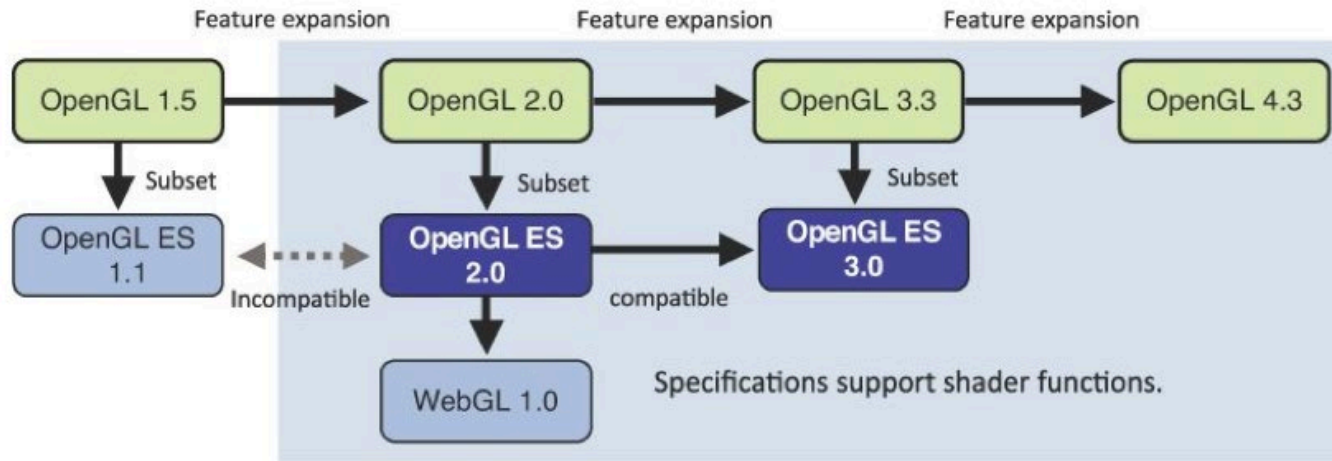
<https://developer.mozilla.org/en-US/docs/Web/JavaScript>

What is WebGL?

Let's start with a word about OpenGL

- Open standard for 3D graphics programming
 - Developed by Silicon Graphics in 1992
 - Available on most platforms...
 - Bindings available for lots of languages...
 - It's low level
- "Windowing" typically requires another library
 - e.g. GLUT
- Version 3.0 (2008) introduced programmable shaders
 - Deprecated fixed-function pipeline
- **Vulkan** API is the successor technology (still pretty new...2016ish)

WebGL is not exactly OpenGL



Yes - you can use
WebGL 2.0 in this
course

Figure from *WebGL Programming Guide: Interactive 3D Graphics Programming with WebGL* by Matsuda and Lea

WebGL Application Structure

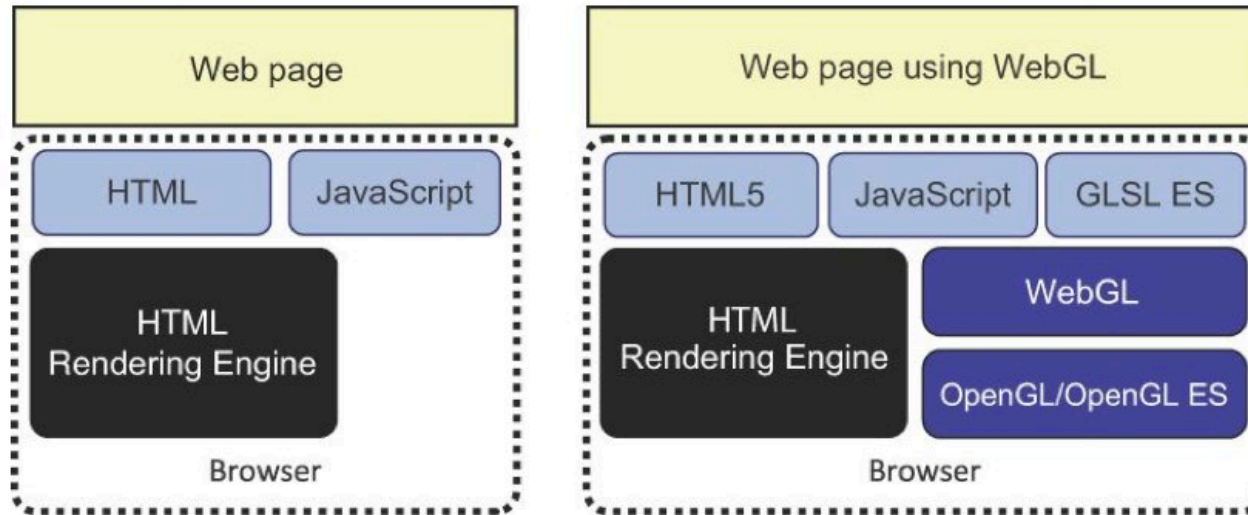


Figure from *WebGL Programming Guide: Interactive 3D Graphics Programming with WebGL* by Matsuda and Lea

Your application will generally just have HTML and JavaScript files

WebGL and GLSL

- WebGL requires you provide shader programs
- GLSL OpenGL Shading Language
- C-like with
 - Matrix and vector types (2, 3, 4 dimensional)
 - Overloaded operators
 - C++ like constructors
- WebGL functions compile, link and get information to shaders

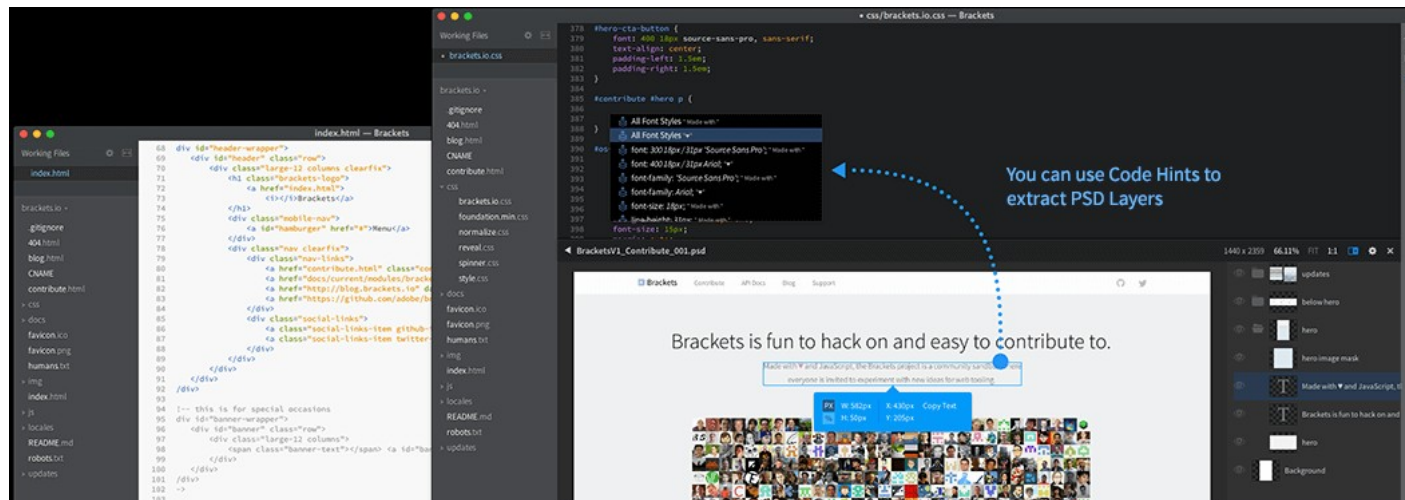
Shaders

- Shader source code will be in the HTML file or a JS file...usually
- Vertex Shaders generally move vertices around
 - Projection, animation, etc.
 - Assign a value to the built-in variable `gl_Position`
- Fragment Shaders generally determine a fragment color
 - Assign a value to the built-in variable `gl_FragColor`

You Need a Text Editor

Brackets is a good choice...but whatever works for you is fine

<http://brackets.io/>



Time to Write Some HTML

A few notes

- We will keep everything in a single HTML file for this example
- ...for larger programs we will separate the HTML and JavaScript

Using WebGL entails writing a bunch of startup code

- Complexity comes from the flexibility of the API
 - Will enable you to do really sophisticated stuff later on....
- Eventually we'll use a helper library for the startup code...

[https://illinois-cs418.github.io/Examples/Hello%20Triangle.html](https://illinois-cs418.github.io/Examples>Hello%20Triangle.html)

The HTML

```
<!DOCTYPE HTML>
<html lang="en">
<head>
<title>Hello Triangle</title>
<meta charset="utf-8">
</head>
<body onload="startup();">
  <canvas id="myGLCanvas"
    width="500" height="500">
</canvas>
</body>
</html>
```

We create an HTML page

We create an HTML5 **<canvas>**

That is 500 x 500 pixels which we will draw into.

We give it an id so we can refer to it in the javascript that we will write.

onload specifies an entry point into the JavaScript we will write...a function named **startup()** will be called on a page load

Adding JavaScript

```
<script type="text/javascript"> var gl;
var canvas;
var shaderProgram;
var vertexBuffer;

function startup(){
    canvas=document.getElementById("myGLCanvas");
    gl=createGLContext(canvas);
    setupShaders();
    setupBuffers();
    gl.clearColor(0.0, 0.0, 0.0, 1.0);
    draw();
}
</script>
```

JavaScript is included inside **<script>** tags

We have some global variables...
...and our initial function calls some other functions.

Bolded functions are the ones we will write.

clearColor is a WebGL function that sets the initial color of the pixels in the raster

getElementById is a Document Object Model (DOM) function that gets us a reference to the canvas created in the HTML document

Getting a WebGL Context

```
function createContext(canvas) {  
  var names = ["webgl", "experimental-webgl"]; var context = null;  
  for (var i=0; i < names.length; i++) {  
    try {  
      context = canvas.getContext(names[i]);  
    }  
    catch(e) {}  
    if (context) { break;}  
  }  
  if (context) {  
    context.viewportWidth = canvas.width; context.viewportHeight  
    = canvas.height;  
  } else {  
    alert("Failed to create WebGL context!");  
  }  
  return context;  
}
```

We need to make sure the browser supports WebGL...so we try to get a reference to a WebGL context using the two names under which it might exist

If we get a context, we set the viewport dimensions of the context to match the size of the canvas.

You can choose to use less than the full canvas.

Creating Vertex Shader

```
var vertexShaderSource =  
  "attribute vec3 aVertexPosition;  \n" +  
  "void main() {                      \n" +  
  "  gl_Position = vec4(aVertexPosition, 1.0);  \n" +  
  "}"                                     \n"
```

We'll talk more about shaders later but for now you should know:

We need to create a vertex shader program written in GLSL

We will use a JavaScript string to hold the source code for the vertex shader. **We'll see a better way to do this later.**

The shader must assign a value to `gl_Position`

Our shader basically just takes the position of an incoming vertex and assigns that position to `gl_Position`.

It actually does one thing to the incoming position...do you know what that is?

Creating Fragment Shader

```
var fragmentShaderSource =  
    "precision mediump float;          \n"+  
    "void main() {                      \n"+  
    " gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0); \n"+ "}  
    \n";
```

Like the vertex shader program, the fragment shader code is written in GLSL and held in a string.

You can think of fragments as being almost pixels...they are produced by the WebGL rasterizer and have a screen space position and some other data related to them.

Our shader simply assigns each fragment the same color.

Again, we'll talk more about what the shaders do later...

Compiling the Shaders

```
function setupShaders() {  
    var vertexShaderSource = ...  
    var fragmentShaderSource = ...  
  
    var vertexShader = loadShader(gl.VERTEX_SHADER,  
    vertexShaderSource);  
  
    var fragmentShader = loadShader(gl.FRAGMENT_SHADER,  
    fragmentShaderSource);  
  
    ...  
}
```

We have a homemade helper function that compiles the shader and checks if there were compilation errors.

If there was an error, a JavaScript alert is issued and the shader object deleted.

Otherwise the compiled shader is returned.

Important:

- You can create multiple shader programs
- You can switch which one you use while drawing a single frame
 - ...just use the **useProgram** function in WebGL
- Each shader program needs a vertex shader and fragment shader

```
function loadShader(type, shaderSource) {  
    var shader = gl.createShader(type);  
    gl.shaderSource(shader, shaderSource);  
    gl.compileShader(shader);  
  
    if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {  
        alert("Error compiling shader" +  
            gl.getShaderInfoLog(shader));  
        gl.deleteShader(shader);  
        return null;  
    }  
    return shader;  
}
```

Linking the Shaders

```
function setupShaders() {  
    ...  
    shaderProgram = gl.createProgram();  
    gl.attachShader(shaderProgram, vertexShader);  
    gl.attachShader(shaderProgram, fragmentShader);  
    gl.linkProgram(shaderProgram);  
  
    if (!gl.getProgramParameter(shaderProgram,  
        gl.LINK_STATUS)) {  
        alert("Failed to setup shaders");  
    }  
  
    gl.useProgram(shaderProgram);  
  
    shaderProgram.vertexPositionAttribute =  
        gl.getAttribLocation(shaderProgram,  
            "aVertexPosition");  
}
```

We create a program object and attach the compiled shaders and link. At this point, we have a complete shader program that WebGL can use.

attributes are user-defined variables that contain data specific to a vertex.

The **attributes** used in the vertex shader are bound to an index (basically a number given to a slot). Our code needs to know the index associated with the attributes we use in the shader so that our draw function can feed the data correctly.

vertexPositionAttribute is a user-defined property in which we remember the index value

Setting up the Vertex Buffers

```
function setupBuffers() {  
  vertexBuffer = gl.createBuffer();  
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);  
  var triangleVertices = [  
    0.0, 0.5, 0.0,  
    -0.5, -0.5, 0.0,  
    0.5, -0.5, 0.0  
  ];  
  gl.bufferData(gl.ARRAY_BUFFER,  
    new Float32Array(triangleVertices),  
    gl.STATIC_DRAW);  
  vertexBuffer.itemSize = 3;  
  vertexBuffer.numberOfItems = 3;  
}
```

We next need to create a buffer that will hold the vertex data...this is the geometric data of the shapes we wish to render.

We create a WebGL buffer object and bind it so that WebGL knows it is the current buffer to work with.

triangleVertices is a user-defined JavaScript array containing the 3D coordinates of a single triangle.

We call gl.bufferData to copy the vertex positions into the current WebGL buffer.

Two user-defined properties are used to remember how many vertices we have and how many coordinates per vertex.

Drawing the Scene

```
function draw() {  
    gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);  
  
    gl.clear(gl.COLOR_BUFFER_BIT);  
  
    gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,  
                           vertexBuffer.itemSize, gl.FLOAT, false, 0, 0);  
  
    gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);  
  
    gl.drawArrays(gl.TRIANGLES, 0, vertexBuffer.numberOfItems);  
}
```

The ***viewport*** method lets us tell WebGL how to convert from *clipspace* in which coordinates range from -1 to 1 into pixel coordinates.

Here we use our two user-defined properties to set it to the full size of the canvas.

clear initializes the color buffer to the color set with ***clearColor***.

We then tell WebGL to take values for aVertexPosition from the buffer currently bound to gl.ARRAY_BUFFER....

And then we draw....meaning the data from buffers bound to shader attributes will be sent to the GPU to be processed and we'll see the pixel values they produce.

...a little more about attributes

```
shaderProgram.vertexPositionAttribute =  
gl.getAttribLocation(shaderProgram,  
"aVertexPosition");
```

- What is happening here?

```
gl.vertexAttribPointer(  
shaderProgram.vertexPositionAttribute,  
vertexBuffer.itemSize, gl.FLOAT, false, 0,  
0);
```

- What is happening here?

```
gl.enableVertexAttribArray(  
shaderProgram.vertexPositionAttribute);
```

- And here?

...a little more about attributes

```
shaderProgram.vertexPositionAttribute =  
gl.getAttribLocation(shaderProgram,  
"aVertexPosition");
```

```
gl.vertexAttribPointer(  
shaderProgram.vertexPositionAttribute,  
vertexBuffer.itemSize, gl.FLOAT, false, 0,  
0);
```

```
gl.enableVertexAttribArray(  
shaderProgram.vertexPositionAttribute);
```

- Get an index for a variable from the shader program and remember it in a property tied to the shader program object
- Set size, type, etc, of data for the attribute
- Bind the attribute to the data in the buffer

...a little more about attributes

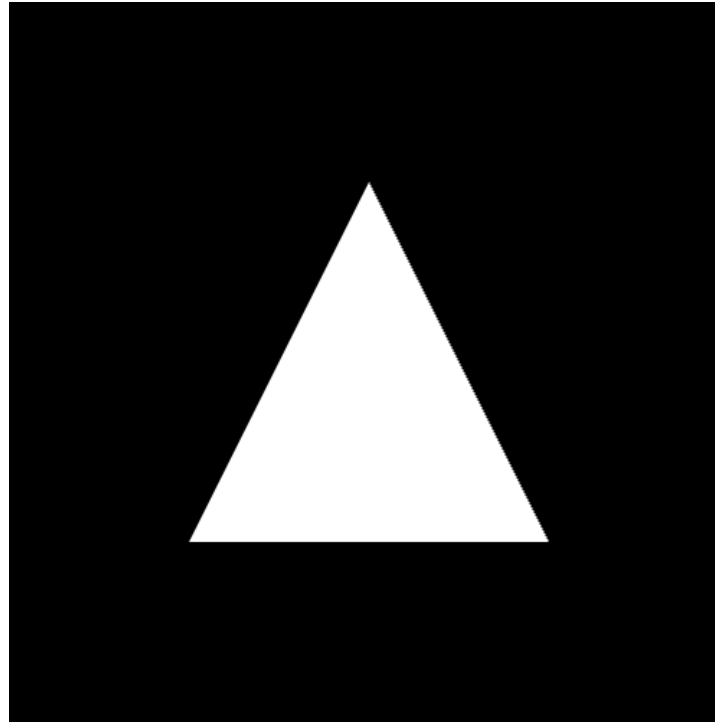
Your shader program can have multiple attribute variables

You need to bind each one to a buffer

For example:

```
// Activate the model's vertex Buffer Object  
gl.bindBuffer(gl.ARRAY_BUFFER, triangles_vertex_buffer_id);  
  
// Bind the vertices Buffer Object to the 'a_Vertex' shader variable  
gl.vertexAttribPointer(a_Vertex_location, 3, gl.FLOAT, false, 0, 0);  
gl.enableVertexAttribArray(a_Vertex_location);  
  
// Activate the model's color Buffer Object  
gl.bindBuffer(gl.ARRAY_BUFFER, triangles_color_buffer_id);  
  
// Bind the color Buffer Object to the 'a_Color' shader variable  
gl.vertexAttribPointer(a_Color_location, 3, gl.FLOAT, false, 0, 0);  
gl.enableVertexAttribArray(a_Color_location);
```

Result...



Can You?

Change the triangle color?

Change the background color?

Change the triangle shape?

Draw multiple triangles?

Figure out what the coordinate system of the canvas is?

Where is the origin?

What are the coordinates of the bottom left corner?

What are the coordinates of the top right corner?