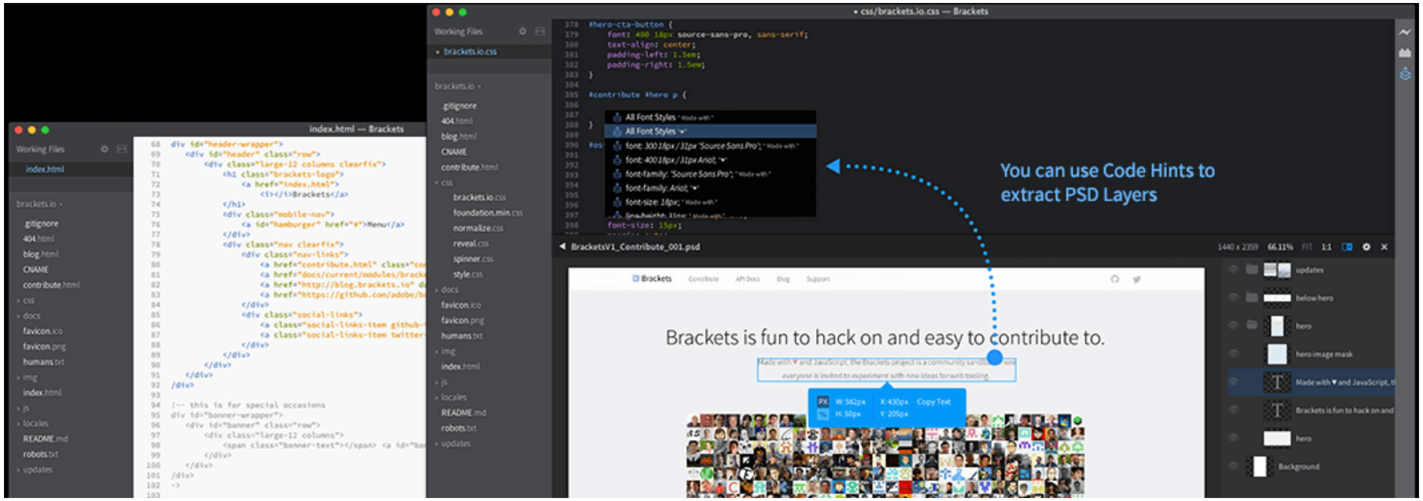
CS 418: Interactive Computer Graphics

Introduction to WebGL: HelloTriangle.html

Eric Shaffer

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…
- You can grab code from

https://courses.engr.illinois.edu/cs418/Examples/HelloTriangle.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

Notice:
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 createGLContext(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);

...

}
```

```
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;
}
```

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.

Creating the Program Object and 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 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 a magic function 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

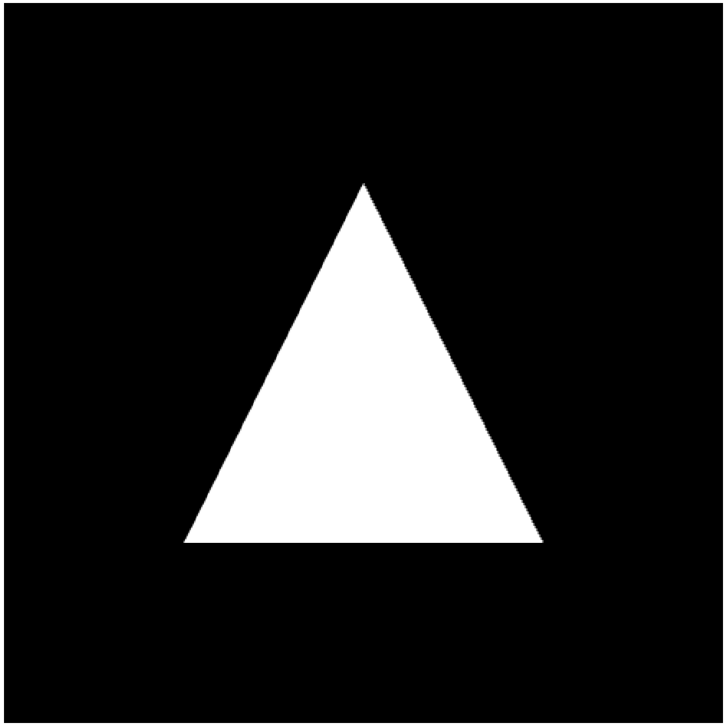
```
function draw() {
  gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);

  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 back 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.

Result…



Can you?

- Change the triangle color?
- Change the background color?
- Change the triangle shape?
- Draw multiple triangles?
- Make the triangle look smaller without changing the vertex data?