

WebGL

Khronos: WebGL Specification

I would highly recommend grabbing the Khronos reference card for WebGL, and any other related APIs you might be interested in. This WebGL reference is very useful, but unfortunately it's gated behind a 30-day trial leading to subscription to Scirbd. This is the official source and the only way I know of to get it, and it's not my place to redistribute this material, so [here's the official link](#). I just subscribed with paypal, downloaded my PDFs (all Khronos references), then canceled my account immediately. Scribd might actually be useful if I had more leisure time to read, but I don't so for now it's a hard pass.

WebGL Fundamentals

If you are familiar with OpenGL already, I would recommend reading through [drawing a simple texture on 3D geometry](#). You will see a lot of familiar concepts are used in the JavaScript API when compared to OpenGL in C++.

Within an HTML page, you can wrap JavaScript within `<script>` tags and directly access OpenGL API and write code similar to what you'd see in C++ using OpenGL. Creating a simple context within a single `index.html` page would look like this -

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Title</title>

  <style>
    .test {
      background-color: gray;
    }
    canvas {
      background-color: gray;
      width: 50;
      height: 50;
      display: block;
    }
  </style>
</head>
<body>
  <div class="test">
    <canvas>
    </canvas>
  </div>
</body>
</html>
```

```

</style>
</head>
<body>

<!-- Shader source code -->
<!-- We don't compile these yet, but this is an example of one way to write your shader code within an HTML
document -->
<script id="vertex-simple-shader" type="x-shader/x-vertex">
attribute vec2 a_position;
uniform vec2 u_resolution;

void main() {
    // Convert pixel coordinates to a float ranging from 0.0 -> 1.0
    vec2 zeroToOne = a_position / u_resolution;
    // Convert from 0->1 to 0->2
    vec2 zeroToTwo = zeroToOne * 2.0;
    // Convert from 0->2 to -1.0->1.0
    vec2 clipSpace = zeroToTwo - 1.0;

    gl_Position = vec4(clipSpace * vec2(1, -1), 0, 1);
}
</script>
<script id="fragment-simple-shader" type="x-shader/x-vertex">
precision mediump float;
uniform vec4 u_color;

void main() {
    gl_FragColor = u_color;
}
</script>

<!-- This is all the HTML required to create an OpenGL canvas. The rest will be done in JS -->
<p class="test">This is test HTML with some CSS. Logging with JS...</br>The following block is an OpenGL
canvas</p>
<canvas id="canvas"></canvas>

<!-- OpenGL / WebGL context creation -->
<script>
    function main() {
        let canvas = document.querySelector("#canvas");

```

```
let gl = canvas.getContext('webgl');
if (!gl) {
  console.log("ERROR: Unable to get OpenGL context");
  return;
}
else {
  console.log("Created OpenGL context on HTML canvas")
}
}
main();
</script>

</body>
</html>
```

The rest of this page will cover some more advanced WebGL as I get more familiar coming from OpenGL in C++. These are just my personal notes, mostly from following tutorials at [WebGL Fundamentals](#). Some snippets from this site may appear here.

WebGL Rendering

To actually render to an HTML canvas using WebGL, we need a few things first.

1. Valid OpenGL context within an HTML web page (we have this already!)
2. Compiled shader source code (we also have shader code, but haven't compiled yet)
3. OpenGL Shader Program
4. Defined geometry
5. Call to an OpenGL draw method
6. Animation callback method (optional for static geometry)

Creating Programs

To render using OpenGL, we need to create a shader program. To do this we need to compile our shader source code and link them together into a shader program. Usually the two shader types used are Vertex and Fragment shaders. You can use other combinations for different reasons, but for this simple example we will only need these two types of shaders.

The Vertex shader is a geometry shader that handles vertex position, and the Fragment shader is used for coloring and blending between these vertices. The vertex shader passes information to the Fragment shader, and we will have to keep this in mind to understand the code.

If you haven't already, I *highly recommend* looking at the Khronos WebGL reference cards that I link to at the top of this page. These references contain lots of useful at-a-glance information regarding GLSL shader code, and might help you get up to speed quickly.

To start, this section assumes you have the following code -

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Title</title>

  <style>
    .test {
      background-color: gray;
    }
    canvas {
      background-color: gray;
      width: 50;
      height: 50;
      display: block;
    }
  </style>
</head>
<body>

<!-- Simple shader source code (WebGL Fundamentals) -->
<!-- We don't compile these yet, but this is an example of one way to write your shader code within an HTML
document -->
<script id="vertex-simple-shader" type="x-shader/x-vertex">
attribute vec2 a_position;
uniform vec2 u_resolution;

void main() {
  // Convert pixel coordinates to a float ranging from 0.0 -> 1.0
  vec2 zeroToOne = a_position / u_resolution;
  // Convert from 0->1 to 0->2
  vec2 zeroToTwo = zeroToOne * 2.0;
  // Convert from 0->2 to -1.0->1.0
  vec2 clipSpace = zeroToTwo - 1.0;
```

```

    gl_Position = vec4(clipSpace * vec2(1, -1), 0, 1);
}
</script>
<script id="fragment-simple-shader" type="x-shader/x-vertex">
precision mediump float;
uniform vec4 u_color;

void main() {
    gl_FragColor = u_color;
}
</script>

```

```

<!-- This is all the HTML required to create an OpenGL canvas. The rest will be done in JS -->
<p class="test">This is test HTML with some CSS. Logging with JS...</br>The following block is an OpenGL
canvas</p>
<canvas id="canvas"></canvas>

```

```

<!-- WebGL context -->
<script>
    function main() {
        let canvas = document.querySelector("#canvas");
        let gl = canvas.getContext('webgl');
        if (!gl) {
            console.log("ERROR: Unable to get OpenGL context");
            return;
        }
        else {
            console.log("Created OpenGL context on HTML canvas")
        }
    }
    main();
</script>

</body>
</html>

```

You can see that we have stored GLSL shader source code in the HTML document by using `<script>` blocks. **Notice the `id` and `type` HTML attribute values for `<script>` blocks containing shader source code.** These are important and will be used later to fetch this source code so we can compile the shaders.

To begin creating our shader program, we'll modify the `WebGL context` section of the code. We first add some helper functions that will simplify compiling shaders into shader programs.

The `createShader` function will compile a single shader and return a `WebGLShader`, and the `createProgram` function will link two `WebGLShader` objects into a single `WebGLProgram`. After defining these two helper functions, we create our the `WebGLProgram` for our current context with a single call to `createProgram`. Once this is done, we'll be ready to start using our shaders to render to the canvas.

```
<!-- WebGL context -->
<script>
function main() {
  //
  // Boilerplate WebGL helper functions

  // Create a shader given shader type and source code text
  function createShader(gl, type, source) {
    let shader = gl.createShader(type);
    gl.shaderSource(shader, source);
    gl.compileShader(shader);
    let success = gl.getShaderParameter(shader, gl.COMPILE_STATUS);
    if (success) {
      return shader;
    }

    // TODO: Does this happen when the shader compilation fails?
    console.log(gl.getShaderInfoLog(shader));
    gl.deleteShader(shader);
  }

  // Create a shader program using vertex and fragment shader
  function createProgram(gl, vertexShader, fragmentShader) {
    // Check to automatically compile shaders if Strings are provided
    // + Strings should be document.querySelector syntax targeting the HTML id for the shader's script block
    if (typeof vertexShader == 'string') {
      let vertexShaderSource = document.querySelector(vertexShader).text;
      vertexShader = createShader(gl, gl.VERTEX_SHADER, vertexShaderSource);
    }
    if (typeof fragmentShader == 'string') {
      let fragmentShaderSource = document.querySelector(fragmentShader).text;
      fragmentShader = createShader(gl, gl.FRAGMENT_SHADER, fragmentShaderSource);
    }
  }
}
```

```

    }

    let program = gl.createProgram();
    gl.attachShader(program, vertexShader);
    gl.attachShader(program, fragmentShader);
    gl.linkProgram(program);
    let success = gl.getProgramParameter(program, gl.LINK_STATUS);
    if (success) {
        return program;
    }

    console.log(gl.getProgramInfoLog(program));
    gl.deleteProgram(program);
}

let canvas = document.querySelector("#canvas");
let gl = canvas.getContext('webgl');
if (!gl) {
    console.log("ERROR: Unable to get OpenGL context");
    return;
}
else {
    console.log("Created OpenGL context on HTML canvas")
}

// At this point the Console in your web browser should log the success message above
// + We continue by using this context to load and compile our shader code into a shader program

// Create our WebGL shader program using document.querySelector parameter syntax to target vertex and
fragment shaders
let program = createProgram(gl, "#vertex-simple-shader", "#fragment-simple-shader");
}
main();

</script>

```

That's it! For this example there's no rendering happening yet, but we're one step closer to actually drawing to the canvas. In the next section, we'll talk about vertex attributes, buffers, and how they're used to pass data between our shader program and OpenGL context.

Vertex Attributes

Now that we have a shader program, we're need to load some data into it so the shaders can be used to actually render something to the canvas.

First, let's take the same single-file approach to writing this WebGL program. Up to the end of the previous section on Shader Programs, you should have the following code within a single `index.html` document -

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Title</title>

  <style>
    .test {
      background-color: gray;
    }
    canvas {
      background-color: gray;
      width: 100%;
      height: 100%;
      display: block;
    }
  </style>

</head>
<body>
<!-- Simple shader source code (WebGL Fundamentals) -->
<script id="vertex-simple-shader" type="x-shader/x-vertex">
attribute vec2 a_position;
uniform vec2 u_resolution;

void main() {
  // Convert pixel coordinates to a float ranging from 0.0 -> 1.0
  vec2 zeroToOne = a_position / u_resolution;
  // Convert from 0->1 to 0->2
  vec2 zeroToTwo = zeroToOne * 2.0;
  // Convert from 0->2 to -1.0->1.0
  vec2 clipSpace = zeroToTwo - 1.0;
```



```

    gl_Position = vec4(clipSpace * vec2(1, -1), 0, 1);
}
</script>
<script id="fragment-simple-shader" type="x-shader/x-vertex">
precision mediump float;
uniform vec4 u_color;

void main() {
    gl_FragColor = u_color;
}
</script>

```

```

<!-- HTML document -->

```

```

<p class="test">This is test HTML with some CSS. Logging with JS...<br>The following block is an OpenGL
canvas</p>

```

```

<canvas id="canvas"></canvas>

```

```

<!-- WebGL context -->

```

```

<script>

```

```

    function main() {
        //
        // Boilerplate OpenGL helper functions
        function createShader(gl, type, source) {
            let shader = gl.createShader(type);
            gl.shaderSource(shader, source);
            gl.compileShader(shader);
            let success = gl.getShaderParameter(shader, gl.COMPILE_STATUS);
            if (success) {
                return shader;
            }

```

```

            console.log(gl.getShaderInfoLog(shader));
            gl.deleteShader(shader);
        }

```

```

    function createProgram(gl, vertexShader, fragmentShader) {
        // Add check to automatically compile shaders if Strings are provided
        // + Strings should be equal to HTML script id attribute value for vertex and fragment shaders
        if (typeof vertexShader == 'string') {
            let vertexShaderSource = document.querySelector(vertexShader).text;

```

```

        vertexShader = createShader(gl, gl.VERTEX_SHADER, vertexShaderSource);
    }
    if (typeof fragmentShader === 'string') {
        let fragmentShaderSource = document.querySelector(fragmentShader).text;
        fragmentShader = createShader(gl, gl.FRAGMENT_SHADER, fragmentShaderSource);
    }

    let program = gl.createProgram();
    gl.attachShader(program, vertexShader);
    gl.attachShader(program, fragmentShader);
    gl.linkProgram(program);
    let success = gl.getProgramParameter(program, gl.LINK_STATUS);
    if (success) {
        return program;
    }

    console.log(gl.getProgramInfoLog(program));
    gl.deleteProgram(program);
}

let canvas = document.querySelector("#canvas");
let gl = canvas.getContext('webgl');
if (!gl) {
    console.log("ERROR: Unable to get OpenGL context");
    return;
}
else {
    console.log("Created OpenGL context on HTML canvas")
}

// Creating shader program and compiling shader source code
let program = createProgram(gl, "#vertex-simple-shader", "#fragment-simple-shader");
}
main();
</script>

</body>
</html>

```

From this point on, we will be working within the final `<script>` block marked with `WebGL context` with a comment. If you are following along, be sure to place your new code in this block. This is just to avoid me having to repaste the same code several times, and to save you from having to scroll through it all each time it reappears.

So, starting at this point we have our shader program compiled and ready to go, now we need to define some geometry data that will define a shape to draw. In this case, we will draw the popular OpenGL RGB triangle, except this time on a web page in WebGL!

```
<!-- WebGL context -->
<script>
  function main() {
    //
    // Boilerplate OpenGL helper functions
    function createShader(gl, type, source) {
      let shader = gl.createShader(type);
      gl.shaderSource(shader, source);
      gl.compileShader(shader);
      let success = gl.getShaderParameter(shader, gl.COMPILE_STATUS);
      if (success) {
        return shader;
      }

      console.log(gl.getShaderInfoLog(shader));
      gl.deleteShader(shader);
    }

    function createProgram(gl, vertexShader, fragmentShader) {
      // Add check to automatically compile shaders if Strings are provided
      // + Strings should be equal to HTML script id attribute value for vertex and fragment shaders
      if (typeof vertexShader == 'string') {
        let vertexShaderSource = document.querySelector(vertexShader).text;
        vertexShader = createShader(gl, gl.VERTEX_SHADER, vertexShaderSource);
      }
      if (typeof fragmentShader == 'string') {
        let fragmentShaderSource = document.querySelector(fragmentShader).text;
        fragmentShader = createShader(gl, gl.FRAGMENT_SHADER, fragmentShaderSource);
      }

      let program = gl.createProgram();
      gl.attachShader(program, vertexShader);
```

```
gl.attachShader(program, fragmentShader);
gl.linkProgram(program);
let success = gl.getProgramParameter(program, gl.LINK_STATUS);
if (success) {
    return program;
}
```

```
console.log(gl.getProgramInfoLog(program));
gl.deleteProgram(program);
}
```

```
let canvas = document.querySelector("#canvas");
let gl = canvas.getContext('webgl');
if (!gl) {
    console.log("ERROR: Unable to get OpenGL context");
    return;
}
else {
    console.log("Created OpenGL context on HTML canvas")
}
```

// Creating shader program and compiling shader source code

```
let program = createProgram(gl, "#vertex-simple-shader", "#fragment-simple-shader");
```

// look up where the vertex data needs to go.

```
let positionAttributeLocation = gl.getAttribLocation(program, "a_position");
```

// look up uniform locations

```
let resolutionUniformLocation = gl.getUniformLocation(program, "u_resolution");
```

```
let colorUniformLocation = gl.getUniformLocation(program, "u_color");
```

// Create a buffer to put three 2d clip space points in

```
let positionBuffer = gl.createBuffer();
```

// Bind it to ARRAY_BUFFER (think of it as ARRAY_BUFFER = positionBuffer)

```
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);
```

// Canvas setup

// Resize canvas to match client size

```
const width = canvas.clientWidth;
```

```
const height = canvas.clientHeight;
```

```
canvas.width = width;
```

```

canvas.height = height;
gl.viewport(0, 0, gl.canvas.width, gl.canvas.height);
// Clear the canvas
gl.clearColor(0, 0, 0, 0);
gl.clear(gl.COLOR_BUFFER_BIT);

// Tell OpenGL to use our shader program
gl.useProgram(program);
// Enable attribute; Bind gl.ARRAY_BUFFER for use with this attribute
gl.enableVertexAttribArray(positionAttributeLocation);
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);

// Tell the attribute how to get data out of positionBuffer (ARRAY_BUFFER)
let size = 2;      // 2 components per iteration (X and Y value for each vertex position)
let type = gl.FLOAT; // Each X and Y value is a 32bit float
let normalize = false; // don't normalize the data
let stride = 0;     // 0 = move forward size * sizeof(type) each iteration to get the next position
let offset = 0;     // start at the beginning of the buffer
gl.vertexAttribPointer(positionAttributeLocation, size, type, normalize, stride, offset);

// Initialize geometry data for a 2D triangle with 3 vertices
let triangle = new Float32Array([
    350, 100,
    500, 300,
    200, 300,
]);
// Write geometry data to positions array buffer
gl.bufferData(gl.ARRAY_BUFFER, triangle, gl.STATIC_DRAW);
// Set a random color
gl.uniform4f(colorUniformLocation, Math.random(), Math.random(), Math.random(), 1);

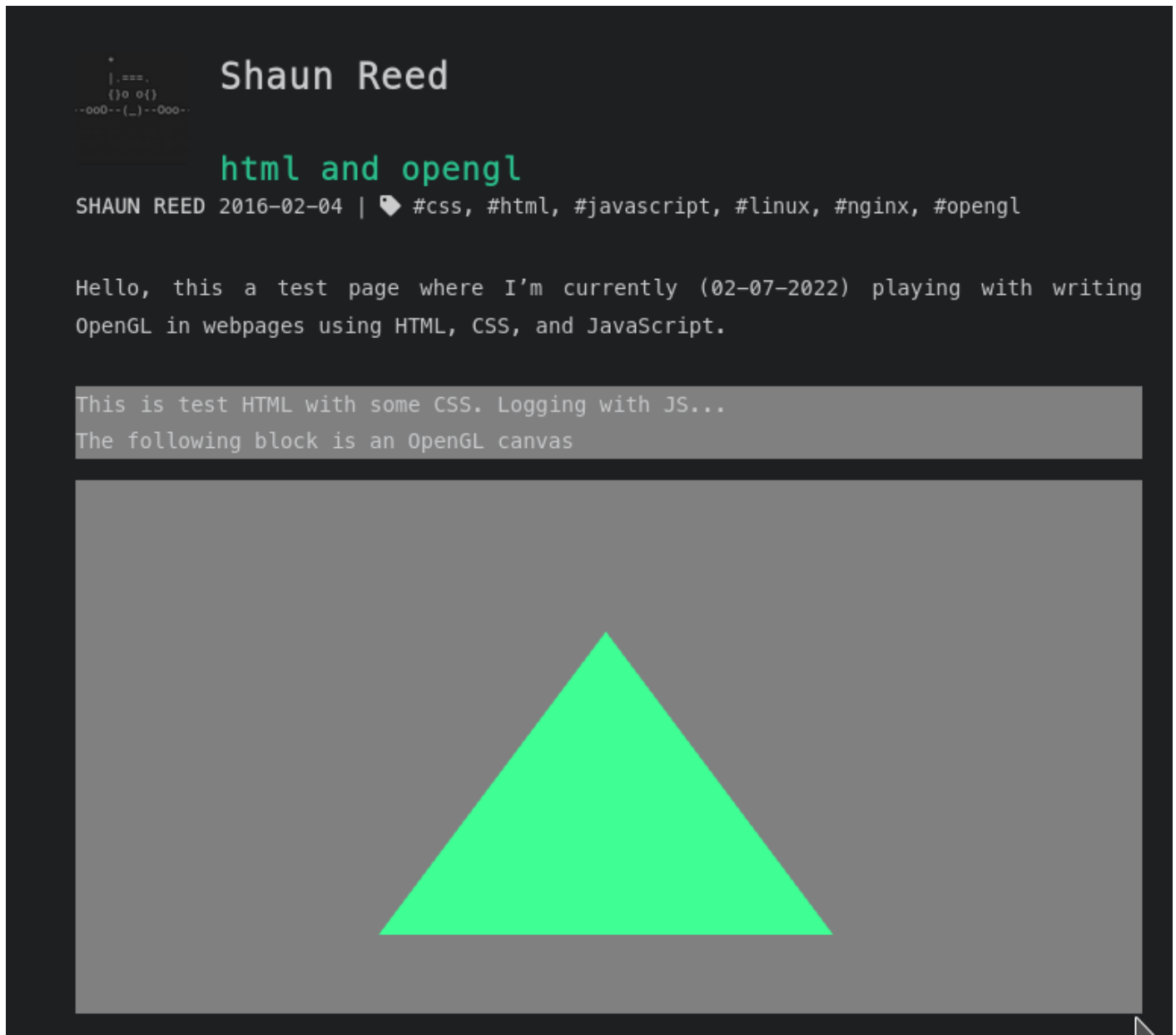
// set the resolution
gl.uniform2f(resolutionUniformLocation, gl.canvas.width, gl.canvas.height);

// Draw the triangle with 0 offset and 3 total vertices
gl.drawArrays(gl.TRIANGLES, 0, 3);

}
main();
</script>

```

After running this code, we get the following result -



But that doesn't have the RGB color we had in mind. Check the next section to see the changes needed to add RGB interpolation between vertices.

Shaders

At the end of this section, we will have modified our vertex and fragment shaders to support using RGB interpolation between the vertex positions of our triangle. What this means is each point of the triangle can be assigned a color, and the triangle will then be shaded with the blending of the different point colors.

To start, I'll assume you have all of the code from the previous section, but I'll also paste it below just to preserve it as it was when I first wrote this.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Title</title>

  <style>
    .test {
      background-color: gray;
    }
    canvas {
      background-color: gray;
      width: 100%;
      height: 100%;
      display: block;
    }
  </style>

</head>
<body>
<!-- Simple shader source code (WebGL Fundamentals) -->
<script id="vertex-simple-shader" type="x-shader/x-vertex">
attribute vec2 a_position;
uniform vec2 u_resolution;

void main() {
  // Convert pixel coordinates to a float ranging from 0.0 -> 1.0
  vec2 zeroToOne = a_position / u_resolution;
  // Convert from 0->1 to 0->2
  vec2 zeroToTwo = zeroToOne * 2.0;
  // Convert from 0->2 to -1.0->1.0
  vec2 clipSpace = zeroToTwo - 1.0;

  gl_Position = vec4(clipSpace * vec2(1, -1), 0, 1);
}
</script>
<script id="fragment-simple-shader" type="x-shader/x-vertex">
precision mediump float;
uniform vec4 u_color;
```

```

void main() {
    gl_FragColor = u_color;
}
</script>

<!-- HTML document -->
<p class="test">This is test HTML with some CSS. Logging with JS...<br>The following block is an OpenGL
canvas</p>
<canvas id="canvas"></canvas>

<!-- WebGL -->
<script>
    function main() {
        //
        // Boilerplate OpenGL helper functions
        function createShader(gl, type, source) {
            let shader = gl.createShader(type);
            gl.shaderSource(shader, source);
            gl.compileShader(shader);
            let success = gl.getShaderParameter(shader, gl.COMPILE_STATUS);
            if (success) {
                return shader;
            }

            console.log(gl.getShaderInfoLog(shader));
            gl.deleteShader(shader);
        }

        function createProgram(gl, vertexShader, fragmentShader) {
            // Add check to automatically compile shaders if Strings are provided
            // + Strings should be equal to HTML script id attribute value for vertex and fragment shaders
            if (typeof vertexShader == 'string') {
                let vertexShaderSource = document.querySelector(vertexShader).text;
                vertexShader = createShader(gl, gl.VERTEX_SHADER, vertexShaderSource);
            }
            if (typeof fragmentShader == 'string') {
                let fragmentShaderSource = document.querySelector(fragmentShader).text;
                fragmentShader = createShader(gl, gl.FRAGMENT_SHADER, fragmentShaderSource);
            }

```



```

    let program = gl.createProgram();
    gl.attachShader(program, vertexShader);
    gl.attachShader(program, fragmentShader);
    gl.linkProgram(program);
    let success = gl.getProgramParameter(program, gl.LINK_STATUS);
    if (success) {
        return program;
    }

    console.log(gl.getProgramInfoLog(program));
    gl.deleteProgram(program);
}

let canvas = document.querySelector("#canvas");
let gl = canvas.getContext('webgl');
if (!gl) {
    console.log("ERROR: Unable to get OpenGL context");
    return;
}
else {
    console.log("Created OpenGL context on HTML canvas")
}

// Creating shader program and compiling shader source code
let program = createProgram(gl, "#vertex-simple-shader", "#fragment-simple-shader");

// look up where the vertex data needs to go.
let positionAttributeLocation = gl.getAttribLocation(program, "a_position");
// look up uniform locations
let resolutionUniformLocation = gl.getUniformLocation(program, "u_resolution");
let colorUniformLocation = gl.getUniformLocation(program, "u_color");

// Create a buffer to put three 2d clip space points in
let positionBuffer = gl.createBuffer();
// Bind it to ARRAY_BUFFER (think of it as ARRAY_BUFFER = positionBuffer)
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);

// Canvas setup
// Resize canvas to match client size

```

```

const width = canvas.clientWidth;
const height = canvas.clientHeight;
canvas.width = width;
canvas.height = height;
gl.viewport(0, 0, gl.canvas.width, gl.canvas.height);
// Clear the canvas
gl.clearColor(0, 0, 0, 0);
gl.clear(gl.COLOR_BUFFER_BIT);

// Tell OpenGL to use our shader program
gl.useProgram(program);
// Enable attribute; Bind gl.ARRAY_BUFFER for use with this attribute
gl.enableVertexAttribArray(positionAttributeLocation);
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);

// Tell the attribute how to get data out of positionBuffer (ARRAY_BUFFER)
let size = 2; // 2 components per iteration (X and Y value for each vertex position)
let type = gl.FLOAT; // Each X and Y value is a 32bit float
let normalize = false; // don't normalize the data
let stride = 0; // 0 = move forward size * sizeof(type) each iteration to get the next position
let offset = 0; // start at the beginning of the buffer
gl.vertexAttribPointer(positionAttributeLocation, size, type, normalize, stride, offset);

// Initialize geometry data for a 2D triangle with 3 vertices
let triangle = new Float32Array([
    350, 100,
    500, 300,
    200, 300,
]);

// Write geometry data to positions array buffer
gl.bufferData(gl.ARRAY_BUFFER, triangle, gl.STATIC_DRAW);
// Set a random color
gl.uniform4f(colorUniformLocation, Math.random(), Math.random(), Math.random(), 1);
// set the resolution
gl.uniform2f(resolutionUniformLocation, gl.canvas.width, gl.canvas.height);

// Draw the triangle with 0 offset and 3 total vertices
gl.drawArrays(gl.TRIANGLES, 0, 3);

```

```
}  
  main();  
</script>  
  
</body>  
</html>
```

Using this code as a starting point, we will make the following changes..

Sorry! This page is a WIP. Check back later for updates.

Revision #19

Created 7 February 2022 16:51:27 by Shaun Reed

Updated 8 February 2022 07:01:00 by Shaun Reed