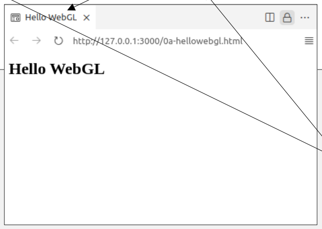
WebGL Program Elements

Steps – (JavaScript unless otherwise specified)

- create web page (HTML) containing a canvas for display
- set up JavaScript program structure
- get a WebGL context for that canvas
- define the vertex and fragment shaders (GLSL)
- compile and link shaders into a program
- set up shader arguments
  - get locations for shader arguments
  - define VBOs (attributes)
  - create typed array with the data (attributes)
  - connect JavaScript-defined values to GPU storage
- draw the scene

Web Page Structure

```
<html>

<head>
  <title>Hello WebGL</title>

<body>
  <h2>Hello WebGL</h2>

  <div id="canvas-holder">
    <canvas width="500" height="500" id="webglcanvas"></canvas>
  </div>
</body>

</html>
```



Hello WebGL

http://127.0.0.1:3000/0a-hellowebgl.html

Hello WebGL

title is shown as the window title

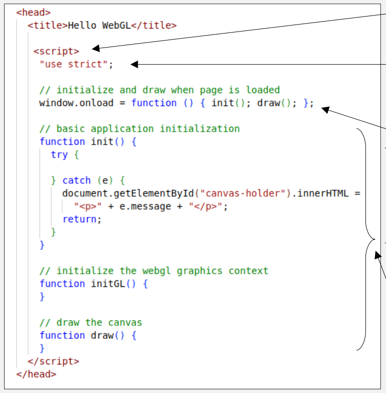
body contains what is visible when viewed in a browser

canvas defines the WebGL drawing area

div defines an area used to display WebGL loading/init error messages

div and canvas ids are unique ids used to reference those elements from javascript

JavaScript Program Structure



```
<head>
  <title>Hello WebGL</title>

   <script>
    "use strict";

    // initialize and draw when page is loaded
    window.onload = function () { init(); draw(); };

    // basic application initialization
    function init() {
      try {

      } catch (e) {
        document.getElementById("canvas-holder").innerHTML =
          "<p>" + e.message + "</p>";
        return;
      }
    }

    // initialize the webgl graphics context
    function initGL() {
    }

    // draw the canvas
    function draw() {
    }
  </script>
</head>
```

JavaScript goes inside <script> tags in the head section

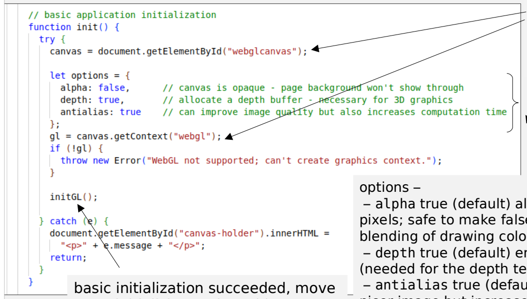
use strict disallows certain “sloppy syntax”, helping to prevent errors – use it!

window.onload defines what happens when the page is loaded (an anonymous function is convenient)

error handling – if anything goes wrong with the basic initialization, display the error message in the element with id canvas-holder defined in the HTML document

init(), initGL(), draw() functions are a convenient way to organize the JavaScript

Getting a WebGL Graphics Context

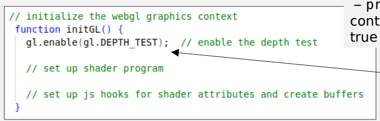


```
// basic application initialization
function init() {
  try {
    canvas = document.getElementById("webglcanvas");

    let options = {
      alpha: false,      // canvas is opaque - page background won't show through
      depth: true,       // allocate a depth buffer - necessary for 3D graphics
      antialias: true    // can improve image quality but also increases computation time
    };
    gl = canvas.getContext("webgl");
    if (!gl) {
      throw new Error("WebGL not supported; can't create graphics context.");
    }

    initGL();

  } catch (e) {
    document.getElementById("canvas-holder").innerHTML =
      "<p>" + e.message + "</p>";
    return;
  }
}
```

webglcanvas and canvas-holder are the ids of elements declared in the <body> section

options –
 – alpha true (default) allows transparent canvas pixels; safe to make false (does not block alpha blending of drawing color with image color)
 – depth true (default) enables depth buffer (needed for the depth test in 3D)
 – antialias true (default) requests antialiasing; nicer image but increased computation time
 – preserveDrawingBuffer false (default) discards contents of drawing buffer after display; only need true if drawing is constructed incrementally

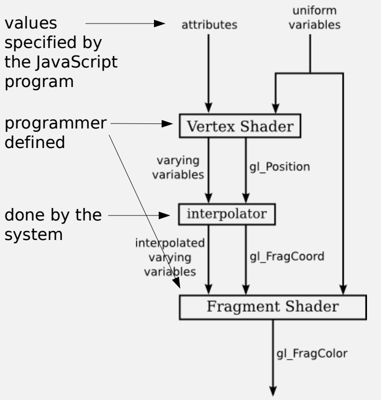
basic initialization succeeded, move on to initializing WebGL things



```
// initialize the webgl graphics context
 function initGL() {
   gl.enable(gl.DEPTH_TEST);  // enable the depth test

   // set up shader program

   // set up js hooks for shader attributes and create buffers
 }
```

enable the depth test (needed for 3D)

Writing Shaders – Pipeline Data Flow



the values for attributes (per vertex) and uniforms (per primitive) are specified in the JavaScript program

vertex shader is called once per vertex

it gets attributes and uniforms as parameters

its job is to compute per-vertex values used in the pipeline – `gl_Position` and anything else (e.g. color)

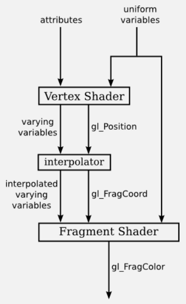
fragment shader is called once per pixel

it gets interpolated values for the per-vertex values computed by the vertex shader (including `gl_Position`) and uniforms (per primitive) as parameters

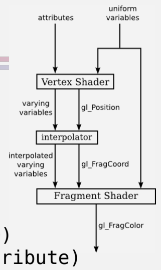
its job is to compute `gl_FragColor`, the color for that pixel

Writing Shaders

- shaders can contain global variable declarations, type definitions, function definitions
  - must include `void main () { … }`
  - global variables are `attribute` (vertex shader only), `uniform`, `varying`
    - varying variables are how information is passed from the vertex shader to the fragment shader
      - vertex shader is responsible for assigning a value
      - interpolator interpolates the value for each pixel based on the vertex values, and passes the interpolated value to the fragment shader
    - varying variables are declared in both vertex and fragment shaders
    - book's convention is to use a_, u_, v_ prefixes to denote attribute, uniform, varying variables
  - local variables lack modifiers



A Simple Vertex Shader

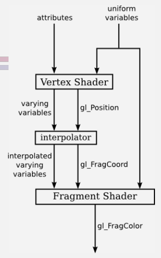


- **vertex shader (GLSL)**
  - **transforms vertex coordinates from OC → CC**
  - **computes values for properties used by fragment shader (e.g. color)**
  - **sets `gl_Position` to the coordinates of the vertex in CC**

```
attribute vec3 a_coords;  // vertex coords (3D)
attribute vec3 a_color;   // vertex color (attribute)
varying vec3 v_color;     // vertex color (result)

void main () {
  gl_Position = vec4(a_coords,1);
  v_color = a_color;
}
```

- modeling, viewing, projection transform are the identity
  - coordinates are already clip coordinates
- vertex color is the assigned color

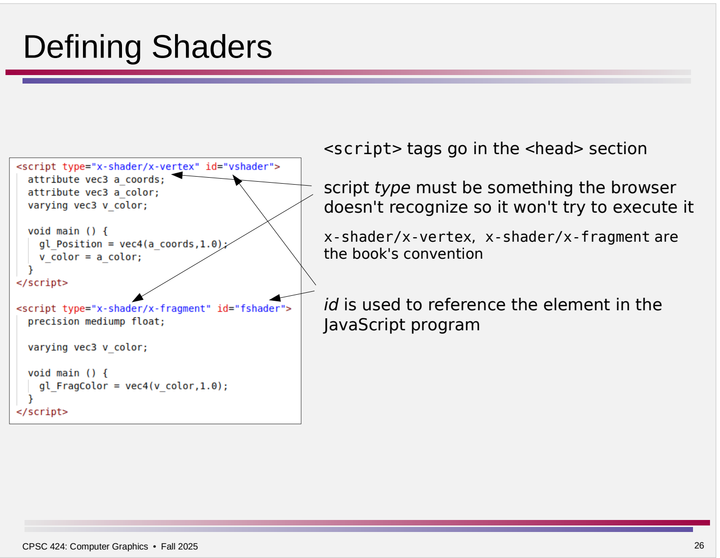
A Simple Fragment Shader



- **fragment shader (GLSL)**
  - **sets `gl_FragColor` to the color of the pixel**

```
precision mediump float;   // set the precision

varying vec3 v_color;      // interpolated color

void main () {
  gl_FragColor = vec4(v_color,1.0);
}
```

- pixel color is the interpolated vertex color



Defining Shaders

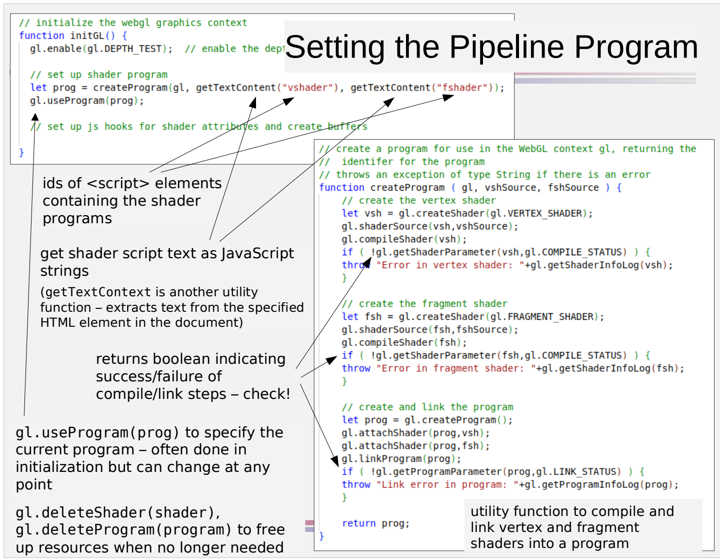
```
<script type="x-shader/x-vertex" id="vshader">
  attribute vec3 a_coords;
  attribute vec3 a_color;
  varying vec3 v_color;

  void main () {
    gl_Position = vec4(a_coords,1.0);
    v_color = a_color;
  }
</script>

<script type="x-shader/x-fragment" id="fshader">
  precision mediump float;

  varying vec3 v_color;

  void main () {
    gl_FragColor = vec4(v_color,1.0);
  }
</script>
```

`<script>` tags go in the <head> section

script *type* must be something the browser doesn't recognize so it won't try to execute it

`x-shader/x-vertex`, `x-shader/x-fragment` are the book's convention

id is used to reference the element in the JavaScript program



Setting the Pipeline Program

```
// initialize the webgl graphics context
function initGL() {
  gl.enable(gl.DEPTH_TEST);  // enable the dep

  // set up shader program
  let prog = createProgram(gl, getTextContent("vshader"), getTextContent("fshader"));
  gl.useProgram(prog);

  // set up js hooks for shader attributes and create buffers

}
```

```
// create a program for use in the WebGL context gl, returning the
//  identifer for the program
// throws an exception of type String if there is an error
function createProgram ( gl, vshSource, fshSource ) {
    // create the vertex shader
    let vsh = gl.createShader(gl.VERTEX_SHADER);
    gl.shaderSource(vsh,vshSource);
    gl.compileShader(vsh);
    if ( !gl.getShaderParameter(vsh,gl.COMPILE_STATUS) ) {
    throw "Error in vertex shader: "+gl.getShaderInfoLog(vsh);
    }

    // create the fragment shader
    let fsh = gl.createShader(gl.FRAGMENT_SHADER);
    gl.shaderSource(fsh,fshSource);
    gl.compileShader(fsh);
    if ( !gl.getShaderParameter(fsh,gl.COMPILE_STATUS) ) {
    throw "Error in fragment shader: "+gl.getShaderInfoLog(fsh);
    }

    // create and link the program
    let prog = gl.createProgram();
    gl.attachShader(prog,vsh);
    gl.attachShader(prog,fsh);
    gl.linkProgram(prog);
    if ( !gl.getProgramParameter(prog,gl.LINK_STATUS) ) {
    throw "Link error in program: "+gl.getProgramInfoLog(prog);
    }

    return prog;
}
```

ids of <script> elements containing the shader programs

get shader script text as JavaScript strings

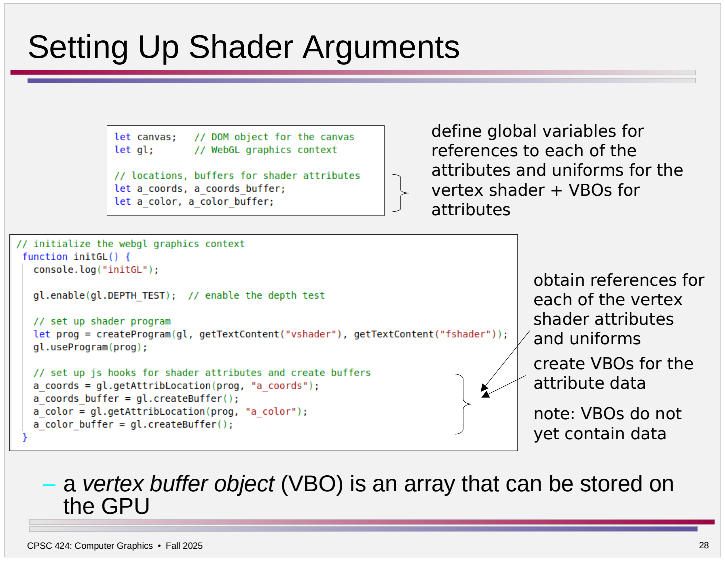
(getTextContext is another utility function – extracts text from the specified HTML element in the document)

returns boolean indicating success/failure of compile/link steps – check!

`gl.useProgram(prog)` to specify the current program – often done in initialization but can change at any point

`gl.deleteShader(shader)`, `gl.deleteProgram(program)` to free up resources when no longer needed

utility function to compile and link vertex and fragment shaders into a program



Setting Up Shader Arguments

```
let canvas;   // DOM object for the canvas
let gl;       // WebGL graphics context

// locations, buffers for shader attributes
let a_coords, a_coords_buffer;
let a_color, a_color_buffer;
```

define global variables for references to each of the attributes and uniforms for the vertex shader + VBOs for attributes

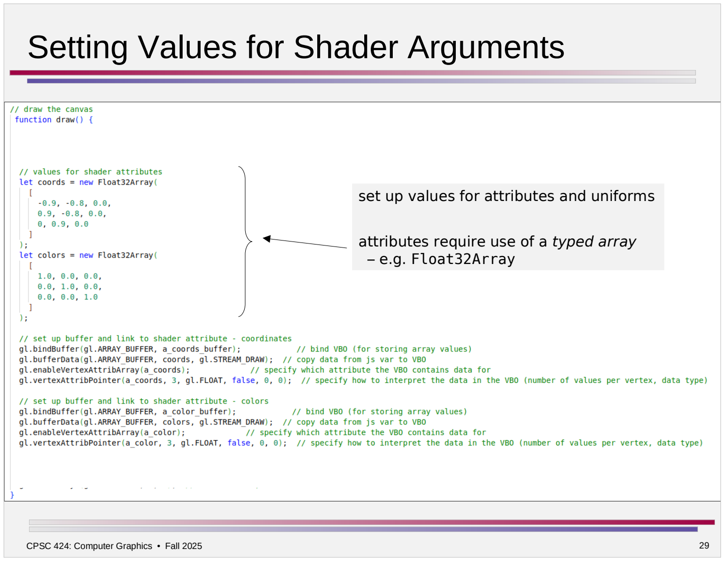
```
// initialize the webgl graphics context
function initGL() {
  console.log("initGL");

  gl.enable(gl.DEPTH_TEST);  // enable the depth test

  // set up shader program
  let prog = createProgram(gl, getTextContent("vshader"), getTextContent("fshader"));
  gl.useProgram(prog);

  // set up js hooks for shader attributes and create buffers
  a_coords = gl.getAttribLocation(prog, "a_coords");
  a_coords_buffer = gl.createBuffer();
  a_color = gl.getAttribLocation(prog, "a_color");
  a_color_buffer = gl.createBuffer();
}
```

obtain references for each of the vertex shader attributes and uniforms

create VBOs for the attribute data

note: VBOs do not yet contain data

– a *vertex buffer object* (VBO) is an array that can be stored on the GPU



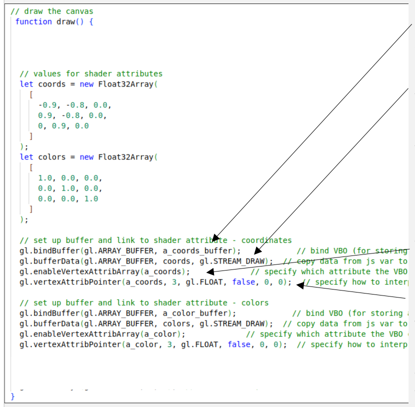
Setting Values for Shader Arguments

```
// draw the canvas
function draw() {



  // values for shader attributes
  let coords = new Float32Array(
    [
      -0.9, -0.8, 0.0,
      0.9, -0.8, 0.0,
      0, 0.9, 0.0
    ]
  );
  let colors = new Float32Array(
    [
      1.0, 0.0, 0.0,
      0.0, 1.0, 0.0,
      0.0, 0.0, 1.0
    ]
  );

  // set up buffer and link to shader attribute - coordinates
  gl.bindBuffer(gl.ARRAY_BUFFER, a_coords_buffer);             // bind VBO (for storing array values)
  gl.bufferData(gl.ARRAY_BUFFER, coords, gl.STREAM_DRAW);  // copy data from js var to VBO
  gl.enableVertexAttribArray(a_coords);             // specify which attribute the VBO contains data for
  gl.vertexAttribPointer(a_coords, 3, gl.FLOAT, false, 0, 0);  // specify how to interpret the data in the VBO (number of values per vertex, data type)

  // set up buffer and link to shader attribute - colors
  gl.bindBuffer(gl.ARRAY_BUFFER, a_color_buffer);             // bind VBO (for storing array values)
  gl.bufferData(gl.ARRAY_BUFFER, colors, gl.STREAM_DRAW);  // copy data from js var to VBO
  gl.enableVertexAttribArray(a_color);             // specify which attribute the VBO contains data for
  gl.vertexAttribPointer(a_color, 3, gl.FLOAT, false, 0, 0);  // specify how to interpret the data in the VBO (number of values per vertex, data type)


}
```

set up values for attributes and uniforms

attributes require use of a *typed array*
– e.g. `Float32Array`

Setting Values for Shader Arguments



```
// draw the canvas
 function draw() {



  // values for shader attributes
  let coords = new Float32Array(
    [
      -0.9, -0.8, 0.0,
      0.9, -0.8, 0.0,
      0, 0.9, 0.0
    ]
  );
  let colors = new Float32Array(
    [
      1.0, 0.0, 0.0,
      0.0, 1.0, 0.0,
      0.0, 0.0, 1.0
    ]
  );

  // set up buffer and link to shader attribute - coordinates
  gl.bindBuffer(gl.ARRAY_BUFFER, a_coords_buffer);          // bind VBO (for storing
  gl.bufferData(gl.ARRAY_BUFFER, coords, gl.STREAM_DRAW);  // copy data from js var to
  gl.enableVertexAttribArray(a_coords);            // specify which attribute the VBO
  gl.vertexAttribPointer(a_coords, 3, gl.FLOAT, false, 0, 0);  // specify how to interp

  // set up buffer and link to shader attribute - colors
  gl.bindBuffer(gl.ARRAY_BUFFER, a_color_buffer);          // bind VBO (for storing
  gl.bufferData(gl.ARRAY_BUFFER, colors, gl.STREAM_DRAW);  // copy data from js var to
  gl.enableVertexAttribArray(a_color);            // specify which attribute the VBO
  gl.vertexAttribPointer(a_color, 3, gl.FLOAT, false, 0, 0);  // specify how to interp



}
```

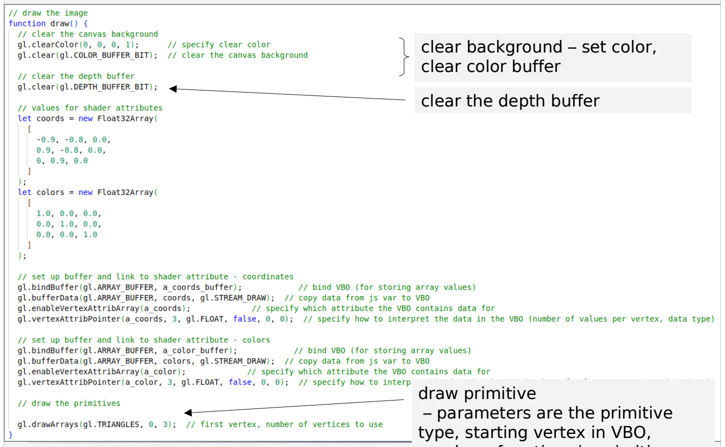
`gl.bindBuffer` specifies how the VBO will be used (ARRAY_BUFFER when storing values for an attribute)

`gl.bufferData` transfers data from typed array to VBO (STATIC_DRAW, STREAM_DRAW, DYNAMIC_DRAW provide hints about how the data will be used so WebGL can manage it efficiently – STATIC_DRAW used many times, STREAM_DRAW used once and discarded)

`gl.enableVertexAttribArray` enables the use of VBO for that attribute (only needs to be done once – could go in `initGL`)

`gl.vertexAttribPointer` takes values for the attribute from the currently-bound buffer
 – parameters are the attribute location, the number of values per vertex, the type of each value, whether to normalize values (convert to range [-1,1]) or use as is, stride (gap between data values in array), offset (to first value to use in array)

Draw the Scene



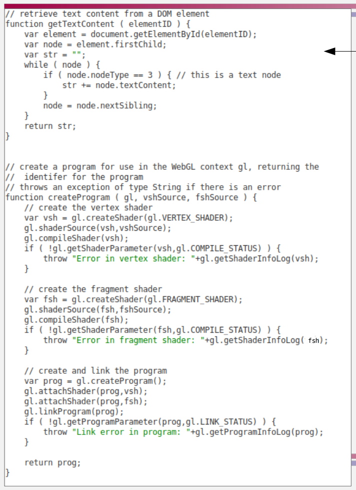
```
// draw the image
function draw() {
  // clear the canvas background
  gl.clearColor(0, 0, 0, 1);      // specify clear color
  gl.clear(gl.COLOR_BUFFER_BIT);  // clear the canvas background

  // clear the depth buffer
  gl.clear(gl.DEPTH_BUFFER_BIT);

  // values for shader attributes
  let coords = new Float32Array(
    [
      -0.9, -0.8, 0.0,
      0.9, -0.8, 0.0,
      0, 0.9, 0.0
    ]
  );
  let colors = new Float32Array(
    [
      1.0, 0.0, 0.0,
      0.0, 1.0, 0.0,
      0.0, 0.0, 1.0
    ]
  );

  // set up buffer and link to shader attribute - coordinates
  gl.bindBuffer(gl.ARRAY_BUFFER, a_coords_buffer);            // bind VBO (for storing array values)
  gl.bufferData(gl.ARRAY_BUFFER, coords, gl.STREAM_DRAW);  // copy data from js var to VBO
  gl.enableVertexAttribArray(a_coords);             // specify which attribute the VBO contains data for
  gl.vertexAttribPointer(a_coords, 3, gl.FLOAT, false, 0, 0);  // specify how to interpret the data in the VBO (number of values per vertex, data type)

  // set up buffer and link to shader attribute - colors
  gl.bindBuffer(gl.ARRAY_BUFFER, a_color_buffer);            // bind VBO (for storing array values)
  gl.bufferData(gl.ARRAY_BUFFER, colors, gl.STREAM_DRAW);  // copy data from js var to VBO
  gl.enableVertexAttribArray(a_color);             // specify which attribute the VBO contains data for
  gl.vertexAttribPointer(a_color, 3, gl.FLOAT, false, 0, 0);  // specify how to interp

  // draw the primitives

  gl.drawArrays(gl.TRIANGLES, 0, 3);  // first vertex, number of vertices to use
}
```

clear background – set color, clear color buffer

clear the depth buffer

draw primitive
 – parameters are the primitive type, starting vertex in VBO, number of vertices in primitive

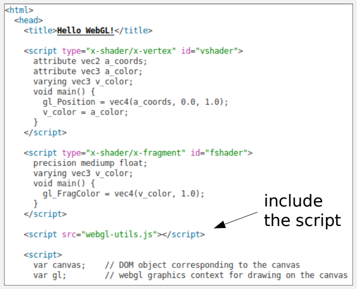
Pulling Out Common Elements



```
// retrieve text content from a DOM element
function getTextContent ( elementID ) {
    var element = document.getElementById(elementID);
    var node = element.firstChild;
    var str = "";
    while ( node ) {
        if ( node.nodeType == 3 ) { // this is a text node
            str += node.textContent;
        }
        node = node.nextSibling;
    }
    return str;
}


// create a program for use in the WebGL context gl, returning the
//  identifer for the program
// throws an exception of type String if there is an error
function createProgram ( gl, vshSource, fshSource ) {
    // create the vertex shader
    var vsh = gl.createShader(gl.VERTEX_SHADER);
    gl.shaderSource(vsh,vshSource);
    gl.compileShader(vsh);
    if ( !gl.getShaderParameter(vsh,gl.COMPILE_STATUS) ) {
        throw "Error in vertex shader: "+gl.getShaderInfoLog(vsh);
    }

    // create the fragment shader
    var fsh = gl.createShader(gl.FRAGMENT_SHADER);
    gl.shaderSource(fsh,fshSource);
    gl.compileShader(fsh);
    if ( !gl.getShaderParameter(fsh,gl.COMPILE_STATUS) ) {
        throw "Error in fragment shader: "+gl.getShaderInfoLog( fsh);
    }

    // create and link the program
    var prog = gl.createProgram();
    gl.attachShader(prog,vsh);
    gl.attachShader(prog,fsh);
    gl.linkProgram(prog);
    if ( !gl.getProgramParameter(prog,gl.LINK_STATUS) ) {
        throw "Link error in program: "+gl.getProgramInfoLog(prog);
    }

    return prog;
}
```

common utility functions can go into a separate file (in the same directory)

note no <script> tags, just JavaScript content



```
<html>
  <head>
    <title>Hello WebGL!</title>

    <script type="x-shader/x-vertex" id="vshader">
      attribute vec2 a_coords;
      attribute vec3 a_color;
      varying vec3 v_color;
      void main() {
        gl_Position = vec4(a_coords, 0.0, 1.0);
        v_color = a_color;
      }
    </script>

    <script type="x-shader/x-fragment" id="fshader">
      precision mediump float;
      varying vec3 v_color;
      void main() {
        gl_FragColor = vec4(v_color, 1.0);
      }
    </script>

    <script src="webgl-utils.js"></script>

    <script>
      var canvas;    // DOM object corresponding to the canvas
      var gl;        // webgl graphics context for drawing on the canvas
```

include the script

Error Checking

- in WebGL errors are typically signaled by setting an error code rather than throwing exceptions
  - check with `gl.getError()`
  - `gl.NO_ERROR` indicates success, anything else indicates an error
    - `gl.INVALID_ENUM` – bad primitive type
    - `gl.INVALID_VALUE` – bad value
    - `gl.INVALID_OPERATION` – shader not set
    - `gl.INVALID_STATE` – vertex attrib array enabled, but no data
- notes
  - error code stays set until cleared by `gl.getError()`, even if other successful operations complete
- usage
  - log to browser console with
    `console.log(“Error code is “+gl.getError());`
    - also check browser console for other errors that may be logged there
  - employ `gl.getError()` to locate problem when there's an issue rather than exhaustively error-checking