CSE160 – Lecture 2

- Linear Algebra Review
- Survey
- About GL WebGL
- GLSL Shader Overview
- Assignment 0 and 1
- Administrative
- Q&A

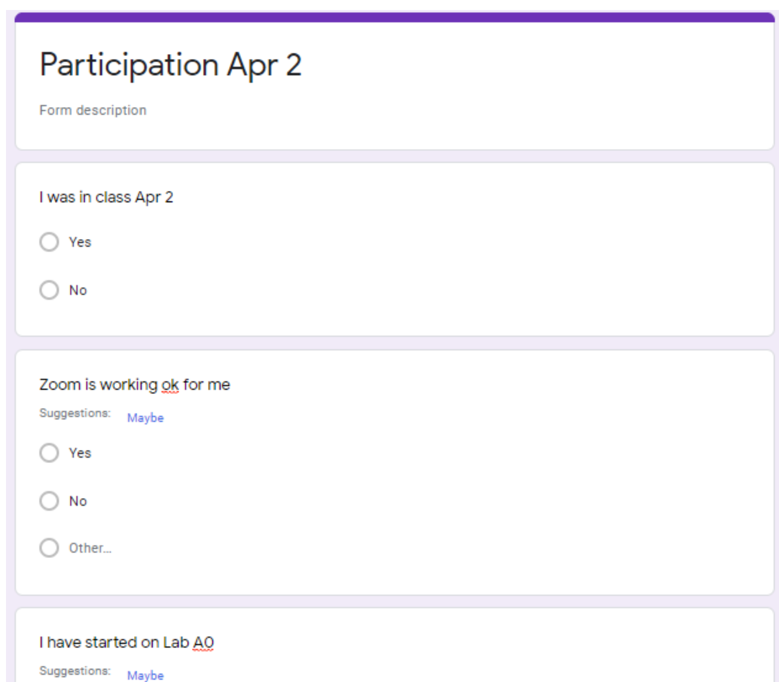
Linear Algebra Review

(On document camera)

- Triangle made of vertices which are Points
- Vectors (P1-P2)
- P1+V=P2
- Vector magnitude (normalizing vectors), what does ||xx|| mean?
- Dot product of two vectors (Wikipedia)
- Cross product of two vectors
- Matrix
- Matrix * Point
- Matrix * Matrix

Break time – 4 minutes

(Survey link in Zoom Chat)



Participation Apr 2

Form description

I was in class Apr 2

- ◯ Yes
- ◯ No

Zoom is working ok for me

Suggestions: Maybe

- ◯ Yes
- ◯ No
- ◯ Other...

I have started on Lab A0

Suggestions: Maybe

About GL and WebGL

OpenGL version history

Version	Release Date	Features
1.1	March 4, 1997	Texture objects
1.2	March 16, 1998	3D textures, BGRA and packed pixel formats,[23] introduction of the *imaging subset* useful to image-processing applications
1.2.1	October 14, 1998	A concept of ARB extensions
1.3	August 14, 2001	Multitexturing, multisampling, texture compression
1.4	July 24, 2002	Depth textures, GLSlang[24]
1.5	July 29, 2003	Vertex Buffer Object (VBO), Occlusion Queries[25]
2.0	September 7, 2004	GLSL 1.1, MRT, Non Power of Two textures, Point Sprites,[26] Two-sided stencil[25]
2.1	July 2, 2006	GLSL 1.2, Pixel Buffer Object (PBO), sRGB Textures[25]
3.0	August 11, 2008	GLSL 1.3, Texture Arrays, Conditional rendering, Frame Buffer Object (FBO)[27]
3.1	March 24, 2009	GLSL 1.4, Instancing, Texture Buffer Object, Uniform Buffer Object, Primitive restart[28]
3.2	August 3, 2009	GLSL 1.5, Geometry Shader, Multi-sampled textures[29]
3.3	March 11, 2010	GLSL 3.30, Backports as much function as possible from the OpenGL 4.0 specification
4.0	March 11, 2010	GLSL 4.00, Tessellation on GPU, shaders with 64-bit precision[30]
4.1	July 26, 2010	GLSL 4.10, Developer-friendly debug outputs, compatibility with OpenGL ES 2.0[31]
4.2	August 8, 2011[32]	GLSL 4.20, Shaders with atomic counters, draw transform feedback instanced, shader packing, performance improvements
4.3	August 6, 2012[33]	GLSL 4.30, Compute shaders leveraging GPU parallelism, shader storage buffer objects, high-quality ETC2/EAC texture compression, increased memory security, a multi-application robustness extension, compatibility with OpenGL ES 3.0[34]
4.4	July 22, 2013[35]	GLSL 4.40, Buffer Placement Control, Efficient Asynchronous Queries, Shader Variable Layout, Efficient Multiple Object Binding, Streamlined Porting of Direct3D applications, Bindless Texture Extension, Sparse Texture Extension[35]
4.5	August 11, 2014[8][36]	GLSL 4.50, Direct State Access (DSA), Flush Control, Robustness, OpenGL ES 3.1 API and shader compatibility, DX11 emulation features
4.6	July 31, 2017[37][38]	GLSL 4.60, More efficient geometry processing and shader execution, more information, no error context, polygon offset clamp, SPIR-V, anisotropic filtering

Example (old style)

type of object

location of vertex

```
glBegin(GL_POLYGON)
 glVertex3f(0.0, 0.0, 0.0);
 glVertex3f(0.0, 1.0, 0.0);
 glVertex3f(0.0, 0.0, 1.0);
glEnd( );
```

end of object definition

Example (GPU based)

- Put geometric data in an array

```
var points = [
 vec3(0.0, 0.0, 0.0),
 vec3(0.0, 1.0, 0.0),
 vec3(0.0, 0.0, 1.0),
];
```

- Send array to GPU
- Tell GPU to render as triangle

OpenGL 3.1

- Totally shader-based
  - No default shaders
  - Each application must provide both a vertex and a fragment shader
- No immediate mode
- Few state variables
- Most 2.5 functions deprecated
- Backward compatibility not required
  - Exists a compatibility extension

Other Versions

- OpenGL ES
  - Embedded systems
  - Version 1.0 simplified OpenGL 2.1
  - Version 2.0 simplified OpenGL 3.1
    - Shader based
- WebGL
  - Javascript implementation of ES 2.0
  - Supported on newer browsers
- OpenGL 4.1, 4.2, …..
  - Add geometry, tessellation, compute shaders

Lack of Object Orientation

The University of New Mexico

- All versions of OpenGL are not object oriented so that there are multiple functions for a given logical function
- Example: sending values to shaders
  - **`gl.uniform3f`**
  - **`gl.uniform2i`**
  - **`gl.uniform3dv`**
- Underlying storage mode is the same

WebGL function format

The University of New Mexico

function name

dimension

`gl.uniform3f(x,y,z)`

belongs to WebGL canvas

`x,y,z` are variables

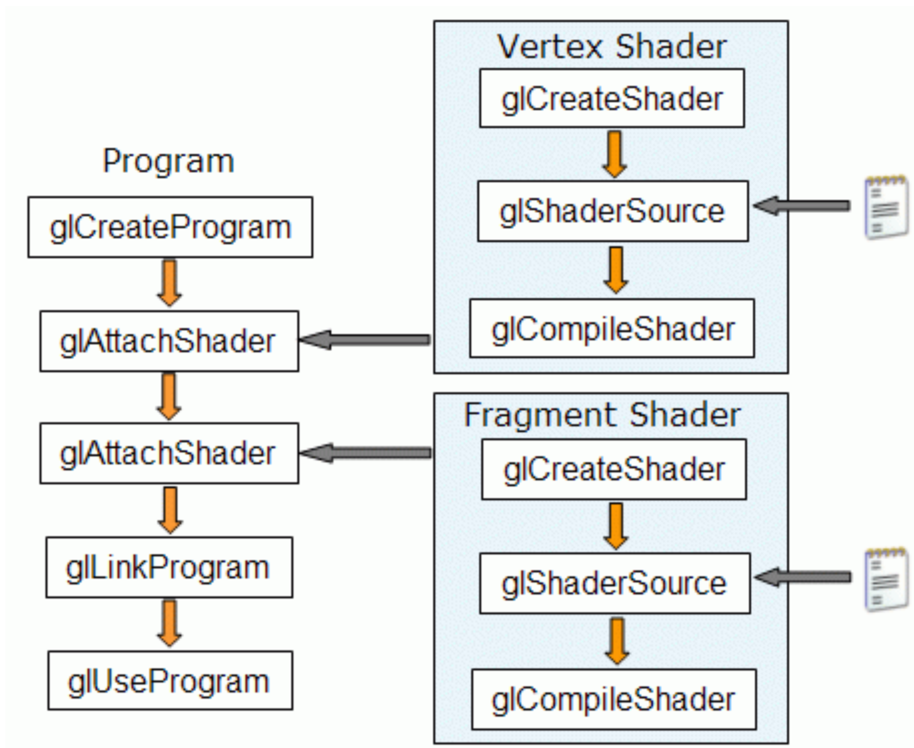
`gl.uniform3fv(p)`

`p` is an array

WebGL constants

- Most constants are defined in the canvas object
  - In desktop OpenGL, they were in #include files such as **`gl.h`**
- Examples
  - **`desktop OpenGL`**
    - **`glEnable(GL_DEPTH_TEST);`**
  - **`WebGL`**
    - **`gl.enable(gl.DEPTH_TEST)`**
  - **`gl.clear(gl.COLOR_BUFFER_BIT)`**

GLSL Shaders Overview


Program
glCreateProgram
glAttachShader
glAttachShader
glLinkProgram
glUseProgram
Vertex Shader
glCreateShader
glShaderSource
glCompileShader
Fragment Shader
glCreateShader
glShaderSource
glCompileShader

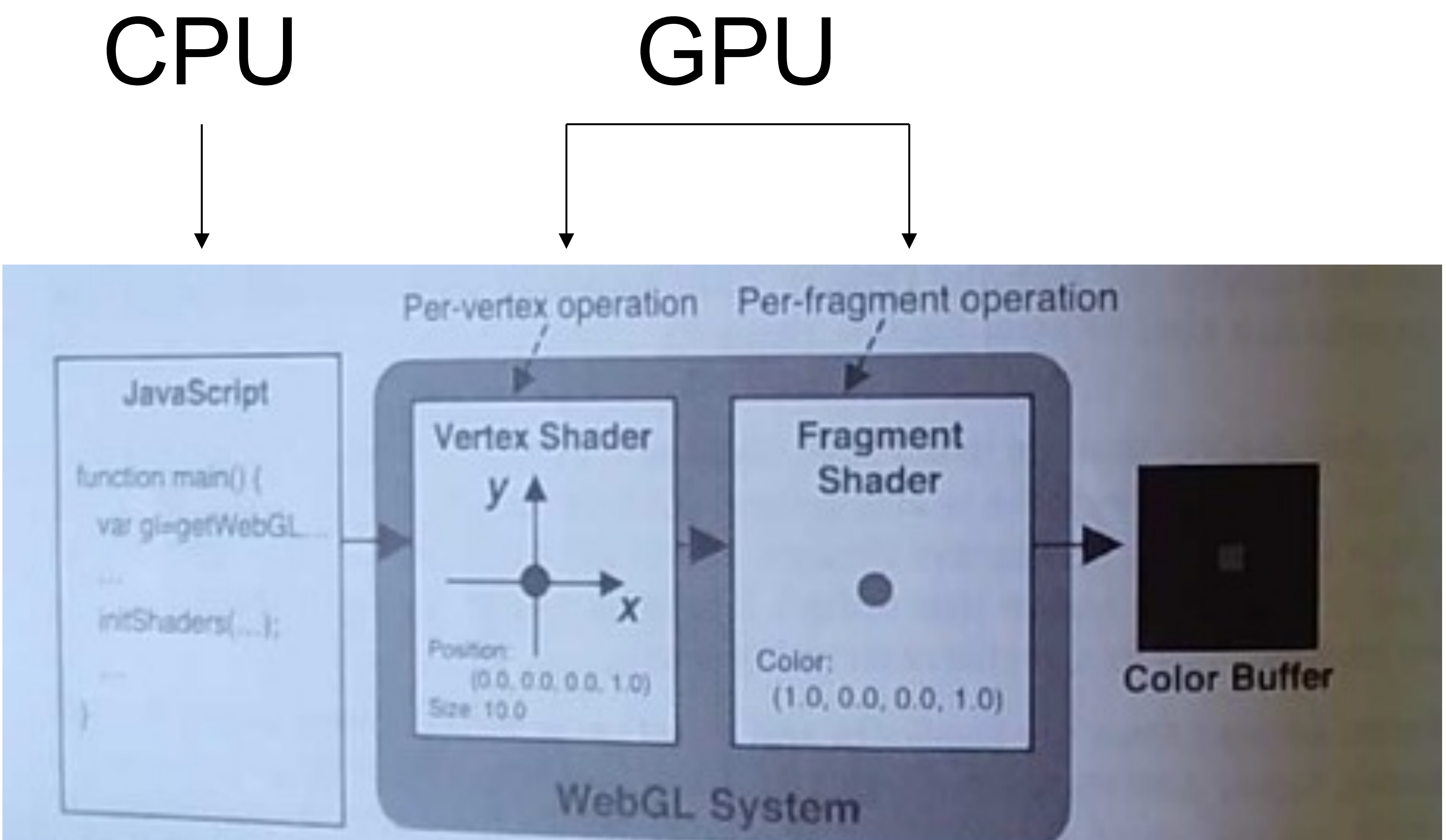


Figure 2.11 The simplified version of Figure 2.9

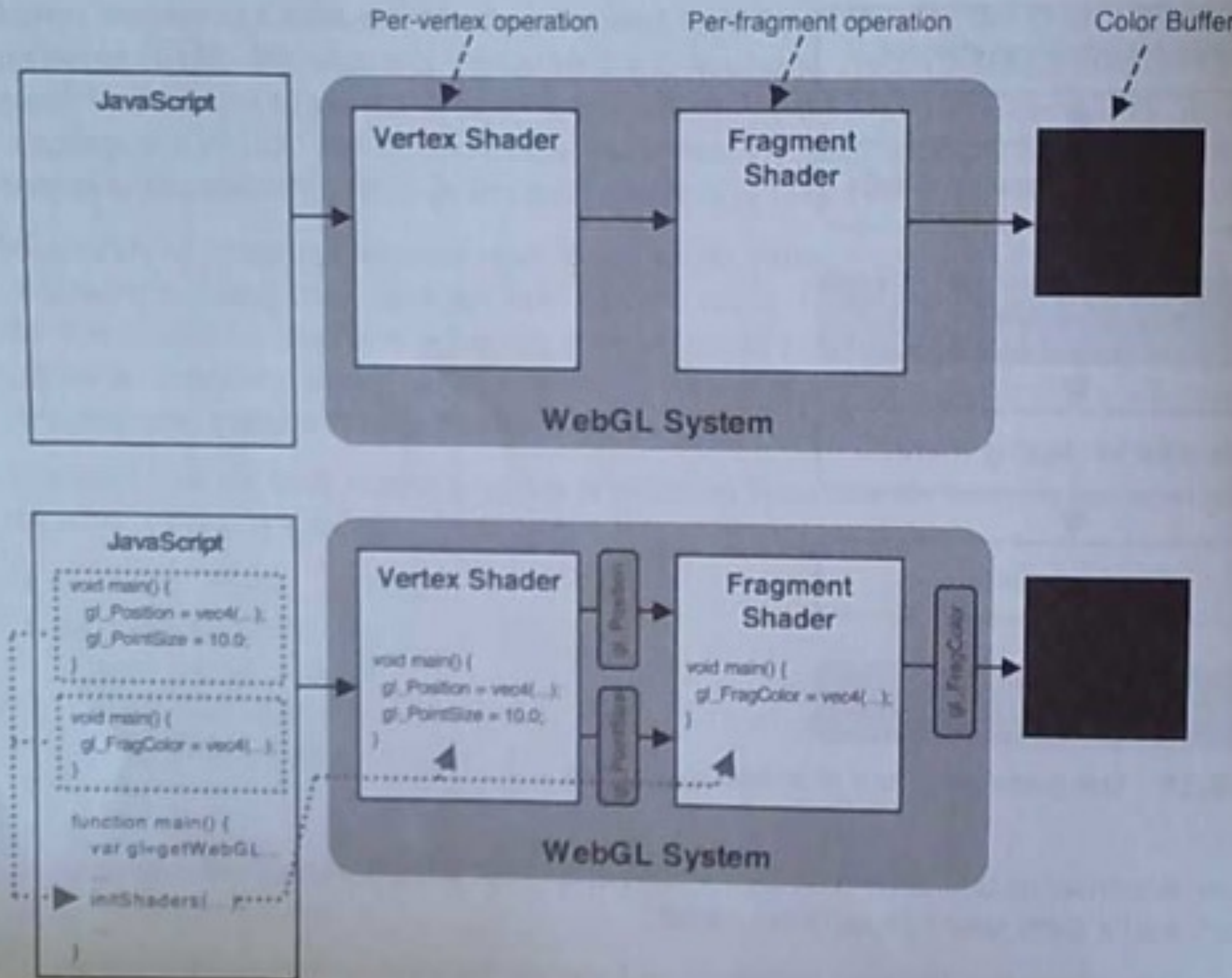


Figure 2.14 Behavior of initShaders()

Example (old style)

type of object

location of vertex

```
glBegin(GL_POLYGON)
 glVertex3f(0.0, 0.0, 0.0);
 glVertex3f(0.0, 1.0, 0.0);
 glVertex3f(0.0, 0.0, 1.0);
glEnd( );
```

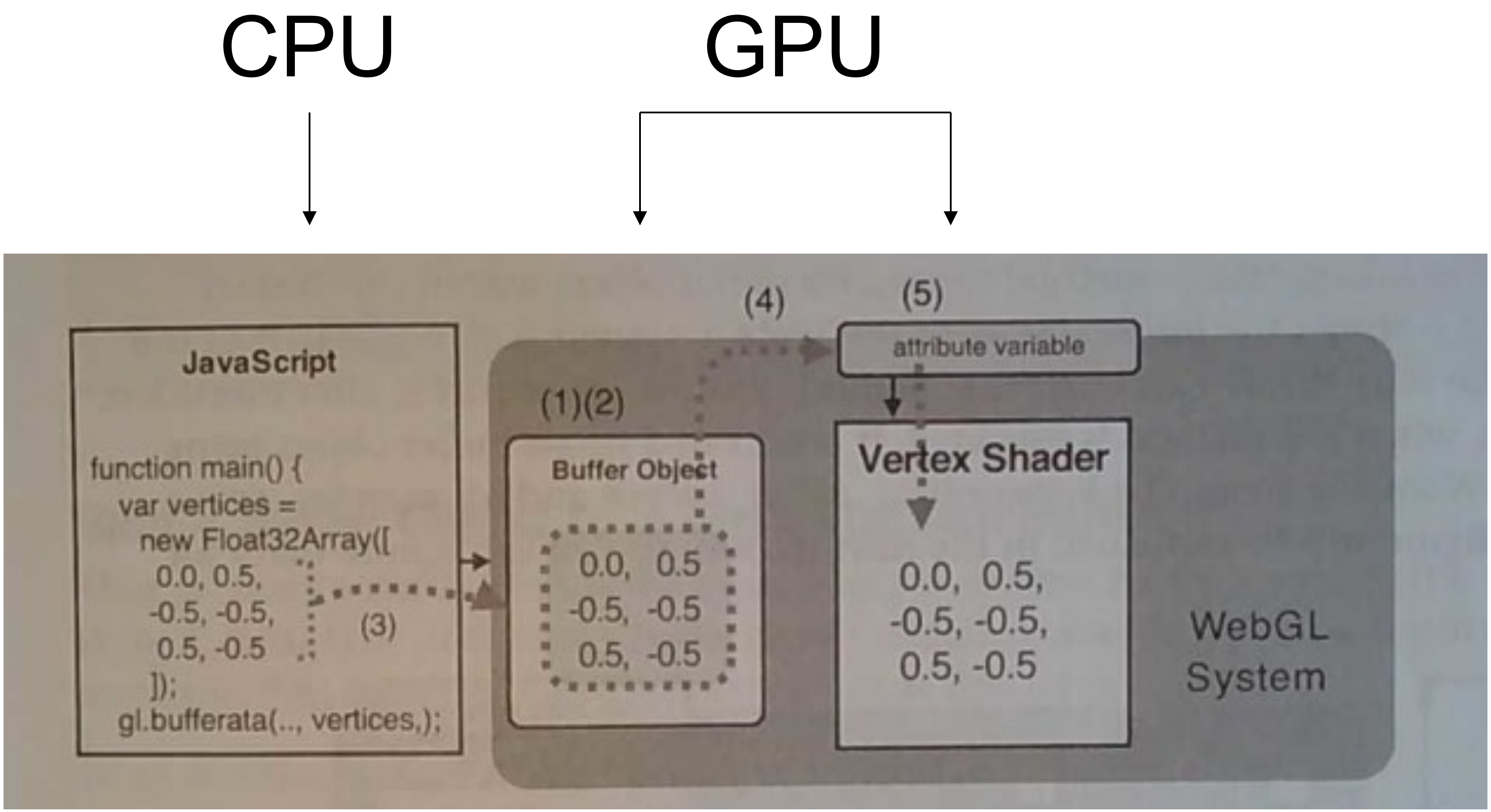
end of object definition

Example (GPU based)

- Put geometric data in an array

```
var points = [
 vec3(0.0, 0.0, 0.0),
 vec3(0.0, 1.0, 0.0),
 vec3(0.0, 0.0, 1.0),
];
```

- Send array to GPU
- Tell GPU to render as triangle


CPU
GPU
JavaScript
function main() {
var vertices =
new Float32Array([
0.0, 0.5,
-0.5, -0.5,
0.5, -0.5
]);
gl.bufferata(.., vertices,);
(3)
(1)(2)
Buffer Object
0.0, 0.5
-0.5, -0.5
0.5, -0.5
(4)
(5)
attribute variable
Vertex Shader
0.0, 0.5,
-0.5, -0.5,
0.5, -0.5
WebGL
System

```
51  var vertices = new Float32Array([
52    0.0, 0.5,   -0.5, -0.5,   0.5, -0.5
53  ]);
54  var n = 3; // The number of vertices
55
56  // Create a buffer object
57  var vertexBuffer = gl.createBuffer();
58  if (!vertexBuffer) {
59    console.log('Failed to create the buffer object ');
60    return -1;
61  }
62
63  // Bind the buffer object to target
64  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
65  // Write date into the buffer object
66  gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);
67
68  var a_Position = gl.getAttribLocation(gl.program, 'a_Position');
...
73  // Assign the buffer object to a_Position variable
74  gl.vertexAttribPointer(a_Position, 2, gl.FLOAT, false, 0, 0);
75
76  // Enable the assignment to a_Position variable
77  gl.enableVertexAttribArray(a_Position);
78
79  return n;
80 }
```

The new function `initVertexBuffers()` is defined at line 50 and used at line 34 to set up the vertex buffer object. The function stores multiple vertices in the buffer object and then completes the preparations for passing it to a vertex shader:

```
33 // Set the positions of vertices
34 var n = initVertexBuffers(gl);
```

The return value of this function is the number of vertices being drawn, stored in the variable `n`. Note that in case of error, `n` is negative.

As in the previous examples, the drawing operation is carried out using a single call to `gl.drawArrays()` at Line 48. This is similar to `ClickedPoints.js` except that `n` is passed as the third argument of `gl.drawArrays()` rather than the value 1:

```
46  // Draw three points
47  gl.drawArrays(gl.POINTS, 0, n); // n is 3
```

Because you are using a buffer object to pass multiple vertices to a vertex shader in `initVertexBuffers()`, you need to specify the number of vertices in the object as the third parameter of `gl.drawArrays()` so that WebGL then knows to draw a shape using all the vertices in the buffer object.

gl.createBuffer()

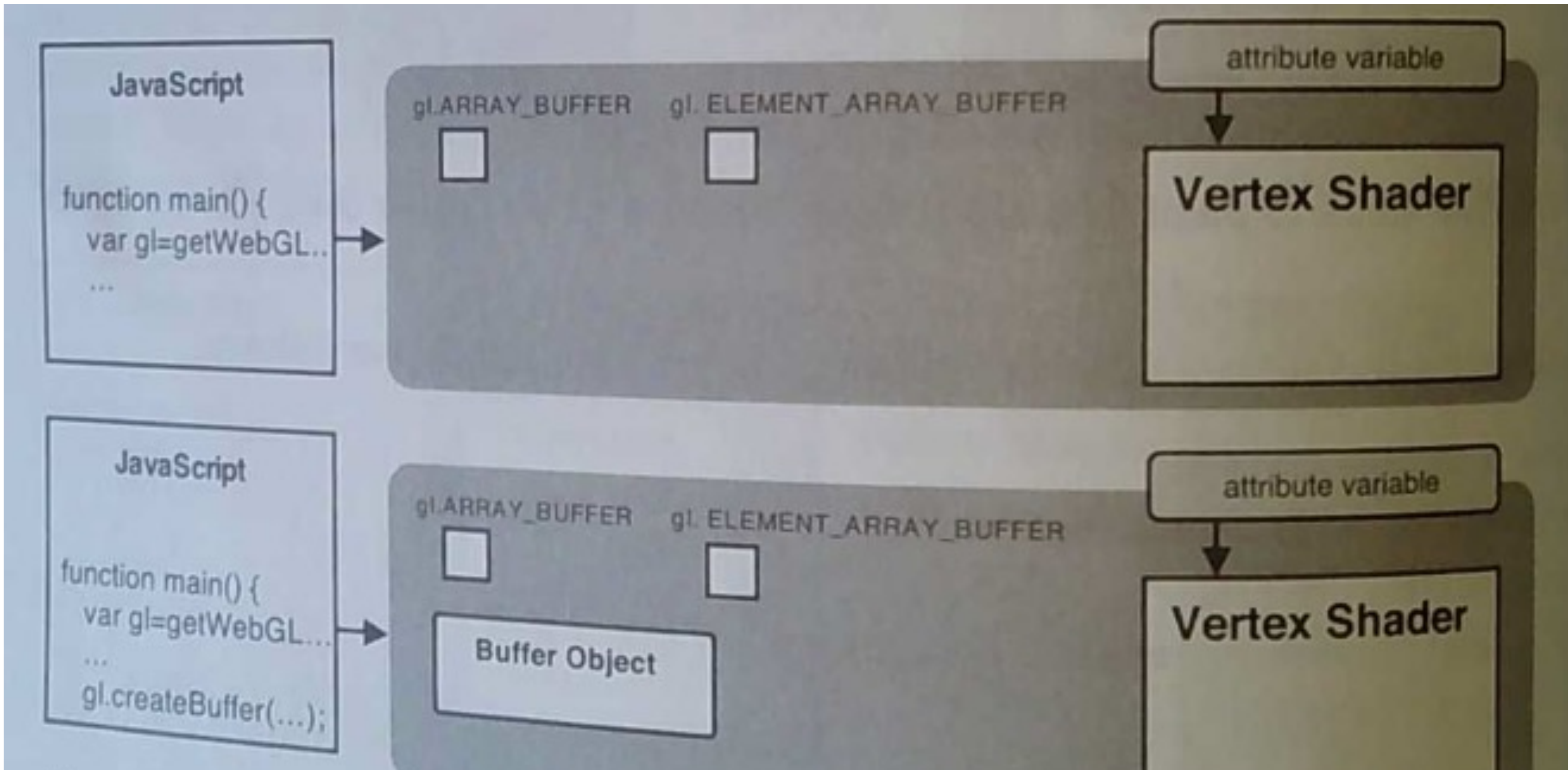


Figure 3.6 Create a buffer object

gl.bindBuffer()

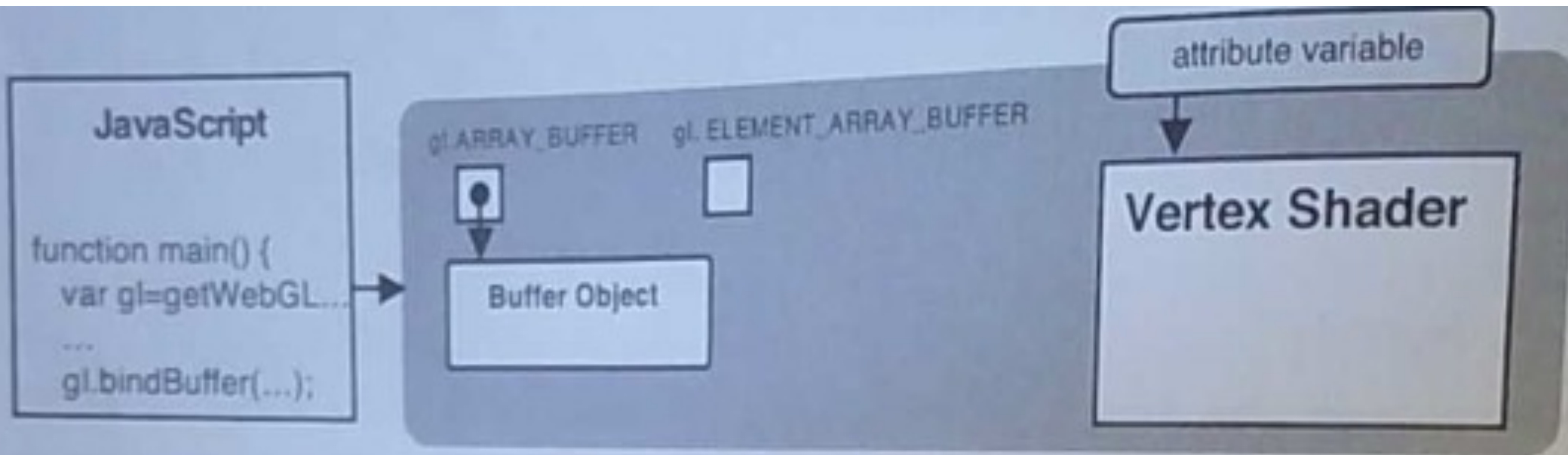


Figure 3.7 Bind a buffer object to a target

gl.bufferData()

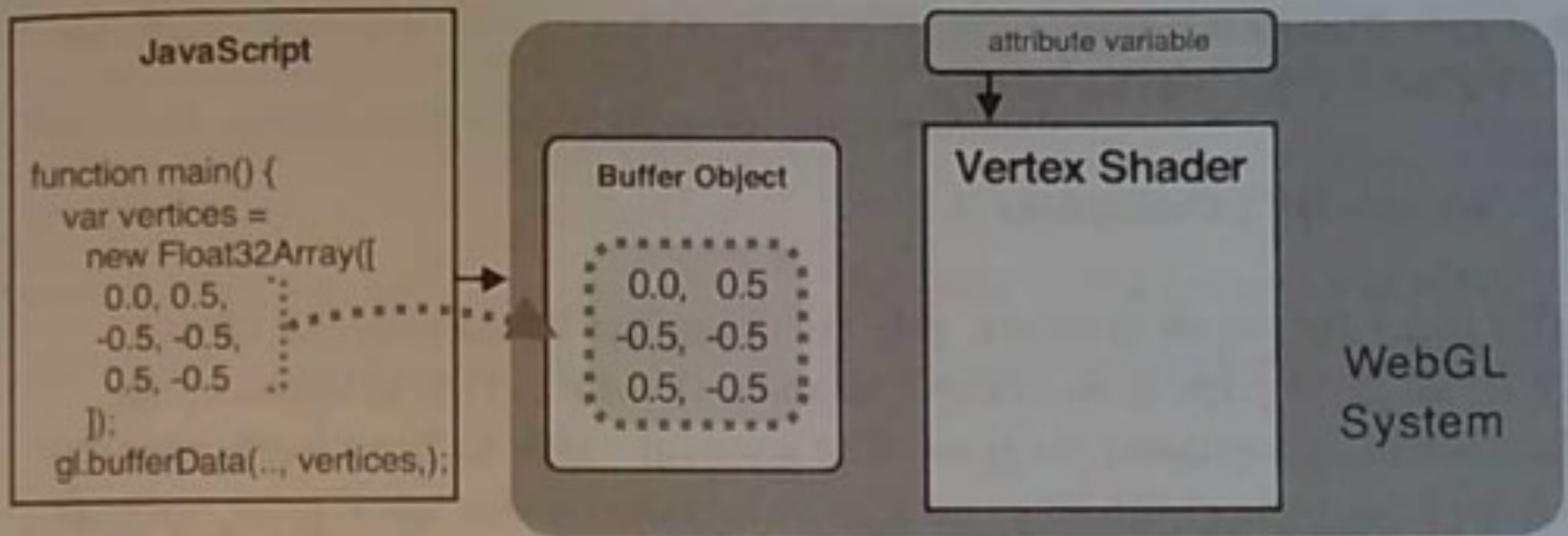


Figure 3.8 Allocate storage and write data into a buffer object

gl.vertexAttribPointer()

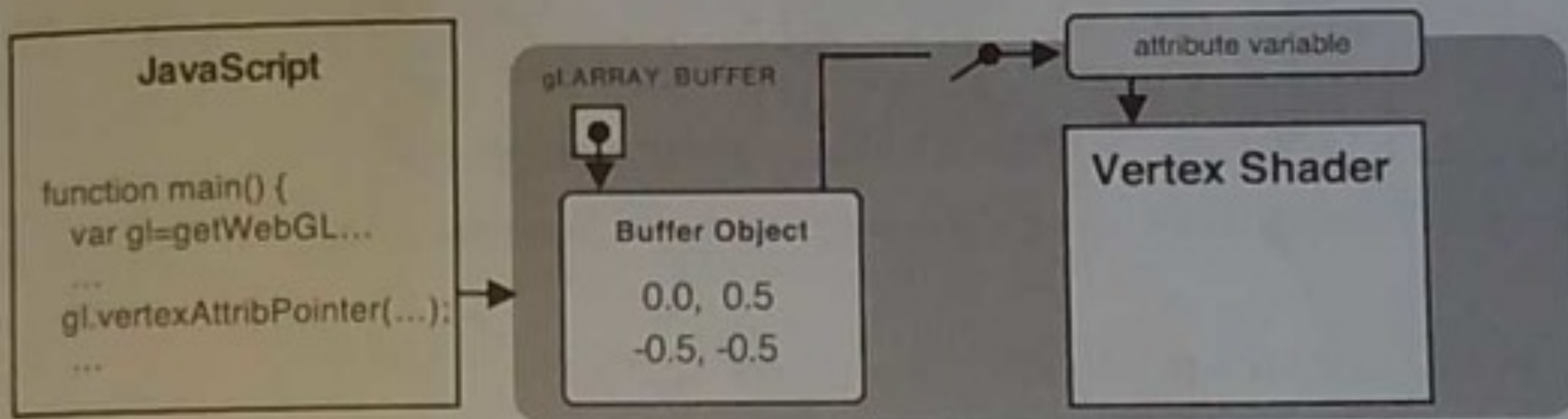


Figure 3.9 Assign a buffer object to an attribute variable

gl.vertexAttribArray()

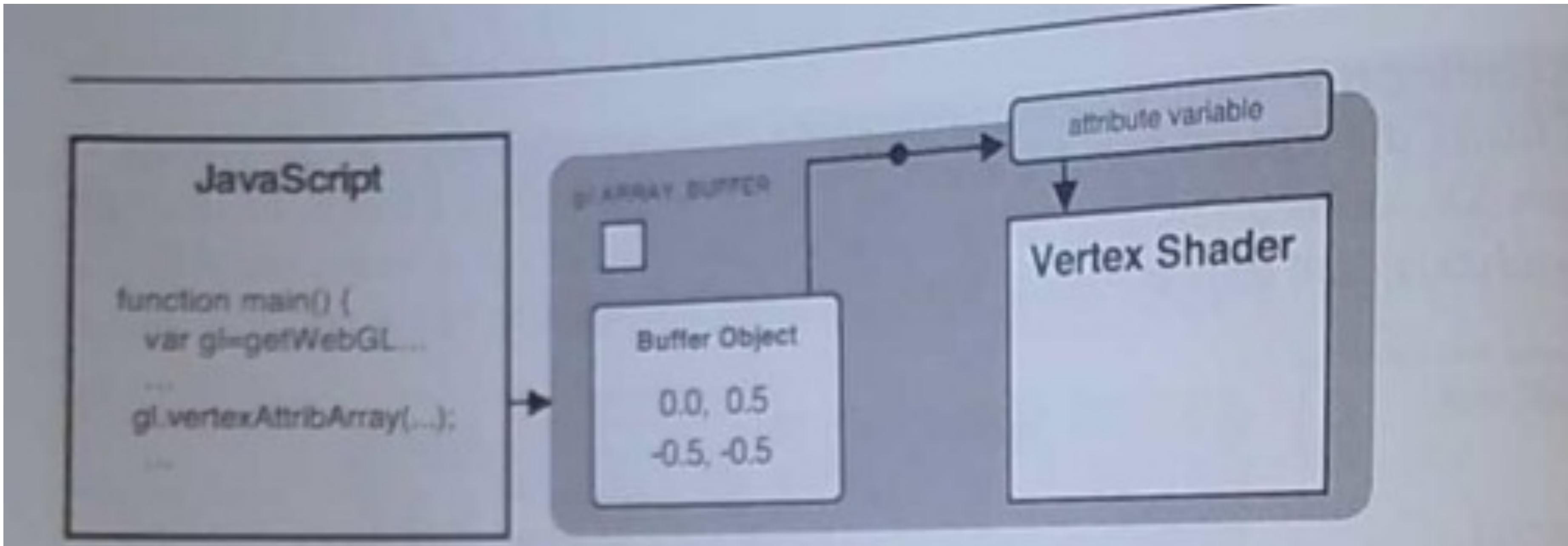


Figure 3.10 Enable the assignment of a buffer object to an attribute variable

Vertex shader runs for *each* item in the buffer

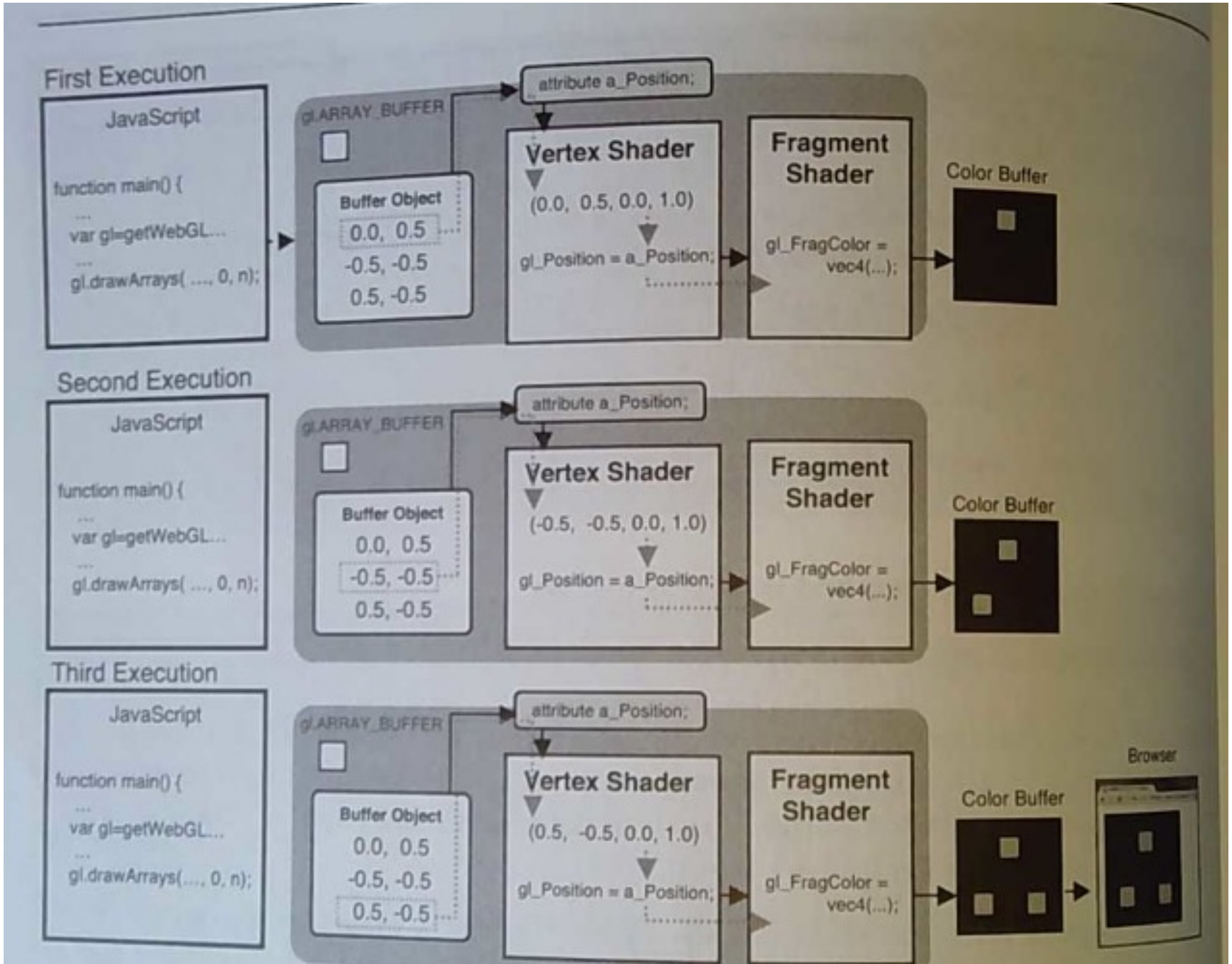
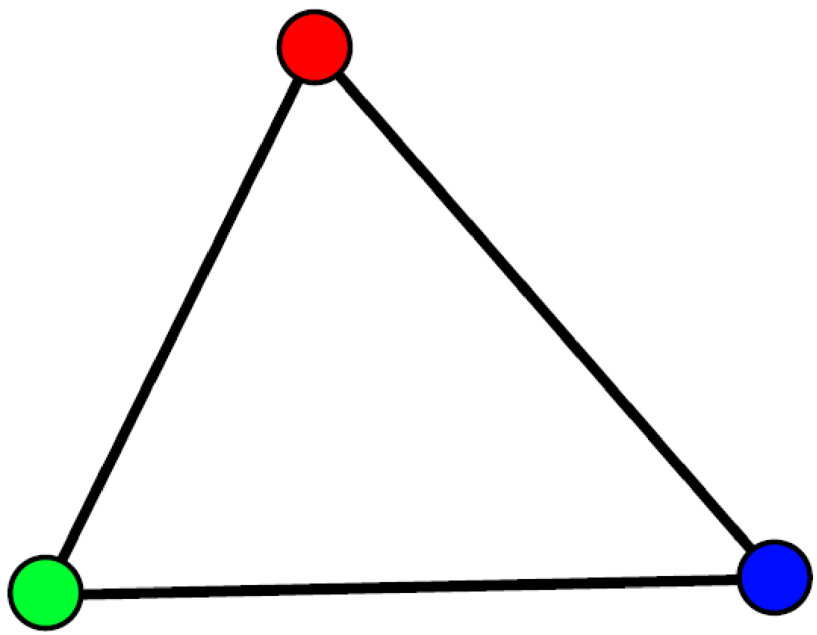


Figure 3.11 How the data in a buffer object is passed to a vertex shader during execution

How is shading done in OpenGL?

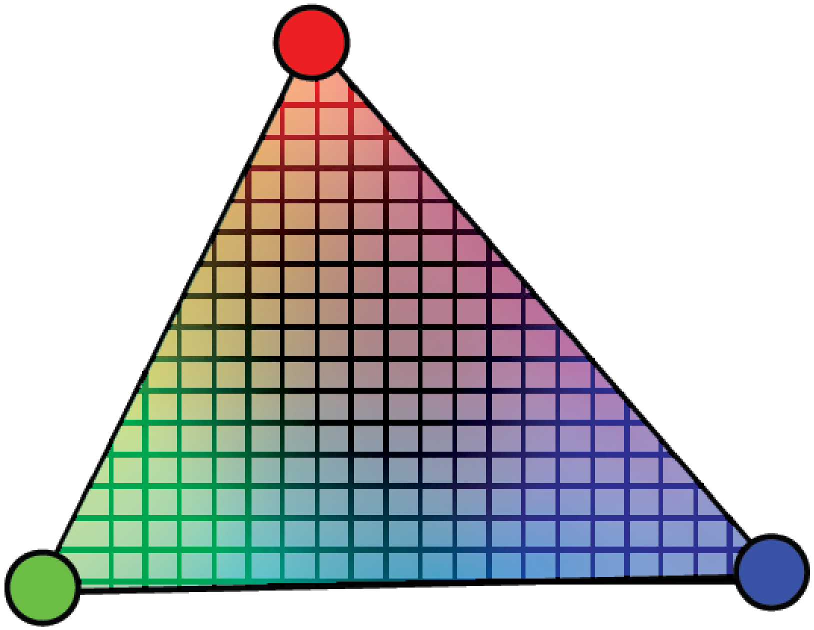
1. Attributes are specified on vertices.



How is shading done in OpenGL?

2. Attributes are interpolated across triangles by the rasterizer

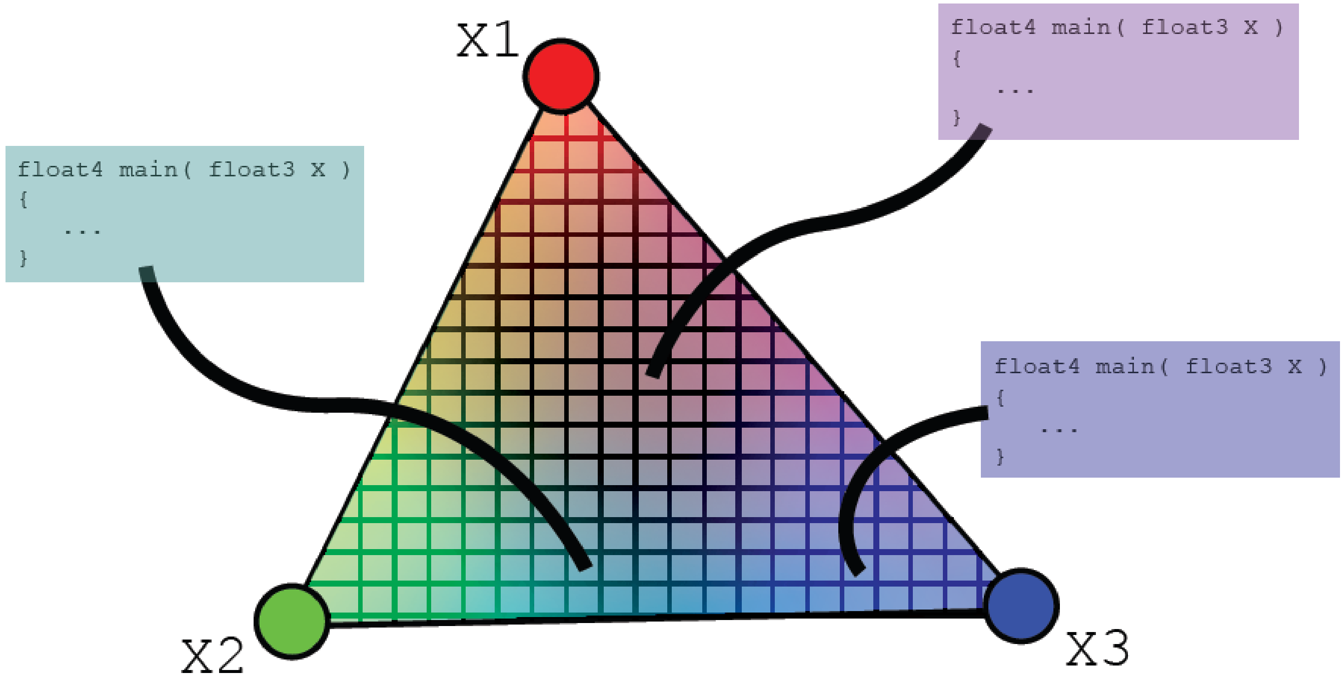
(see appendix for details)



Rasterizer also breaks the triangle into “fragments.”

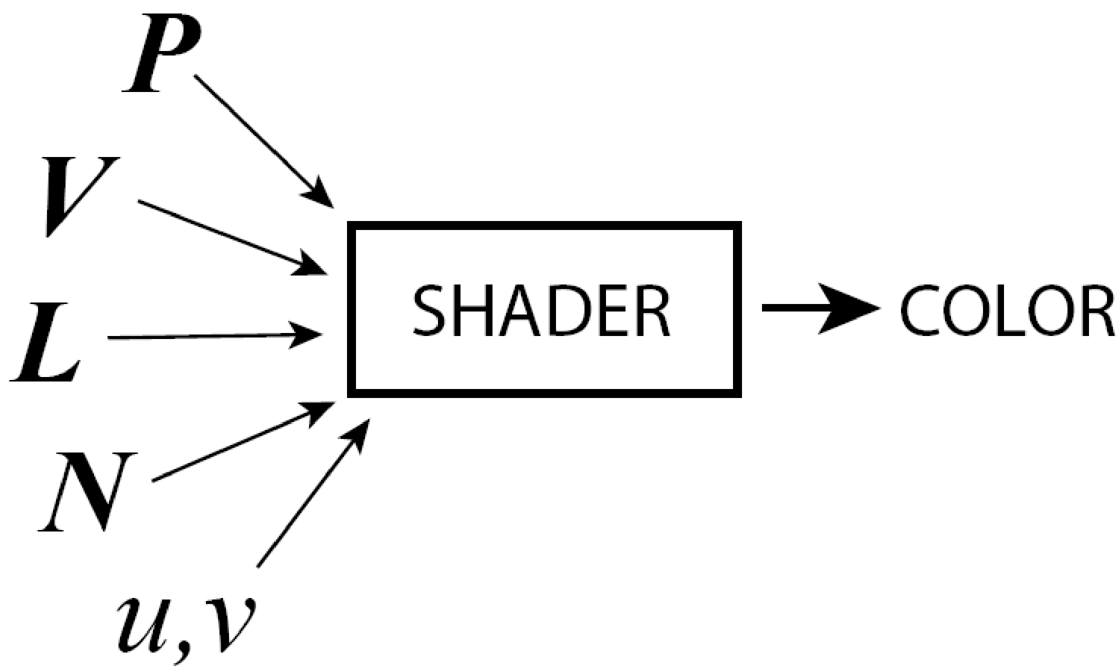
How is shading done in OpenGL?

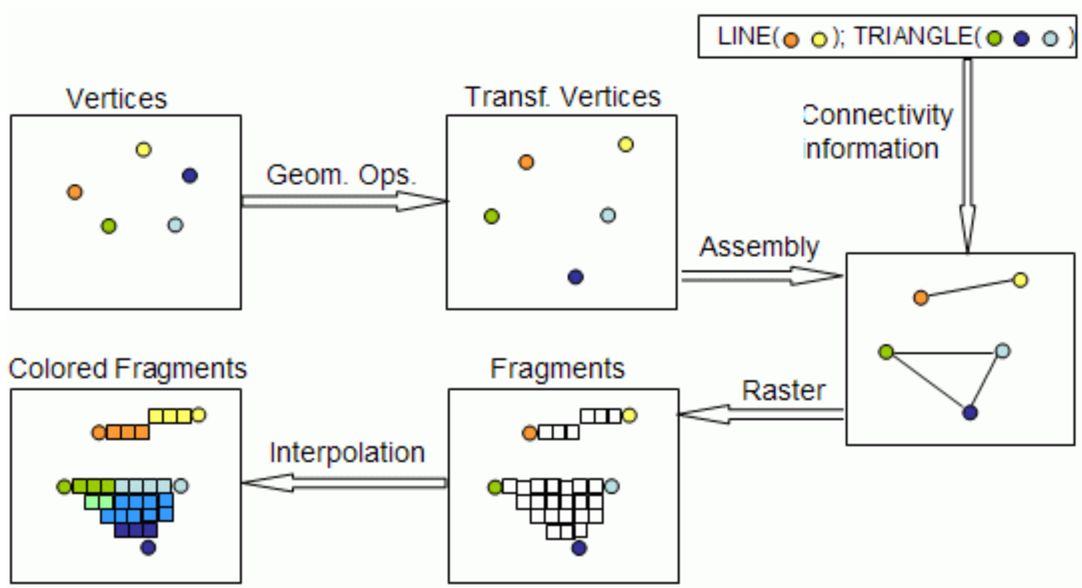
3. Each fragment runs the shader using interpolated values as inputs.



Exact same routine, different inputs.

What is a shader?





LINE( ); TRIANGLE( )
Vertices
Geom. Ops.
Transf. Vertices
Connectivity
information
Assembly
Raster
Colored Fragments
Fragments
Interpolation

What is a shader?

INPUTS → SHADER → POSITION

“displacement” or “vertex” shader

original | normal mapped | displaced

Super simple vertex shader

```
void main()
{

        gl_Position = gl_ProjectionMatrix * gl_ModelViewMatrix * gl_Vertex;
}
```

```
void main(void)
{
        vec4 v = vec4(gl_Vertex);
        v.z = sin(5.0*v.x )*0.25;

        gl_Position = gl_ModelViewProjectionMatrix * v;
}
```

Vertex Shader

```
uniform vec3 lightDir;

varying float intensity;

void main()
{
        vec3 ld;

        intensity = dot(lightDir,gl_Normal);

        gl_Position = ftransform();
}
```

Fragment Shader

```
varying float intensity;

void main()
{
        vec4 color;

        if (intensity > 0.95)
                color = vec4(1.0,0.5,0.5,1.0);
        else if (intensity > 0.5)
                color = vec4(0.6,0.3,0.3,1.0);
        else if (intensity > 0.25)
                color = vec4(0.4,0.2,0.2,1.0);
        else
                color = vec4(0.2,0.1,0.1,1.0);

        gl_FragColor = color;
}
```

Simple diffuse lighting

The following vertex shader shows the GLSL code to achieve this.

```
void main() {

        vec3 normal, lightDir;
        vec4 diffuse;
        float NdotL;

        /* first transform the normal into eye space and normalize the result */
        normal = normalize(gl_NormalMatrix * gl_Normal);

        /* now normalize the light's direction. Note that according to the
        OpenGL specification, the light is stored in eye space. Also since
        we're talking about a directional light, the position field is actually
        direction */
        lightDir = normalize(vec3(gl_LightSource[0].position));

        /* compute the cos of the angle between the normal and lights direction.
        The light is directional so the direction is constant for every vertex.
        Since these two are normalized the cosine is the dot product. We also
        need to clamp the result to the [0,1] range. */
        NdotL = max(dot(normal, lightDir), 0.0);

        /* Compute the diffuse term */
        diffuse = gl_FrontMaterial.diffuse * gl_LightSource[0].diffuse;

        gl_FrontColor =  NdotL * diffuse;

        gl_Position = ftransform();
}
```

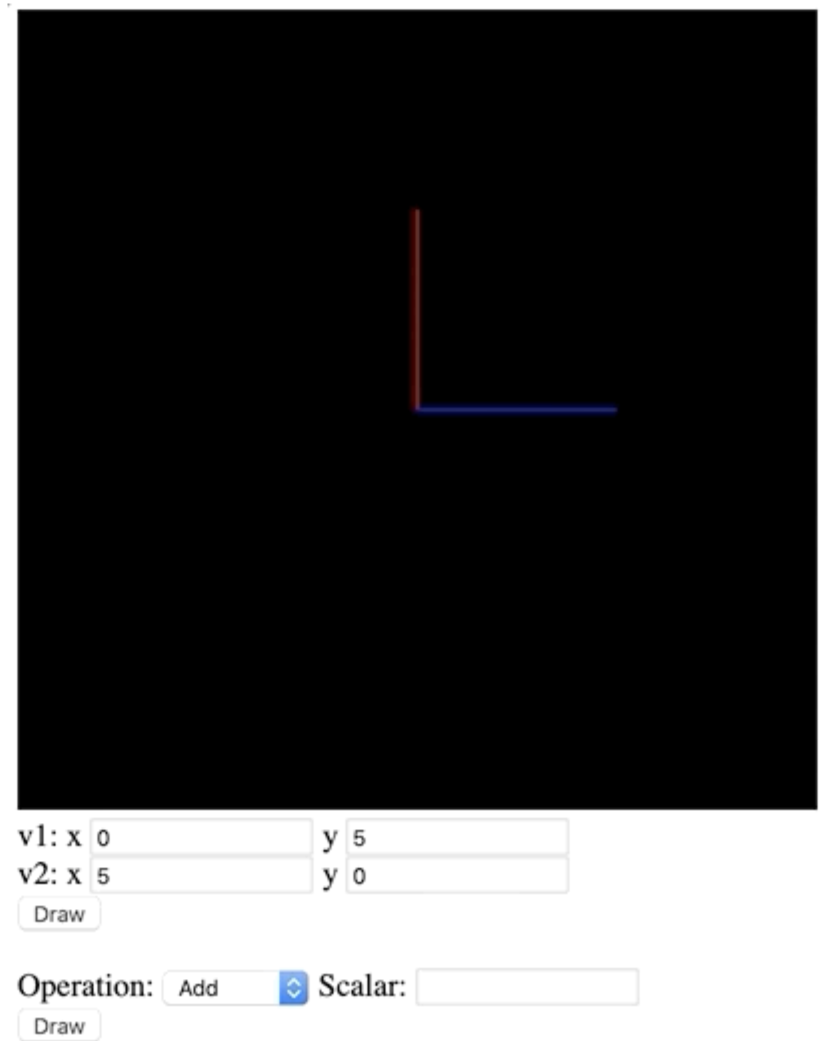
Now in the fragment shader all there is left to do is setting the fragments color, using the varying gl_Color variable.

```
void main()
{
        gl_FragColor = gl_Color;
}
```

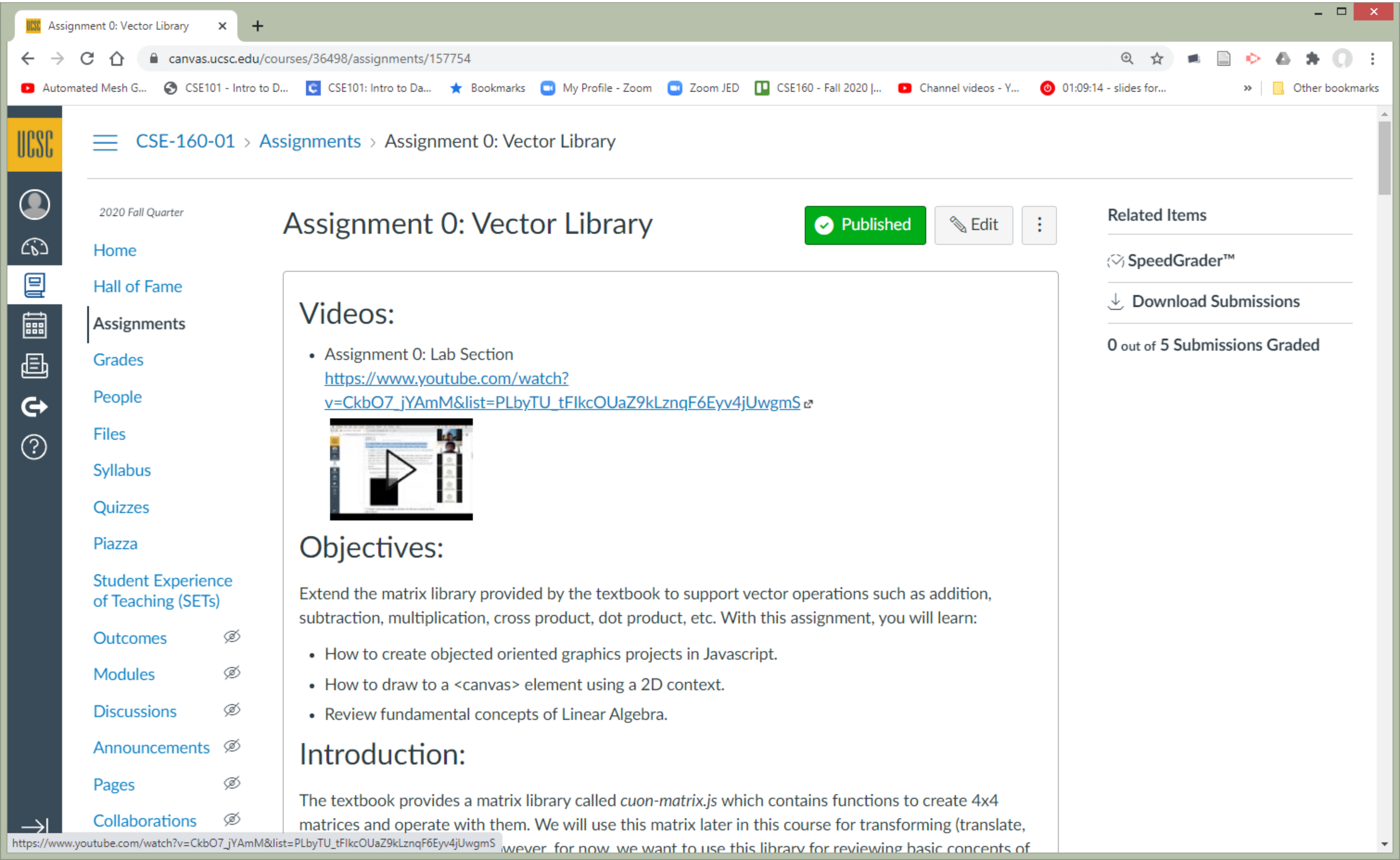
Assignment 0 and 1 Overview

(There are videos in Youtube Channel)

Assignment 0 – Intro HTML, Javascript, Linear Algebra



A0 – Detailed directions



A0 – Detailed grading rubric

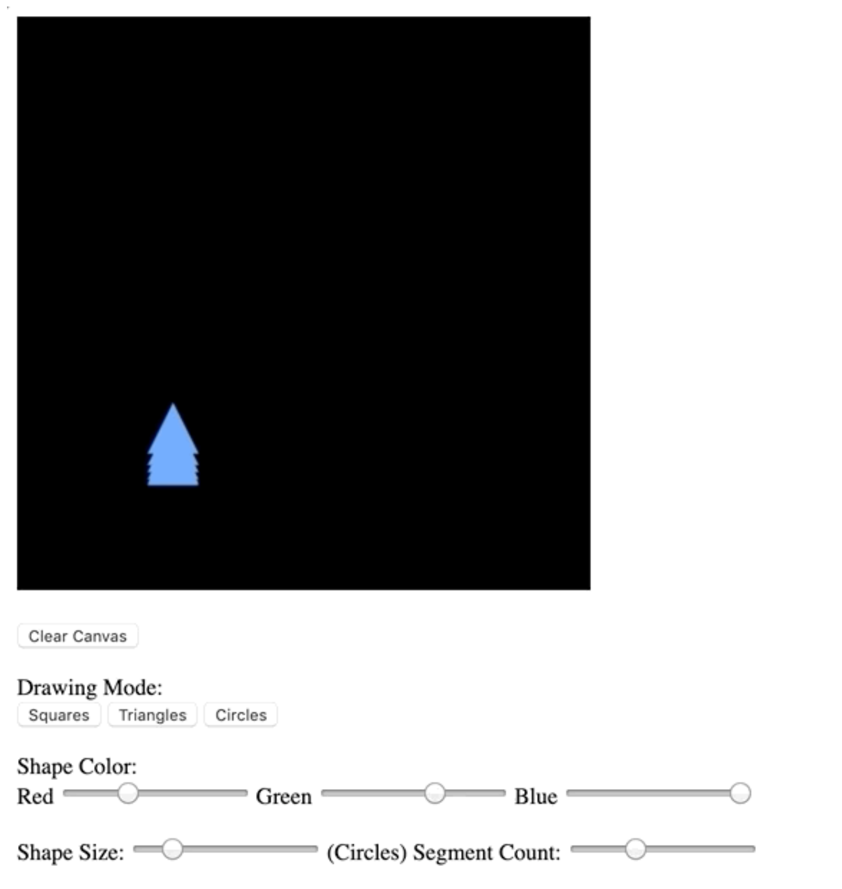
Assignment 0: Vector Library

canvas.ucsc.edu/courses/36498/assignments/157754

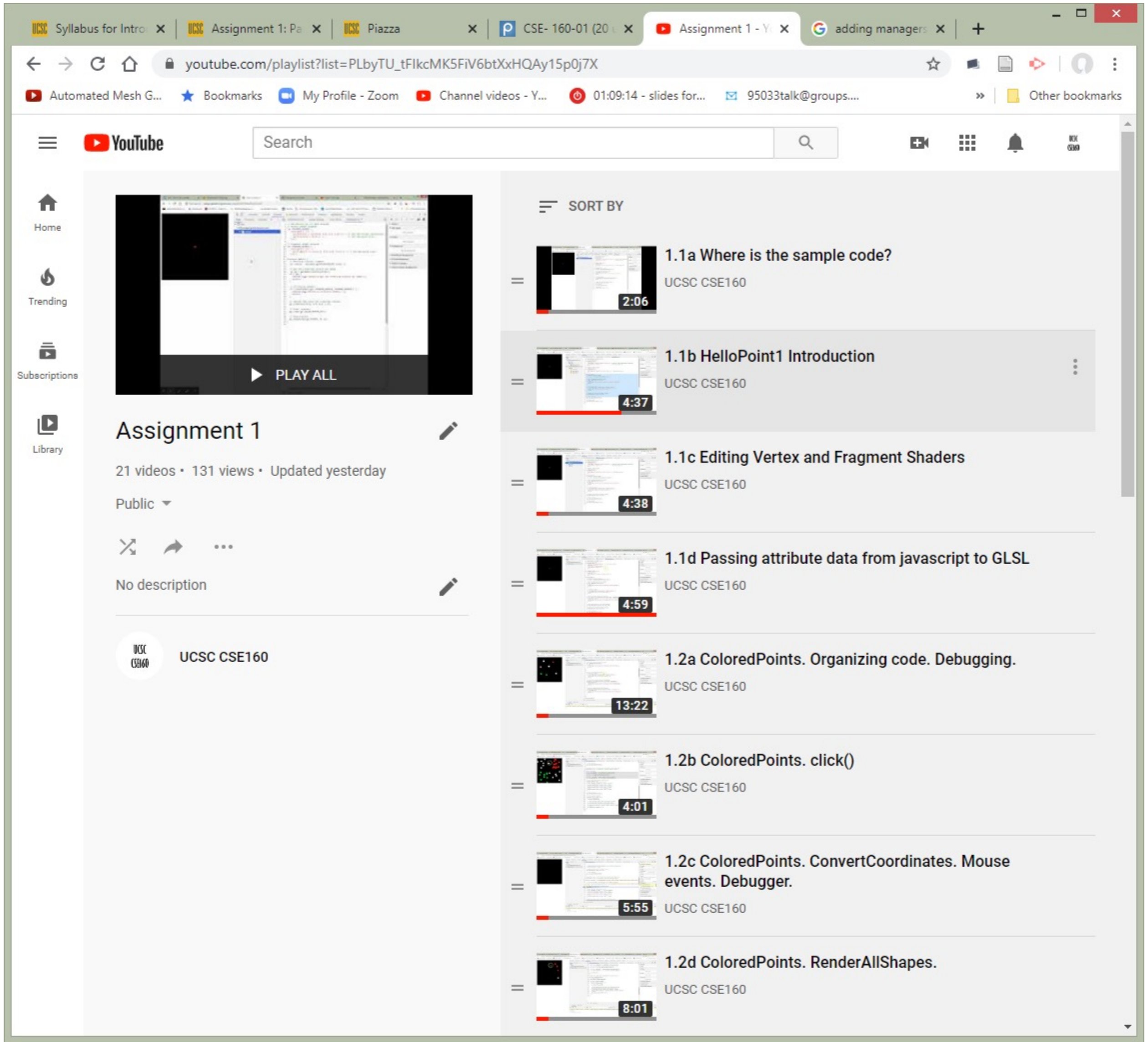
ASG0 Rubric

Criteria	Ratings		Pts
Setup the DrawRectangle example from the book with our matrix library.	1.0 pts Full Marks	0.0 pts No Marks	1.0 pts
Draw a red vector v1 on a black canvas instead of the blue rectangle. The origin of the vector should be the center of the canvas.	1.0 pts Full Marks	0.0 pts No Marks	1.0 pts
Add to your webpage an interface for the user to specify and draw the v1 vector.	2.0 pts Full Marks	0.0 pts No Marks	2.0 pts
Add to your webpage an interface for the user to specify and draw a second vector v2.	1.0 pts Full Marks	0.0 pts No Marks	1.0 pts
Add to your webpage an interface for the user to perform and visualize the results of add, sub, div and mul operations.	2.0 pts Full Marks	0.0 pts No Marks	2.0 pts
Add to your webpage an interface for the user to perform and visualize the results of magnitude and normalize operations.	1.0 pts Full Marks	0.0 pts No Marks	1.0 pts

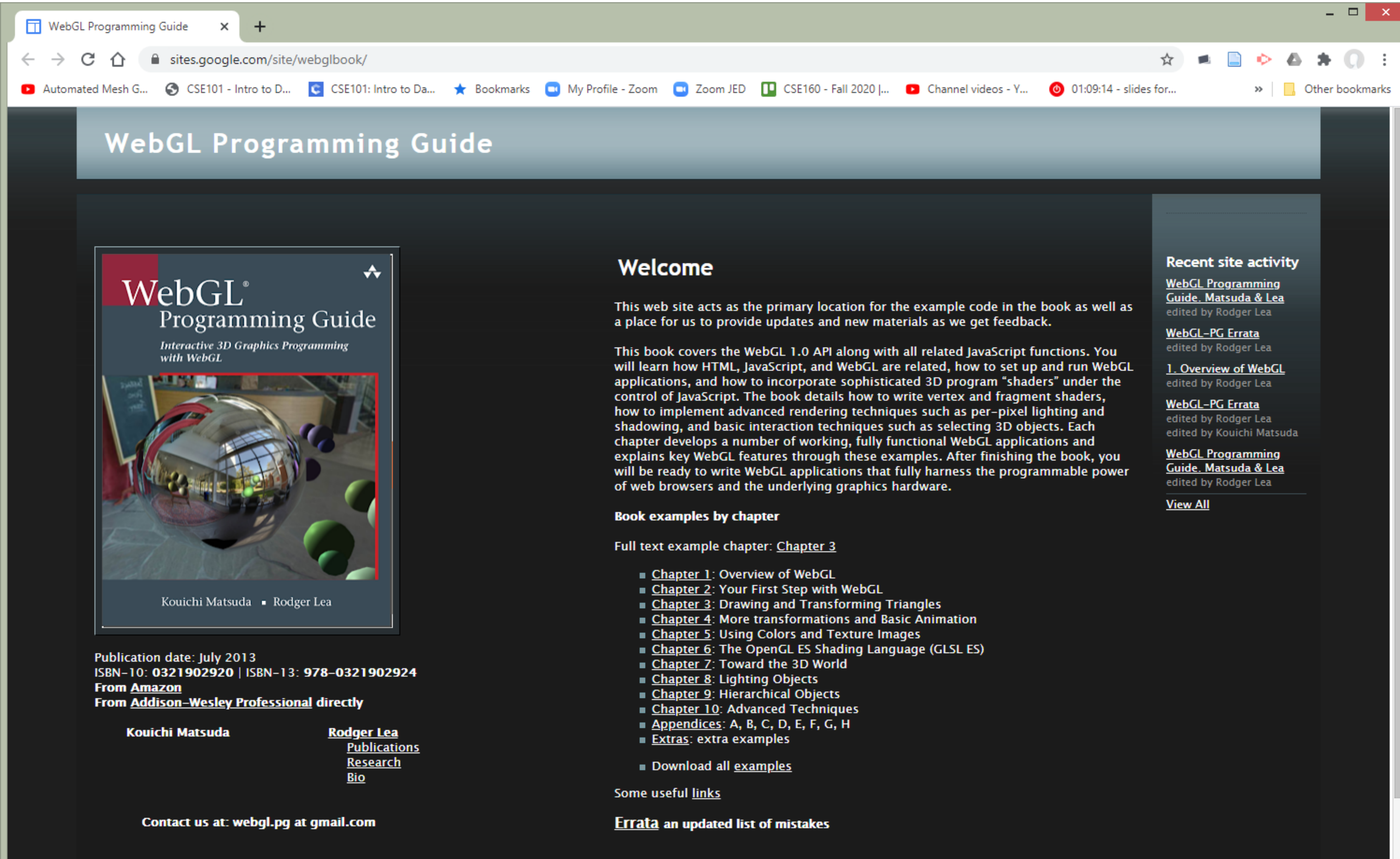
Assignment 1 – WebGL, Paint



Detailed help videos (starting with A1)



Detailed WebGL examples



Administrative

Q&A

End