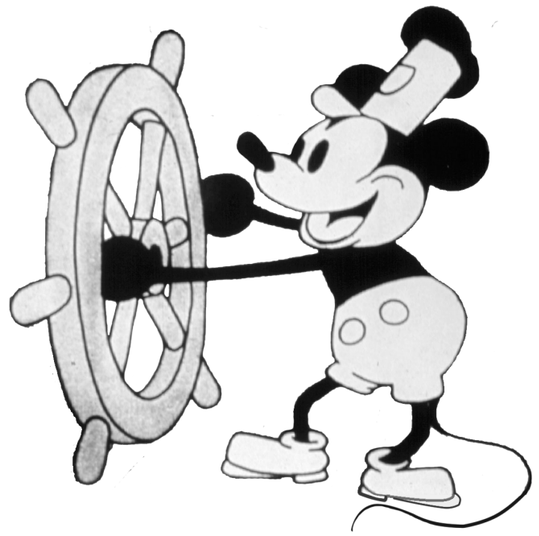
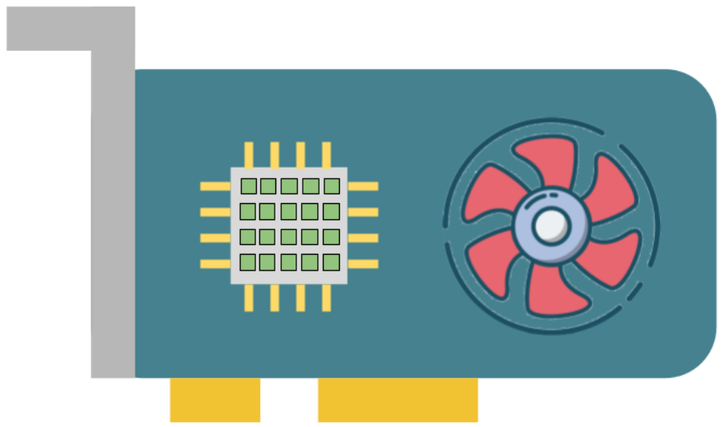


CSCI 461: Computer Graphics

Middlebury College, Spring 2025

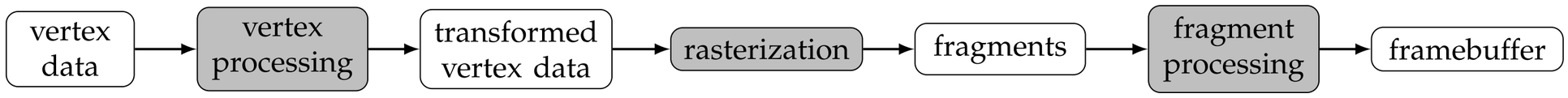
Lecture 10: Rasterization with `WebGL`

By the end of today's lecture, you will be able to:



- **compile** shader **programs**,
- **write** data to the GPU using **buffers**,
- **bind** data in a buffer to an **attribute** in a shader program,
- declare and assign **varyings** passed from the vertex processing stage to the fragment processing stage,
- set and use global variables (**uniforms**) in your shader program,
- write **texture** data to the GPU and fetch texels from a shader,
- write a complete `WebGL` application from start to finish.

Last week we introduced rasterization - recall our rendering pipeline.




```
attribute vec3 a_Position;
attribute vec3 a_Normal;

uniform mat4 u_ModelViewMatrix;
uniform mat4 u_NormalMatrix;
uniform mat4 u_ModelViewProjectionMatrix;

varying vec3 v_Normal;
varying vec3 v_Position;
void main() {
  gl_Position   = u_ModelViewProjectionMatrix * vec4(a_Position, 1.0);
  v_Normal = mat3(u_NormalMatrix) * a_Normal;
  v_Position = (u_ModelViewMatrix * vec4(a_Position, 1.0)).xyz;
};
```

```
precision highp float;

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

Warmup question (event # 9000135): in last week's lab we used

`gl.clear(gl.DEPTH_BUFFER_BIT | gl.COLOR_BUFFER_BIT)`

See `WebGL` constants: https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/Constants

Please evaluate: gl.DEPTH_BUFFER_BIT | gl.COLOR_BUFFER_BIT: 8

- ○ 0x00000500
- ○ 0x00004000
- ○ 0x00004100
- ○ 0x00004110
- ○ 0x00004111

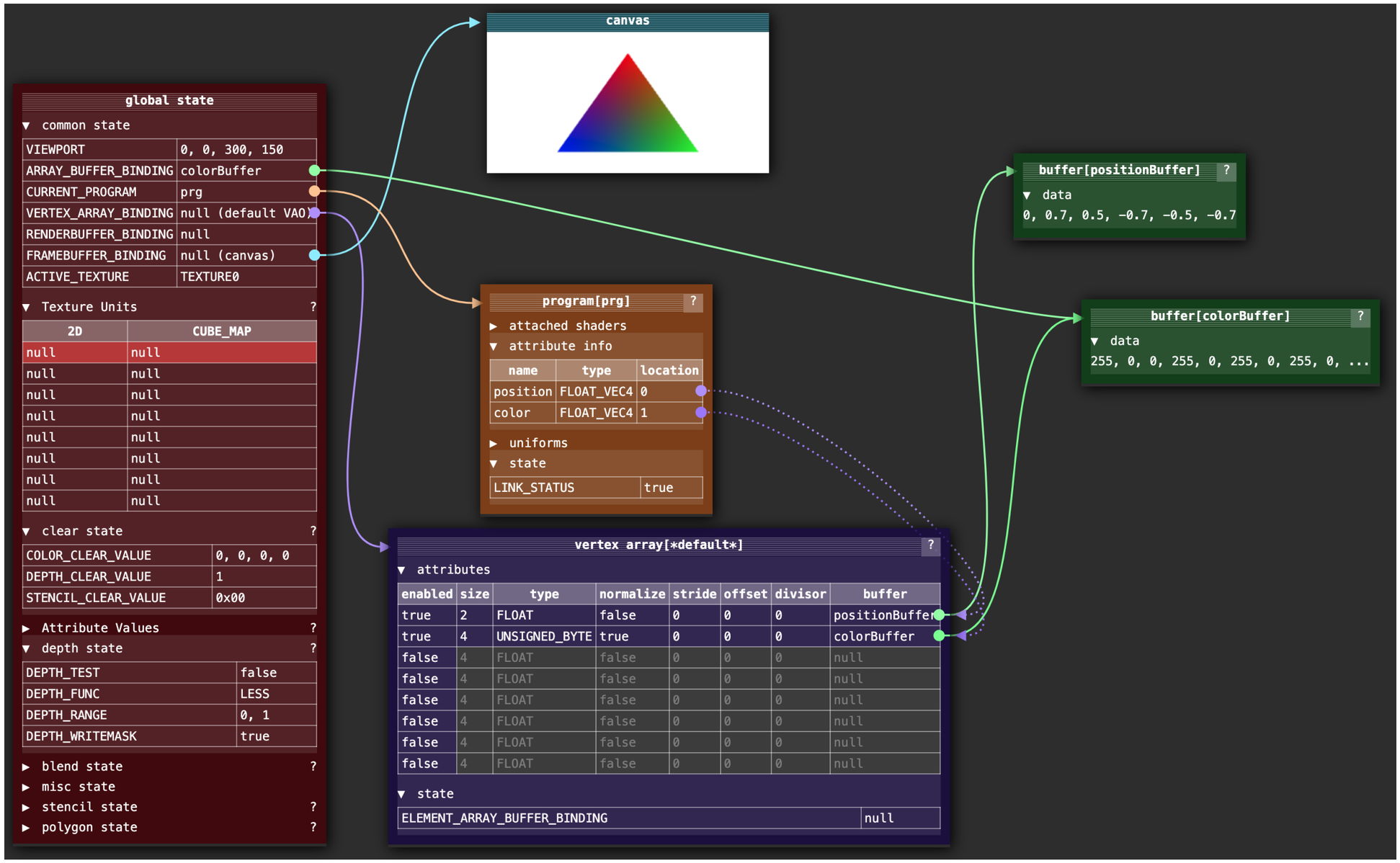
Voting as Anonymous Send

Acceptable Use - Slido Privacy

Hint: Open a console and retrieve a `WebGL` context:

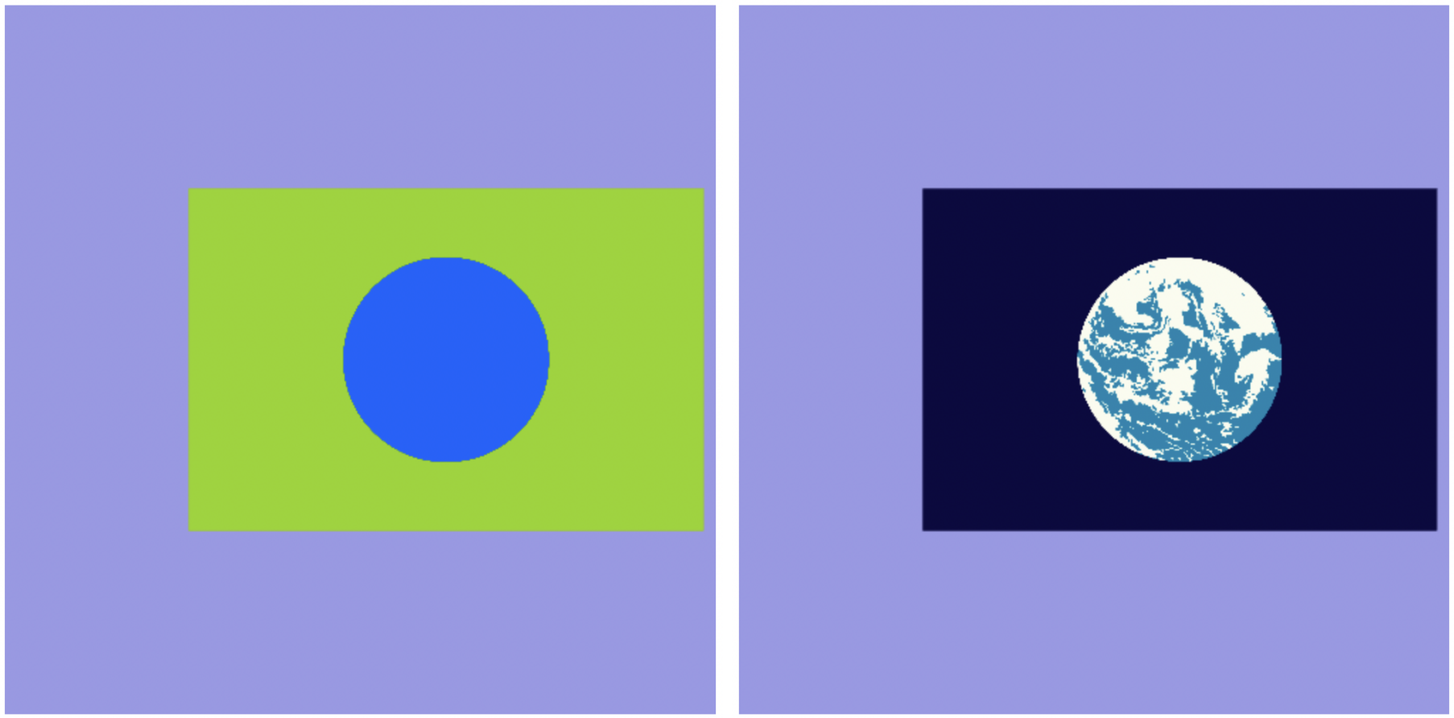
```
const gl = document.createElement("canvas").getContext("webgl");
```

WebGL is a state machine.



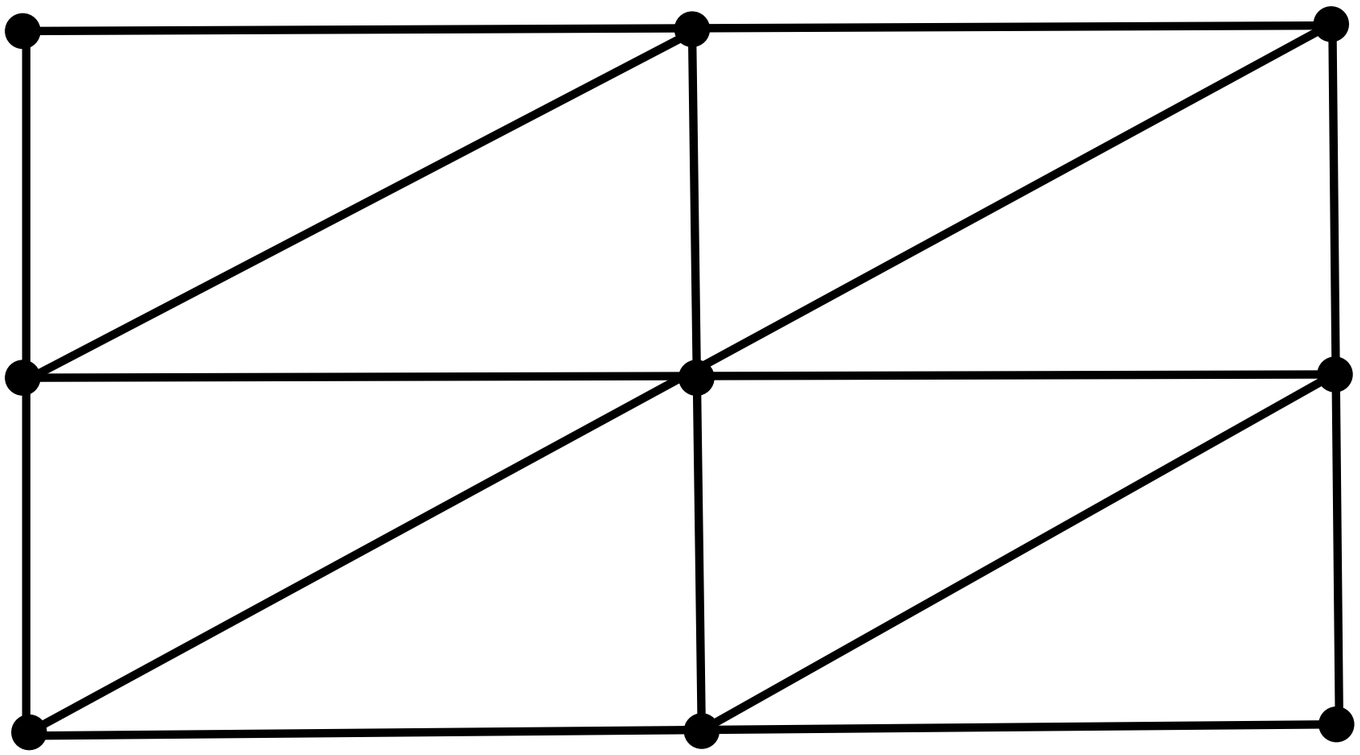
How to develop a `WebGL` application is best learned by writing code.

Open the `exercise` in the row for today's class.
Since it's Earth day, we will render some Earth flags.



Solution is already linked at the end of Chapter 10 (click the link and then **View Page Source**).

Rendering a rectangle defined by 8 triangles and 9 vertices.



Sketch what this will look like using the `colors` array attached to the mesh vertices.

Texturing with WebGL.

In JavaScript:

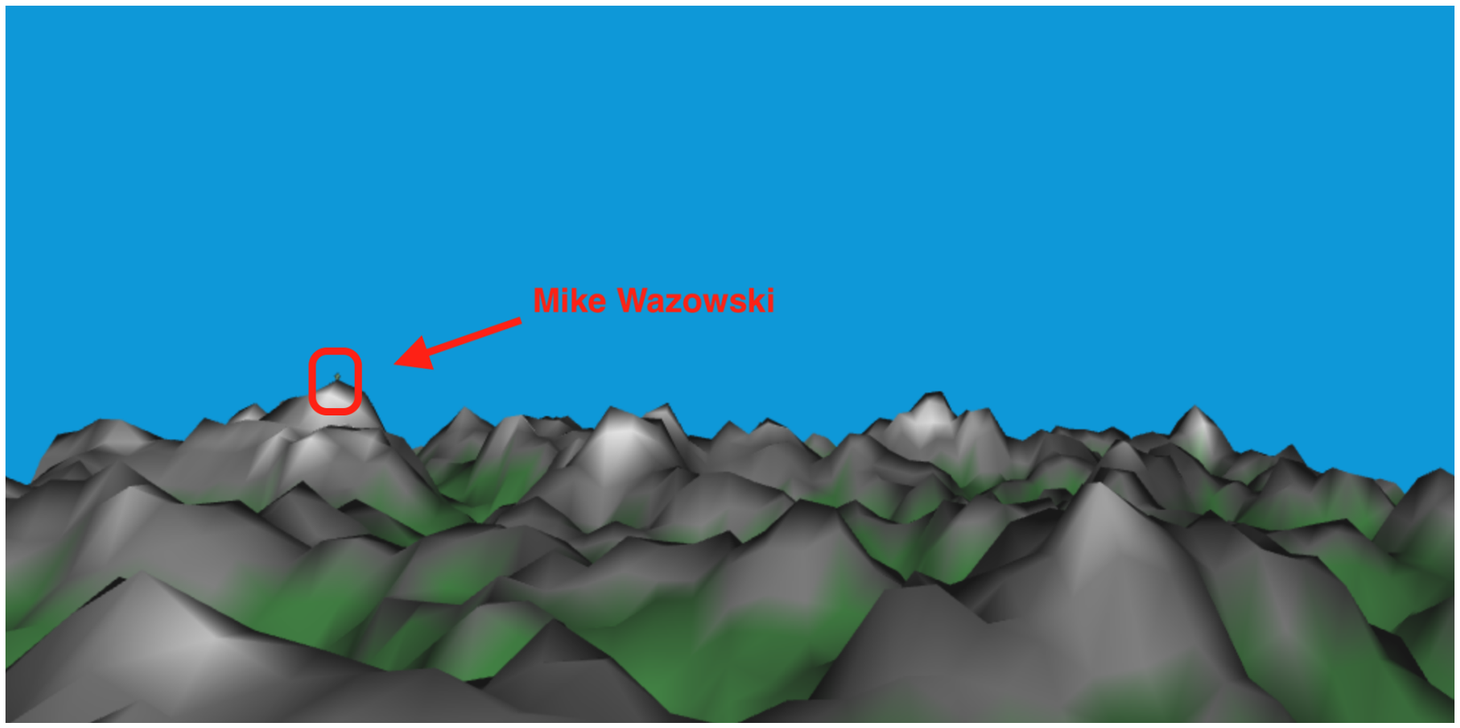
```javascript
// retrieve the image
let image = document.getElementById("flag-texture");

// create the texture and activate it
let texture = gl.createTexture();
gl.activeTexture(gl.TEXTURE0); // <-- important! make a note of N in gl.TEXTUREN
gl.bindTexture(gl.TEXTURE_2D, texture);

// define the texture to be that of the requested image
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGB, gl.RGB, gl.UNSIGNED_BYTE, image);

// set filter parameters
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);

// tell webgl which texture index to use for the uniform sampler2D in the shader
gl.uniform1i(gl.getUniformLocation(program, "tex_Image"), 0);
```

In fragment shader:

```javascript
precision highp float;

uniform sampler2D tex_Image;

varying vec2 v_TexCoord; // Option 1: if using texture coordinates attached to vertices

void main() {
  float s = ...; // Option 2: if computing your own texture coordinates (e.g. on a sphere)
  float t = ...;
  vec3 km = (texture2D(tex_Image, v_TexCoord)).rgb; // Option 1
  //vec3 km = (texture2D(tex_Image, vec2(s, t))).rgb; // Option 2
  gl_FragColor = vec4(km, 1.0);
}
```

Summary

1. Create a `WebGLShaderProgram` from a vertex shader and fragment shader (each a `WebGLShader`).
2. Write data to the GPU using `WebGLBuffer` (`createBuffer`, `bindBuffer`, `bufferData`).
3. Enable the attributes you need in your program (`getAttribLocation`, `enableVertexAttribArray`).
4. Draw:
    - Associate attributes with the buffers you want to use in the pipeline (`bindBuffer`, `vertexAttribPointer`).
    - Set the global state (`viewport`, `enable`, `clear`).
    - Set uniform variables (`getUniformLocation`, `uniform[1234][fi][v]` and `uniformMatrix[234][fi]v`).
    - Invoke the pipeline (`bindBuffer` for the element indices, then `drawElements`).
5. Go to Step 4 whenever a user interacts with the application, or if animating.



`WebGL` debugging checklist.

- ☐ Do your shaders compile without errors?
- ☐ Did you declare vertex shader inputs with `attribute`? Did you enable them on the `JavaScript` side? Did you use the correct string name and location when enabling the attribute?
- ☐ Did you declare `varying` variables in both shaders? Did you set the value of the `varying` in your vertex shader?
- ☐ Did you upload the right data when uploading data to the GPU (check which variable you passed to `bufferData`)? Are you using the correct buffer type? Are you using the correct typed array?
- ☐ Did you bind to the intended buffer? Check for instances of accidentally typing `gl.ARRAY_BUFFER` when you intended `gl.ELEMENT_ARRAY_BUFFER` and vice versa.
- ☐ Did you specify the right type and stride when associating a buffer with an attribute?
- ☐ Check the order of the arguments in `vertexAttribPointer`, `bufferData` and `drawElements`.
- ☐ Check again for instances of accidentally writing `gl.ARRAY_BUFFER` when you intended `gl.ELEMENT_ARRAY_BUFFER`.
- ☐ Check again for spelling of all variables in `JavaScript` and `GLSL` code.