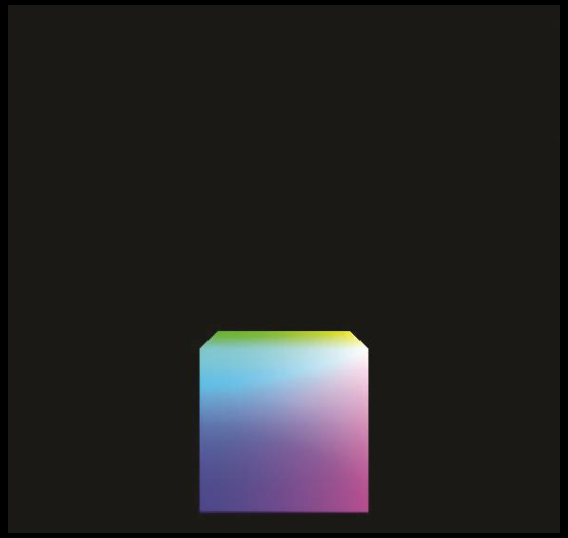
BUFFERS AND VERTEX ATTRIBUTES
OR
LET'S DRAW SOMETHING FINALLY!



OUTLINE

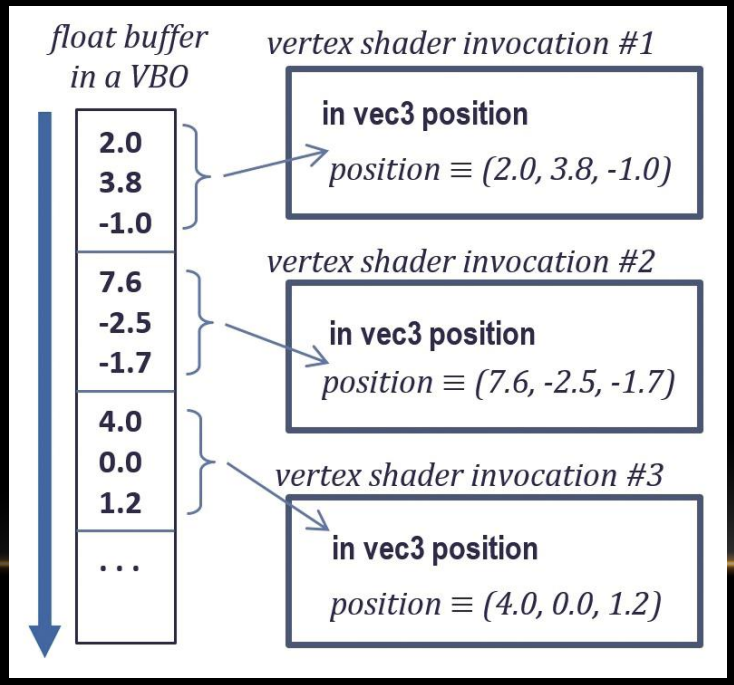
- Buffers and vertex attributes
- Uniform variables
- Interpolation of vertex attributes

BUFFERS AND VERTEX ATTRIBUTES

- Vertices of objects must be available to the vertex shader
  - Until now, we have defined the vertices inside the shader
  - Usually this is done once, on the Java/JOGL side
    - Since only once, usually done in init()
- For every frame (every time the picture changes)
  - Enable the buffer containing the vertices
  - Associate the buffer with a vertex attribute
  - Enable the vertex attribute
  - Call glDrawArrays(…)

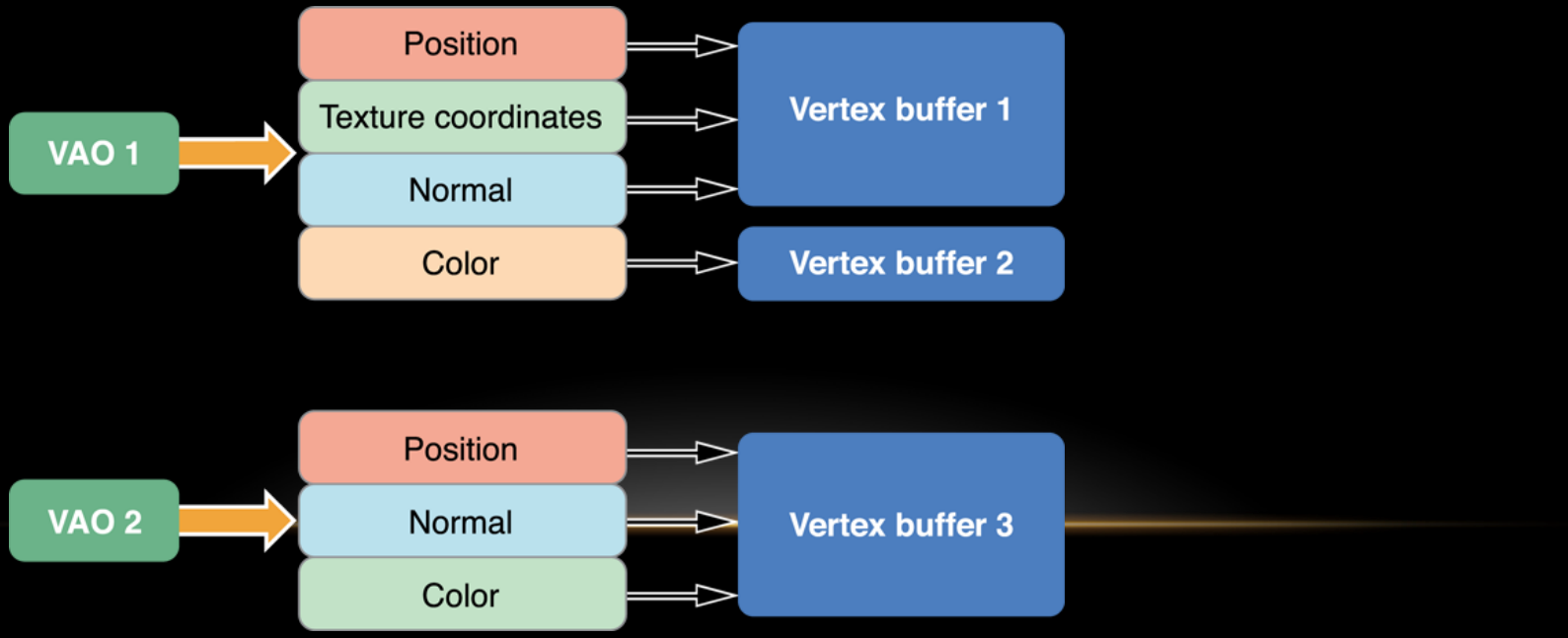
OPENGL BUFFERS

- Buffers are contained in a VBO – Vertex Buffer Object
- Most scenes are going to have several (many?) objects
  - Will need a VBO for each object



VERTEX ARRAY OBJECTS

- OpenGL requires at least one vertex array object (VAO)
- VAO allows you to organize multiple buffer objects in one structure
  - Makes it easier to manipulate multiple objects in a scene



VBO AND VAO JAVA/JOGL CODE

```
// These will hold the addresses of the buffer and array objects
private int vao[] = new int[1];
private int vbo[] = new int[2];
…
// Create 1 VAO, store its address in vao, offset of 0
gl.glGenVertexArrays(1, vao, 0);
// Bind the vao so that is active
gl.glBindVertexArrays(vao[0]);
// Create 2 VBOs, store their addresses in vbo, offset of 0
gl.glGenBuffers(2, vbo, 0);
```

VAO AND VBO GLSL CODE

In the vertex shader:

```
layout (location = 0) in vec3 position
```

- layout (location = 0) describes how the vertex attribute and the buffer will be associated together
- in means this is an input variable (coming in from the Java/JOGL code)
- vec3 it is a 3 element vector – each time the shader is run it will grab three float values
- position the variable name

PUTTING VERTEX DATA INTO A BUFFER

- So far, all we've done is set up the VBO's and the VAO
- Still need to put vertex data into the buffer
- Assume we have the data in an array called vPositions:

```
// Make the first vbo active
gl.glBindBuffer(GL_ARRAY_BUFFER, vbo[0];
// Create a FloatBuffer from the vertex positions
FloatBuffer vBuf = Buffers.newDirectFloatBuffer(vPositions);
// Copy the FloatBuffer data into the active buffer
gl.glBufferData(GL_ARRAY_BUFFER, vbuf.limit()*4, vbuf,
                GL_STATIC_DRAW);
```

CAN WE SEE THE OBJECT YET?

- No…
- All the previous code has been to set up the buffers
  - It is only needed once, so it is either in the init() method or in a method called by init()
- In the display() method, we need to set up the rest of the Java/JOGL code that will be called for each frame:

```
// Make the first vbo active
gl.glBindBuffer(GL_ARRAY_BUFFER, vbo[0];
// Associate the 0th vertex attribute with the active buffer
gl.glVertexAttribPointer(0, 3, GL_FLOAT, false, 0, 0);
// Enable the 0th vertex attribute
gl.glEnableVertexAttribArray(0);
```

ARE WE THERE YET?

- Sigh… No…
- Remember all the math from the last lectures?
  - Yeah. Now we need that.
- Recall that the model-view matrix holds the objects' transformation parameters along with any camera positioning transformations
  - One matrix because you multiply (concatenate) them all together
- You also need a projection matrix to describe you want to go from 3D to 2D
- These matrices are usually defined in the Java/JOGL code and passed in to the vertex shader

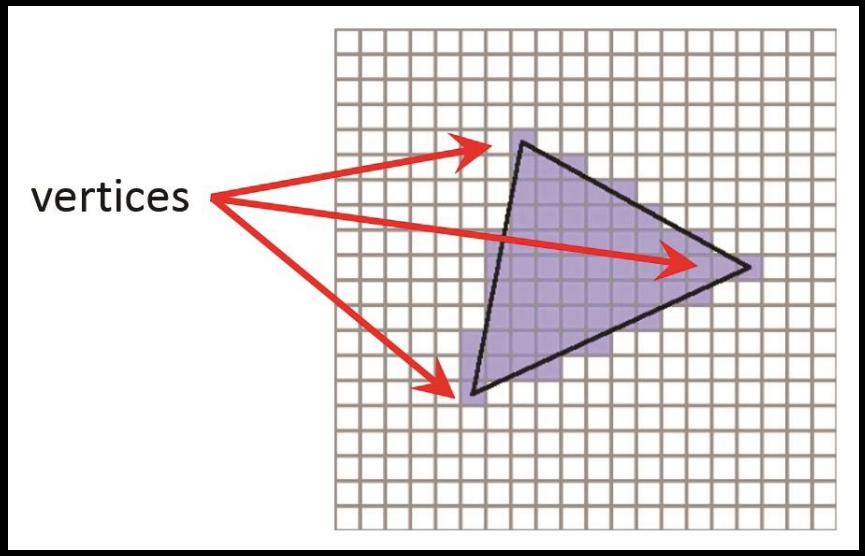
JAVA/JOGL AND GLSL CODE

```
// On the Java side, assuming you have matrices mvMat and Pmat,
//   the following gets the memory address so you can transfer these
//   to the vertex shader
int mv_loc = gl.glGetUniformLocation(rendering_program,
                                     "mv_matrix");
int proj_loc = gl.glGetUniformLocation(rendering_program,
                                       "proj_matrix");
gl.glUniformMatrix4fv(mv_loc, 1, false, mvMat.getFloatValues(), 0);
gl.glUniformMatrix4fv(proj_loc, 1, false, pMat.getFloatValues(), 0);
// On the GLSL side, in the vertex shader
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
```

UNIFORM VARIABLES

- When we send vertex attributes down the pipeline:
  - the vertex shader sets positions (and possibly other things…)
  - the rasterizer interpolates attributes between vertices
  - the fragment shader determines which are potential pixels
- Vertex data goes into the shader as an “in” data type
- Uniform variables are different
  - They are not interpolated, like vertex data
  - Treated as a constant for each vertex in a buffer
- Can also have “out” data types
  - Not necessary to use “out” for vertex shader positions – gl_position is a built-in variable (global) so it doesn’t need to be “output”

INTERPOLATION



NO

SUMMARY

- Buffers and vertex attributes
- Uniform variables
- Interpolation of vertex attributes