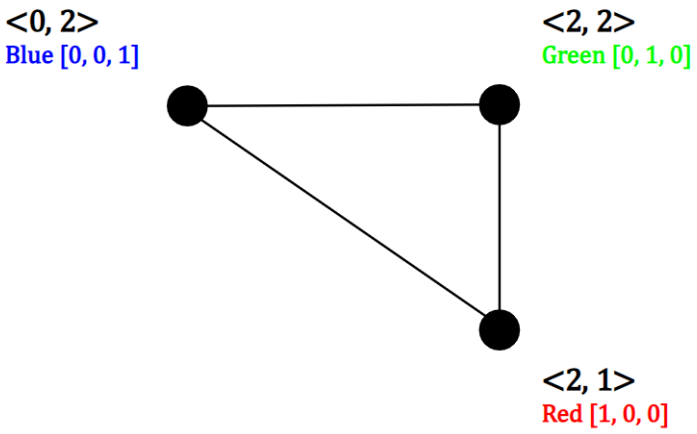
1 Introduction to Shader Programs

We'll begin by showing the process behind drawing a triangle. Consider the following triangle:



Here, we want to make the vertex at:

- $\langle 2, 1\rangle$ red. We'll call this vertex R.
- $\langle 2, 2\rangle$ green. We'll call this vertex G.
- $\langle 0, 2\rangle$ blue. We'll call this vertex B.

Note that the tuple next to the color, $[R, G, B]$, is the RGB color specification. Then, when we actually draw this triangle, the color at any particular point will just be the averages of the other base colors. For examples:

- The vertex directly in the center (the midpoint) of B and R would have the averages of the two RGB values of the two vertices; in other words, at that particular point, we would have the RGB value $\langle \frac{1}{2}, 0, \frac{1}{2}\rangle$ (dark magenta).
- The vertex directly in the center (the midpoint) of B and G would have the RGB value $\langle 0, \frac{1}{2}, \frac{1}{2}\rangle$ (cyan).
- The vertex directly in the center of G and R would have the RGB value $\langle \frac{1}{2}, \frac{1}{2}, 0\rangle$ (yellow).
- The vertex directly in the center of all three vertices R, G, and B (i.e. at the center of the triangle) would have the RGB value $\langle \frac{1}{3}, \frac{1}{3}, \frac{1}{3}\rangle$ (gray).

We'll now see how the corresponding C++ code works.

1. Build a C++ array of vertex attributes[1]. That is, the position and color.

```
float verts[] = {
    2, 1, // bottom-right vertex position; point (2, 1)
    1, 0, 0, // color of vertex at (2, 1)
    2, 2, // top-right vertex position; point (2, 2)
    0, 1, 0, // color of vertex at (2, 2)
    0, 2, // top-left vertex position; point (0, 2)
    0, 0, 1 // color of vertex at (0, 2)
};
```

2. We now load the data from the C++ array into the OpenGL buffers on the GPU. We'll just explain a subset of the commands due to there being many commands.

[1] Numeric properties of the vertex.

```
glBufferData(GL_BUFFER_ARRAY, sizeof(verts), verts, GL_STATIC_DRAW);
```

What `glBufferData` does is it loads the C++ array into the OpenGL buffer. The parameters are as follows:

- `GL_BUFFER_ARRAY`: This tells the program where to send the data to. This is sometimes called the Vertex Buffer Object.
- `sizeof(verts)`: The amount of data to send, in bytes.
- `verts`: The data to send. This starts from the beginning of the array and sends `sizeof(verts)` data (the previous parameter).
- `GL_STATIC_DRAW`: A "hint" telling OpenGL how your data should be used. Here, we're saying that this is some fixed data that we won't be changing much.

All we did was send a bunch of data to the buffer. We now need to tell OpenGL what the data is.

```
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 5 * sizeof(float), (void*) 0);
```

This command tells OpenGL *what exactly* we sent. The parameters are:

- `0`: The location which holds the xy coordinates.
- `2`: The number of data items that are a part of this location. In this particular example, this is x and y.
- `GL_FLOAT`: We're adding floating point numbers. Pairing this with the previous parameters, this means that we're adding two floating point numbers.
- `5 * sizeof(float)`: The stride, or the *step*. In this case, we're essentially saying that

$$\{\underbrace{2,1,1,0,0}_{\text{Stride}},\underbrace{2,2,0,1,0}_{\text{Stride}},\underbrace{0,2,0,0,1}_{\text{Stride}}\}$$

  Put it another way, we're just telling OpenGL what set of data represents a vertex.
- `(void*) 0`: The starting position, or the first xy.

Now, we need to turn this on. So, we have the command:

```
glEnableVertexAttribArray(0);
```

So, the previous command tells OpenGL where the data starts. This command tells OpenGL to use that data at the specified location (the parameter). Here, the parameter is:

- `0`: Location 0. In other words, this parameter is the same value as the first parameter in the previous command. This tells OpenGL to use the data at location 0.

Note that the above was for xy values. How would we change this for the color values? In particular:

- We now need to tell OpenGL what the data is.

  ```
  glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 5 * sizeof(float), (void*) 0);
  ```

  This command tells OpenGL *what exactly* we sent. The parameters are:
  - `0`: The location which will hold the RGB values.
  - `2`: The number of data items that are a part of this location. In this particular example, this is x and y.
  - `GL_FLOAT`: We're adding floating point numbers. Pairing this with the previous parameters, this means that we're adding two floating point numbers.

- `5 * sizeof(float)`: The stride, or the *step*. In this case, we're essentially saying that

$$\{2, 1, \underbrace{1, 0, 0, 2, 2}_{\text{Stride}}, \underbrace{0, 1, 0, 0, 2}_{\text{Stride}}, 0, 0, 1\}$$

  Put it another way, we're just telling OpenGL what set of data represents the colors.
- `(void*) 2 * sizeof(float)`: The starting position, or the position of the first set of RGB values.

Now, we need to turn this on. So, we have the command:

```
glEnableVertexAttribArray(1);
```

So, the previous command tells OpenGL where the data starts. This command tells OpenGL to use that data at the specified location (the parameter). Here, the parameter is:

- `1`: Location 1. In other words, this parameter is the same value as the first parameter in the previous command. We're telling OpenGL to read the data from location 1.

3. Compile and link the shadow programs. The shader program consists of two different types of shaders.

- Vertex Shader: Runs on each vertex.

```
// GLSL Version
#version 330 core
layout(location = 0) in vec3 vertPos;
```

Here, this is saying that we have a variable `vertPos` which is a `vec3` (a vector of 3 floating point numbers), identified at location 0, which we described in the previous step. `vertPos` described the vertex's position. Note that, in the VBO, we only gave 2 values; thus, `z` will be set to 0.

```
layout (location = 1) in vec3 color;
```

This is similar to the previous line of code, except that we're defining this for RGB values instead.

```
out vec3 theColor;
```

We're now outputting a `vec3` called `theColor`.

```
void main() {
    gl_Position = vec4(vertPos.x - 1, vertPos.y - 1, 0, 1);
```

Here, we're outputting a vector with four components (it'll always be a `vec4`); an x, y, z, and w component, respectively. Note that $x, y, z, w \in [-1, 1]$. The reason why we subtract 1 from the x, y coordinates is because we have vertices whose x- and y-coordinates are in the range $[0, 2]$.

```
    theColor = color;
}
```

We just keep the color positions as is. This concludes this particular program; note that this program is executed once per vertex.

- Fragment Shader: This is run once for every pixel covered by the triangle. This is given by:

```
// GLSL Version
#version 330 core
in vec3 theColor;
```

Here, the input `vec3` is given by the output `vec3`.

```
out vec4 pixelColor;
```

This defines the output color of the pixel.

```
void main() {
    pixelColor = vec4(theColor, 1);
}
```

Here, the `1` indicates no transparency; this is sometimes known as the alpha value.

4. Draw the triangle using a C++ command.

```
glDrawArrays(GL_TRIANGLE, 0, 3);
```

Here:

- `0` means to start with vertex 0 in the array of data that was uploaded.
- `3` means the number of vertices to process.

This tells the shader program to render the triangle.

2 Smooth & Flat Shading

By default, OpenGL uses what we call *smooth shading*. This means that we average colors across the triangles. This is very similar to what we talked about earlier with the shading of the pixels based on the position of the pixel. *Note* that the vertex shader and fragment shader program that we talked above above is the same here.

Flat shading is a bit different: the color of the *last vertex* is used throughout the triangle. The vertex shader and fragment shader programs are a bit different; in particular, it's shown like so, respectively:

```
#version 330 core
layout(location = 0) in vec3 vertPos;
layout(location = 1) in vec3 vertColor;
flat out vec3 theColor; // The change
main() {
    gl_Position = vec4(vertPos, 1);
    theColor = vertColor;
}
```

and

```
#version 330 core
flat in vec3 theColor;
out vec4 pixelColor;
main() {
    pixelColor = theColor;
}
```