2D Triangle Color Per Vertex

Table of Contents

Chapter 1. 2D Triangle Color Per Vertex

1.1. Summary

This example renders a single triangle with different colors for each vertex, on a red background every frame. The vertex colors are interpolated across the triangle by the rasterizer.

1.1.1. Shader loading

In our parent example (2D_myFirstTriangle) our Vertex and Fragment shaders were each defined as a `std::string` directly in our C++ program. It's better practice to load our GLSL shader from files. This gives a better program structure. As shaders are loaded, compiled, and linked at run-time, it also allows us to change what our shaders do without re-building our C++ program. To read files we use `fstream` from the standard library (`#include <fstream>`), and add a simple `loadShader` function.

```
std::string loadShader(const string filePath) {
    std::ifstream fileStream(filePath, std::ios::in | std::ios::binary);
 if (fileStream)
 {
  string fileData( (std::istreambuf_iterator<char>(fileStream)),
                   (std::istreambuf_iterator<char>()          ));

  cout << "Shader Loaded from " << filePath << endl;
  return fileData;
 }
 else
 {
        cerr << "Shader could not be loaded - cannot read file " <<
 filePath << ". File does not exist." << endl;
        return "";
 }
}
```

1.1.2. Fragment Shader

The color of fragments is determined by the Fragment Shader. In order to color our triangle appropriately (with different colors per vertex, smoothly interpolated across the triangle) we need more information coming in to the fragment shader - we want to pass in the color for each fragment.

Input variables are passed to the Fragment Shader from the Rasteriser, which creates them from Output variables from the previous stage, which is usually the Vertex Shader. The Rasterizer, by default, interpolates these variables across the surface of the primitive, though this can be disabled if required.

Let's update our Fragment Shader to this effect

```
#version 330
in vec4 fragmentColor;
out vec4 outputColor;
void main()
{
  outputColor = fragmentColor;
}
```

1.1.3. Vertex Shader

In order for the Fragment Shader to receive more information, we need to output more information from the Vertex Shader. This is done with an output variable, which **MUST** have the same **name** and **signature** as the input variable to the Fragment Shader - except `in` should be `out`.

Our Vertex Shader could now look as follows. We'll hard-code the output variable for now. Let's use just Blue, to make it clear this is working.

```
#version 330
in vec2 position;
out vec4 fragmentColor;
void main()
{
     gl_Position = vec4(position, 0.0, 1.0);
     fragmentColor = vec4(0.0, 0.0, 1.0, 1.0);
}
```

Next we need to pass more information into the Vertex Shader. Uniform variables as we've used before are the same for all vertices, we need to have different color values for each vertex, just like we have different position values. This is what **Attributes** do.

Let's add that to our Vertex Shader, and pass that incoming value onwards.

```
#version 330
in vec2 position;
```

```
in vec4 vertexColor;
out vec4 fragmentColor;

void main()
{
  gl_Position = vec4(position, 0.0, 1.0);
  fragmentColor = vertexColor;
}
```

1.1.4. Vertex Data

That was easy enough, but if you run it you'll notice that the triangle is now Black. That's because the incoming attribute `vertexColor` hasn't been set by **our** C++ program, so `vertexColor` is still the default value for a vec4 ({0.0, 0.0, 0.0, 0.0}). We'll have to change our C++ program to provide this information.

The first thing we need is the color information in our C++ program. There are three main approaches to this:

1. have an array for position and another array for color
2. have a single array for both, with all the positions data, then all the color data
3. have a single array for both, with all the information for each vertex together, then the next vertex. This is called **INTERLEAVING** - the color information is interleaved with the position information.
    - we'll take this approach.

Our variable `vertexData` will now look like this:

vertexData

```
//the data about our geometry
const GLfloat vertexData[] = {
//  X        Y          R     G     B      A
  0.000f,  0.500f,    1.0f, 0.0f, 0.0f,  1.0f,
 -0.433f, -0.250f,    0.0f, 1.0f, 0.0f,  1.0f,
  0.433f, -0.250f,    0.0f, 0.0f, 1.0f,  1.0f
};
```

1.1.5. Attribute location

Everything we've been doing for the position attribute we also need to do for the color attribute.

We need to have a `GLint` to store in C++ the location of the `vertexColor` attribute in our GLSL.

GLVariables

```
//our GL and GLSL variables
GLuint theProgram; //GLuint that we'll fill in to refer to the GLSL
 program (only have 1 at this point)
int positionLocation; //GLuint that we'll fill in with the location of the
 `position` attribute in the GLSL
GLint vertexColorLocation; //GLuint that we'll fill in with the location
 of the `vertexColor` attribute in the GLSL

GLuint vertexDataBufferObject;
GLuint vertexArrayObject;
```

We then need to ask OpenGL for that location once we've compiled and linked our GLSL.

initializeProgram

```
void initializeProgram()
{
 std::vector<GLuint> shaderList;

 shaderList.push_back(createShader(GL_VERTEX_SHADER,
 loadShader("vertexShader.glsl")));
 shaderList.push_back(createShader(GL_FRAGMENT_SHADER,
 loadShader("fragmentShader.glsl")));

 theProgram = createProgram(shaderList);
 if (theProgram == 0)
 {
  cerr << "GLSL program creation error." << std::endl;
  SDL_Quit();
  exit(1);
 }
 else {
  cout << "GLSL program creation OK! GLUint is: " << theProgram <<
 std::endl;
 }

 positionLocation = glGetAttribLocation(theProgram, "position");
 vertexColorLocation = glGetAttribLocation(theProgram, "vertexColor");
 //clean up shaders (we don't need them anymore as they are no in
 theProgram
```

```
 for_each(shaderList.begin(), shaderList.end(), glDeleteShader);
}
```

1.1.6. glVertexAttribPointer

We need to tell OpenGL to use that location, and how to fill it from the buffer object. **VertexArray** objects store this information, which we then use when we render. This change is pretty important. Look up the specification for `glVertexAttribPointer`, and figure out what each parameter is doing.

initializeVertexArrayObject

```
//setup a GL object (a VertexArrayObject) that stores how to access data
 and from where
void initializeVertexArrayObject()
{
 glGenVertexArrays(1, &vertexArrayObject); //create a Vertex Array Object
 cout << "Vertex Array Object created OK! GLUint is: " <<
 vertexArrayObject << std::endl;

 glBindVertexArray(vertexArrayObject); //make the just created
 vertexArrayObject the active one

  glBindBuffer(GL_ARRAY_BUFFER, vertexDataBufferObject); //bind
 vertexDataBufferObject

  glEnableVertexAttribArray(positionLocation); //enable attribute at index
 positionLocation
  glEnableVertexAttribArray(vertexColorLocation); //enable attribute at
 index vertexColorLocation

  glVertexAttribPointer(positionLocation,    2, GL_FLOAT, GL_FALSE, (6
* sizeof(GL_FLOAT)), (GLvoid *) (0 * sizeof(GLfloat))); //specify that
 position data contains four floats per vertex, and goes into attribute
 index positionLocation
  glVertexAttribPointer(vertexColorLocation, 4, GL_FLOAT, GL_FALSE, (6
* sizeof(GL_FLOAT)), (GLvoid *) (2 * sizeof(GLfloat))); //specify that
 position data contains four floats per vertex, and goes into attribute
 index vertexColorLocation

 glBindVertexArray(0); //unbind the vertexArrayObject so we can't change
 it

 //cleanup
```

```
 glDisableVertexAttribArray(positionLocation); //disable vertex attribute
 at index positionLocation
 glBindBuffer(GL_ARRAY_BUFFER, 0); //unbind array buffer

}
```

1.1.7. Render

Our Render function is unchanged, except for removing the unused **uniform** variable for color. At this point nothing is changing between frames, but we could add some new, useful, uniform variables to allow change.

render

```
void render()
{
 glUseProgram(theProgram); //installs the program object specified by
 program as part of current rendering state

 //load data to GLSL that **may** have changed

 glBindVertexArray(vertexArrayObject);

 glDrawArrays(GL_TRIANGLES, 0, 3); //Draw something, using Triangles, and
 3 vertices - i.e. one lonely triangle

 glBindVertexArray(0);

 glUseProgram(0); //clean up

}
```