CG CSE 533 Lab 02

Introduction to 3D

September 7, 2022

1 Introduction

In this lab you'll be learning how to communicate with the shader programs and the usage of glm for matrix transformations. Also as an extension we will look into how 3D objects are created using triangles. Further we will look into how fragment shader can be used to interpolate colors and how the underlying communication takes place between fragment shader and vertex shader.glm provides several matrix/vector based data structures and enables manipulation by providing pre-built functions for matrix transformation such as rotation, translation, etc. It further facilitates common matrix operations such as matrix multiplication, addition etc.

1.1 Lab Code instructions

Build the program by running make and run Lab2 executable thus generated.

1.1.1 Display a 3D object

Until now we have only seen how to render 2D objects such as square or a triangle. Now as an extension to we learn how to display a 3D object. To do this we need to extend our previous know to generate triangles arranged in a 3D space i.e instead of the points being part of R^2 the points are now part of R^3 . The coordinates of each triangle have to be selected such they form a 3D object for eg. a cube. In order to make a cube consisting of triangles in R^3 . We can divide a cube into smaller triangle just as we did to make a square. Each face is now composed of 2 smaller triangles joined along the diagonal to give a face of the cube. Since a cube has 6 faces, each face is composed of 2 triangles, a total of 6×2=12 triangles would be needed to render a cube. For each triangle 3 coordinates are needed for each vertex thus the total number of points being $3 \times 3 \times 12$ as marked in the code for each triangle beginning and ending.

1.1.2 Passing values to shaders

Instead of passing a particular color value as done in previous lab we will look into how we can assign different colors depending on the vertex location. Previously after `glUseProgram(myShaderProgram);` you can fetch the location of the uniform variable using:

```
int location = glGetUniformLocation(program, "ourColor");
//glGetUniformLocation takes the program and the variable name to fetch location}
```

Once the location has been obtained, you can check it if the location is a non- negative integer, for a valid location. You can pass the data using a variant of glUniform(4f/3f/3fv etc). and pass the location along with values to it. You can find more variants and data type supported by it from here. However now we need to assign the values to each coordinate of the triangle and see how the fragment shader automatically interpolates between these values. To do this we create a separate VBO for colors to store the per vertex color information. In the vertex shader we create a new variable that will store the vertex information for color. We need then create a color array which holds the RGB information for every vertex of the cube. To make it more vibrant we assign a unique color to every point of all the rendered triangles. Next we need to pass this value to vertex shader. So to do it we do the following -

```
GLuint colours_vbo;
glGenBuffers( 1, &colours_vbo );
glBindBuffer( GL_ARRAY_BUFFER, colours_vbo );
glBufferData( GL_ARRAY_BUFFER, sizeof(colors), colors, GL_STATIC_DRAW );
//create buffers to store colors and load
//the color values in the buffer.
```

To activate these colors we enable the Vertex ArrtibPointer associated to this in the vertex shader using -

```
glBindBuffer( GL_ARRAY_BUFFER, colours_vbo );
glVertexAttribPointer( 1, 3, GL_FLOAT, GL_FALSE, 0, NULL );
glEnableVertexAttribArray( 1 );
```

This will bind the color values to the variable vertex colour in the vertex shader since it is mapped to location 1 by `glVertexAttribPointer`. Now since fragment shader is responsible for assigning colors this is passed to the fragment shader by an out variable of vertex shader.

```
out vec3 colour;
```

which then stores the color value as shown `colour = vertex colour`; This will pass the value to the fragment shader for interpolation. The fragment shader captures this as an in variable and assigns it to the output variable `out vec4 FragColor`. Keep in mind that the supplied color contained had only RGB values thus a vec3 but `FragColor` is a vec4 and also stores the alpha channel (transparency) information.

1.1.3 Translation

Translation is the process of adding another vector on top of the original vector to return a new vector with a different position, thus moving the vector based on a translation vector.

```
glm::vec4 vec(1.0f, 0.0f, 0.0f, 1.0f);
glm::mat4 trans = glm::mat4(1.0f);
trans = glm::translate(trans, glm::vec3(1.0f, 1.0f, 0.0f));
```

1.1.4 Rotation

glm provides several matrixvector based data structures and enables manipulation by providing pre-built functions for matrix transformation such as rotation, translation, etc. It further facilitates common matrix operations such as matrix multiplication, addition etc. You can find more supported operations from here and here. An important note is that OpenGL stores matrices in column major format. For programming purposes, OpenGL matrices are 16-value arrays with base vectors laid out contiguously in memory. The translation components occupy the 13th, 14th, and 15th elements of the 16-element matrix, where indices are numbered from 1 to 16 for a 4X4 matrix. So if you need a C-style matrix implementation you will need to track the indices appropriately.

```
glm::mat4 trans = glm::mat4(1.0f);
trans = glm::rotate(trans, glm::radians(90.0f), glm::vec3(0.0, 0.0, 1.0));
```

1.1.5 Scaling

Scaling a vector we are increasing the length of the arrow by the amount we'd like to scale, keeping its direction the same. Since we're working in either 2 or 3 dimensions we can define scaling by a vector of 2 or 3 scaling variables, each scaling one axis (x, y or z).

```
trans = glm::scale(trans, glm::vec3(0.5, 0.5, 0.5));
```

2 Deliverables

The task for the lab is to refer to the lab code and docs provided here and submit an implementation for continuously rotating the cube. The slider implementation has been provided along with the time information obtained by glfwGetTime. Change the code such that for every 0.5 secs the cube rotates by 1 radians. (Hint: overwrite the rotation matrix values by checking the timing value obtained.) Submit the output along with code for evaluation. Further record screen and upload a gif/video demonstrating the widgets. You can use Kazam to record. Upload the zip file of code and output. Name the zip file as Lab02_<name_ roll no>.zip