Assignment 02 - Report.

Yelugoti Mohana Datta

Contents

1 Problem Statement

3D Rendering Using MVC Architecture

Rendering a 3D Scene using more than two objects in the scene, the objects will be rendered using their surface meshes given in .ply file format.

The Rendered scene should be able to do following things:

- We should be able to add new objects to the scene from .ply files.

- Scene should support object centric transformations i.e, translation, rotation, scaling. These actions should be associated with keyboard keys. Rotation should be implemented using quaternions.
- Color of each vertex should be it's vertex normal, which can be found by averaging the normals of the faces sharing the vertex.
- Implement splat for each triangle in the mesh. The mesh should be colored using the normal co-ordinates.
- Implement lightning, the light source should be movable when certain keyboard keys are pressed.

2 Approach

2.1 Code Design:

2.1.1 Model part:

Each time we add a .ply file, a model is constructed from the data in the file and stored in a Mesh object. This Mesh object contains information about number of vertices, faces etc

To change anything related to a model, that message has to pass via its mesh object.

Some Brief explanation of mesh's methods are:

- To get information about model:

```
int getVerticesNum();
int getFacesNum();

// Get Face type of the model, currently only FaceType 3 is
// supported.
int getFaceType();
```

- Utility functions:

```
// Subtracts given two vectors.
glm::vec3 subVectors(glm::vec3 a, glm::vec3 b);
```

```
// Given three vertices, calculates normal of the triangle formed
// by those vertices.
glm::vec3 triangleNormal(glm::vec3 a, glm::vec3 b, glm::vec3 c);

// Calcultes a coordinates of cube surrounding the mesh.
// this is done after normalising the mesh.
void initializeCube();
```

- Transformation functions:

```
// Translation functions.
void translateLeft();
void translateRight();
void translateUp();
void translateDown();

// Rotation functions implemented using quartenions.
void rotateLeft();
void rotateRight();

// Given an angle and axis, create rotationMatrix
// (Uses quarternions for this purpose).
void rotateAngle2(GLfloat angle, glm::vec3 axis);

// Given start vertex, and end vertex does the rotation
// Used for trackball kind implementation.
void rotateAngle(glm::vec3 startAngle, glm::vec3 endAngle);

// Scaling functions.
void scaleUp();
void scaleDown();
```

- Rendering:

```
void renderMesh(GLuint programID);
```

- To control state of model:

```
// This function toggles the splatBool value,
// which controls whether we are showing splats
```

```
// or normal vertex.
void setBool();
```

- Binding Buffers:

```
// Variables to store Buffer id's
GLuint VBO, IBO, VSBO, VSIO, VSCO;
GLuint VAO, VCO, VNO, VSO;

// Bind the data from the mesh to an
// openGL buffer id, which will be
// used when passing to shaders.
void createVertexBuffer();
void createIndexBuffer();
void createColorBuffer();
void createNormalBuffer();
```

- Computation functions:

```
// Calculates splat for each triangle in the mesh
// and stores the values in the g_splat_buffer_data.
void computeSplat();

// This function calculates the normal for each vertex,
// by averaging normals of the triangle sharing that
// corresponding vertex.
void createNormals();

// Color of each vertex would be its normal calculated
// by above function.
void createNormalColors();


// Normalise mesh's coordinates so that entire mesh
// fits in a unit cube.
void normalise();
```

Calculating Splats:

- Since, each face is a triangle, first we calculate the inCentre and inRadius of the triangle.

- Since, openGl doesn't have a circle primitive, we need to construct one. We will construct this by keeping triangles side by side. (Same way a circular pizza is cut into triangle slices, we are doing opposite we are joining those triangle slices to get a circle)
- I chose to divide the circle into 20 triangles.
- Now, we need to first scale this circle, so that the circle's radius is same as the inradius of the face to which this circle belongs.
- Then we need to translate the circle, so that centre of the circle is same as inCircle of the face.
- Now, the circle initially will have its normal in the z direction, so we need to change it's orientation so that it will have normal same as face of the mesh.
- We cross the normal of face and z axis to get the axis about which we need to rotate to achieve above point, then calculate the angle using dot product. We now have axis, angle - So, we calcuate the quarternion using axis and angle, and perform the rotation on the circle as a whole i.e, we multiply the rotation matrix to all the vertices of the triangle which form the circle.
- We now have a circle/splat to a face whose normal is same as the face, and its radius is same as the circle which fits in the face/triangle.

2.1.2 Controller part:

Very little freedom was given to controller, in the sense that it has no logic/procedures related to mesh objects in it. Controller just takes what the keyboard input was and passed that message to the mesh objects, who would carry out the actual work.

It has very few functions too.

```
// So, controller mains a std::vector of mesh objects.
// A mesh object is created and added to this vector of mesh
// objects.
void addModel(char s[]);
```

```
// To get trackball type functionality.
glm::vec3 getCursorPos(GLFWwindow *window);

// To handle keyboard actions.
void handleKeys(GLFWwindow *window, int key, int code, int action, int mode);

// This binds buffers to Vertex array, element objects in the mesh.
// i.e vertex data is bound to VAO to be passed accordingly in shaders etc.
void createAndBind();

// This function would be called after createAndBind() function, in this
// we draw all the mesh objects on the screen. This works by calling renderMesh
// on each mesh object.
void renderModels();
```

We draw light from controller itself, though how light affects the mesh object is handled in shaders, the display of light is controlled from controller itself, we actually draw all models first, then light, since the refresh rate is very fast, we can't differentiate between these two. We only need light position, and we pass this to shaders, and we draw this element in the controller.

Keyboard actions for moving the light are also supported.

2.1.3 View part:

I felt there is no need to create another class for it, as the view here only concerns with background color and perspective/orthogonal projection. So, i included those in the main.cpp itself.

2.1.4 main.cpp

The compiling and loading of shaders is done in the main.cpp itself.

First we create a Controller object to work with.

Since, main.cpp is also like the view, we add objects to it from here itself.

```
c.addModel("plyfiles/car.ply");
c.addModel("plyfiles/beethoven.ply");
```

```
c.addModel("plyfiles/ant.ply");
```

We then call controller's *createAndBind*, which binds array, element buffers with corresponding objects.

Then we start a while loop, and in while loop we call the controller to renderModels, which inturn calls renderMesh on each mesh, thus rendering it on screen.

In this while loop, we also create functionality to send any keyboard input to the controller.

Lightning:

- From controller, we pass light Position to vertex shader, which passes it again to Fragment shader.
- We calculate the ambient component, by multiplying a white light color with ambient strength.

```
float ambientStrength = 0.1;
vec4 ambient = ambientStrength * vec4(lightColor, 1.0);
```

- For lightning a point, we calculate the light direction, using that we calculate diffuse component.

```
vec3 lightDir = normalize(lightPos - vPos);

float diff = max(dot(norm, lightDir), 0.0);
vec4 diffuse = diff * vec4(lightColor, 1.0f);
```

- We combine ambient and diffuse component and apply it on the actual color, we then get color of vertex as though it is illuminated by a light source

```
FragColor = (ambient + diffuse) * vCol;
```

3 Report Questions:

3.0.1 Question 1:

We can use Instance Rendering for this purpose. An instance means a single occurence of the model that we want to render, instanced rendering means

we can render multiple instances with a single draw call, we can also provide each instance with unique attributes.

So, using instanced rendering you only have to specify the geometry of an object once in the GPU, then we can pass additional buffer of matrices, one for each instance of object we are rendering.

We generally combine this with glVertexAttribDivisor, which controls the rate at which vertex attribute is updated.

3.0.2 Question 2:

Mostly with the help of shadows. If light from point source is in left direction, if we move towards left, the objects at the right should get dimmer and dimmer and if light reaches left end, everything becomes dark. If this is how it happens, we can say that lightning is working properly.

3.0.3 Question 3:

So, we need to draw splats for each vertex instead of face. Since, the main thing is these splats shouldn't overlap.

Algorithm:

-> For each triangle, calculate length of its sides, divide each side by 2, call that x.

-> For each vertex, check what sides it is part of, chose the smallest x among x's of all sides. This will be the radius of circle/splat around this vertex.

-> The normal of splat around a vertex will be same as that of the vertex's normal, which was calculated by averaging the normal's of the faces around it.

4 References & Help:

4.0.1 Help:

For some bugs, i took the help of Lalith kota(IMT2016132) , he really helped and we debugged the code together many times when facing a problem. Including how to approach solving splat problem in general.

4.0.2 References:

- Instanced Rendering: `http://ogldev.atspace.co.uk/www/tutorial33/tutorial33.html`
- Lightning: `https://learnopengl.com/Lighting/Basic-Lighting`
- Lightning2: `https://learnopengl.com/PBR/Lighting`
- Lightning3: `http://www.opengl-tutorial.org/beginners-tutorials/tutorial-8-basic-shading/`

5 Conclusions:

I learnt how we render a 3D model in computer graphics, and how lightning works. I was also able to appreciate the importance of dividing the code into Model, Controller and view part, and how it makes easy to add features.