

3D graphics and animation

Final Project

Issa Haddad

H00278537

Nov 22, 2017

Table of Contents

[Design and Scene-graph 3](#_Toc499090398)

[Model and instancing 4](#_Toc499090399)

[View, objects and camera 5](#_Toc499090400)

[Objects 5](#_Toc499090401)

[View and Camera 6](#_Toc499090402)

[Materials, animation and interaction 7](#_Toc499090403)

[Materials 7](#_Toc499090404)

[Light Model 7](#_Toc499090405)

[How scene is animated with user interaction 8](#_Toc499090406)

[Screen effects 9](#_Toc499090407)

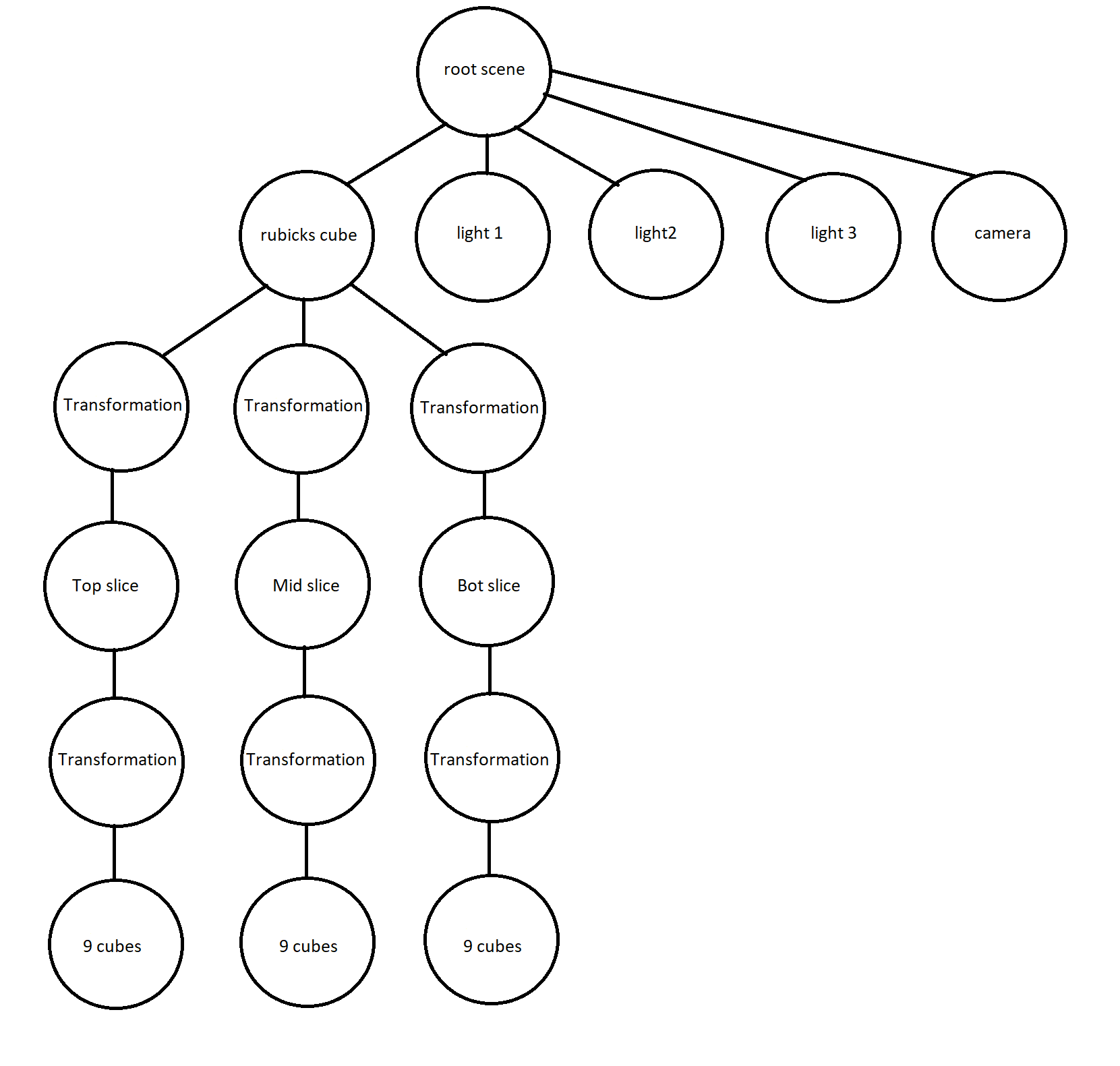
[Future 10](#_Toc499090408)

[Appendix 11](#_Toc499090409)

[Screenshots 11](#_Toc499090410)

[Source Code and shader code 14](#_Toc499090411)

## Design and Scene-graph



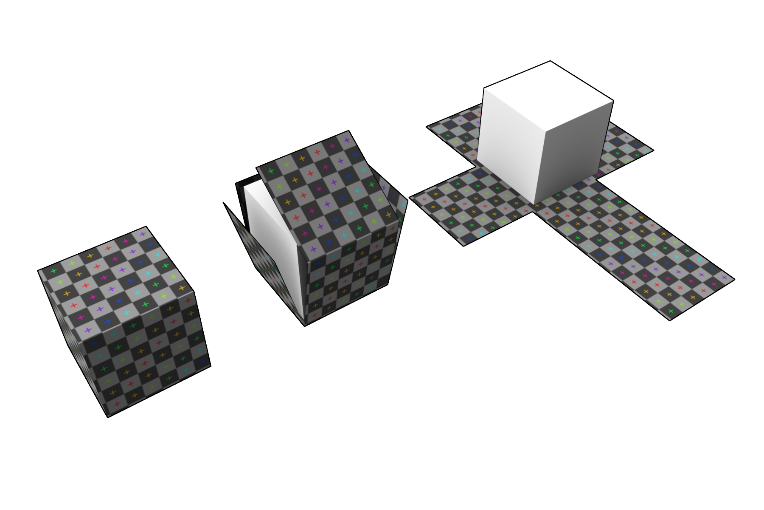
This scene graph consists of three main concepts. The first concept is the camera, which is implemented as an FPS look around camera. This camera is not bounded to move in one direction only, instead it can freely move around its axis exactly like a camera in a first person shooter game would behave. The second concept is the lighting. The lighting is not restricted to only one, instead, many lights can be inserted to give a more realistic look to my scene. The last and most important concept is the model, which is a 3d representation of an object called Rubik’s cube.

A Rubik’s cube is generally consisted of 27 single cubes formed to shape just 1 giant cube. This cube is made of 6 faces, with each face brushed with a different color. When transformed, each face of the Rubik’s cube has to be rotated around the origin of the entire cube. For the sake of simplicity in the scene graph, we can imagine that the Rubik’s cube is divided into three sections, the upper slice, the middle slice, and the bottom slice. A slice is a group of 9 cubes placed directly next to each other to form 3x3 grid. Because each slice is slightly below the other, we can say a transformation has taken place on them. A transformation is also required to place the 9 cubes in a 3x3 grid like structure.

## Model and instancing

As discussed earlier, the model used in my system is a Rubik’s cube. To start implementing the Rubik’s cube, I used a tool called Blender. Blender is a professional open source software that allows you to artistically create 3D models for various implementations in the computer graphics industry.

As you may already know, A Rubik’s cube is composed of 27 distinct smaller cubes. I used Blender to create and model exactly one of those distinct 3D cube shapes. But these cubes alone are not physically capable on their own of representing the Rubik’s cube along with all of its different colors. I needed to apply a certain texture to these cubes in order to give them the look and feel of an original Rubik’s cube. Using Blender UV-unwrapping technique, we can split open the 3D model, export it as PNG, and begin drawing the various colors and shadings necessary to visually represent it. please see graph below how we can use UV unwrapping to split open all of the cube faces in order to draw and color on them. Each of the 27 small cubes that make up the Rubik’s cube is physically distinct in its colors. For that, we need to create and apply different textures to different 3D cube models.



After our textures are ready, we need to convert their format to a one suitable and understood by OpenGL. For that, we used a tool called PVRTex to transform all of our textures to KTX. This tool is not only useful to manipulate the format, instead it also compresses the textures in order to guarantee the least conceivable texture memory overhead at application run time.

Now our cube is ready and our textures are ready, but remember we need 27 cubes in order to put together our Rubik’s cube. Instancing can be used to render multiple copies of the same object model and place them in different locations on the screen. This is exactly what is required, we can create 27 different cubes without having to model each one separately. This allows for a better performance and a quicker way of drawing the same mesh multiple times on the screen. For each of the instanced cubes, we apply specific transformation in order to place them next to each other and under each other in a way that represents how the Rubik’s cube looks as a whole.

## View, objects and camera

### Objects

From the previous sections we come to know how our cube object was created, and how many textures can be created for that particular model. We also know that we have our KTX texture format files that our OpenGL application is going to read. The following steps take place in our code after all the files are ready and stored in the correct directory:

* Storing model vertices and binding their position to the VAO and specifying the format

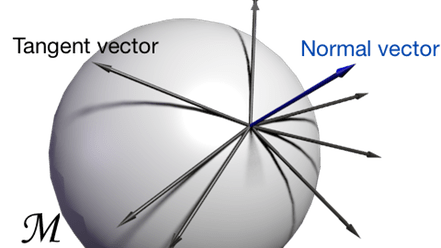
Our model is described by its vertices. The process of storing these vertices and their position is necessary to set up the objects for rendering with a specific shader program. This process includes creating a stream of vertices and allowing OpenGL to interpret that stream.

* Storing texture coordinates and binding their position to the VAO and specifying the format

These UV coordinates specify which part of the texture image will correspond to the vertex your triangles. After binding, filling, and configuring the textures, they need to be sent to the fragment shader which determines the color of each fragment before it is sent to the frame buffer. The fragment shader also requires a smapler2d in order to know which texture requires accessing.

* Storing the normal vectors and binding their position to the VAO and specifying the format

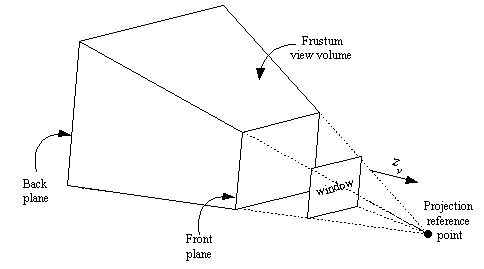
Because light bounces of the material of an object, it is important to grab the normal vectors of our model. The normal vectors are then used to calculate exactly the lighting effect, or how light bounces and affects each object in our space. An example of a normal vector with relevance to an object can be seen in the graph below.



### View and Camera

The view is how the world space is seen from the perspective of the eye. The view matrix is really defined by three variables. First your camera position, this specifies the exact position of your camera in the world space. The third variable or parameter is the camera target; this specifies which direction you want to look at. The third and final variable is the camera up. Usually the camera up is set as vec3(0,1,0) unless you want to see the world space flipped upside down, you would set it as vec3(0, -1,0).

For our project, we used the perspective projection. this type of projection simulates how we humans look at the real world from our eyes. The concept behind this projection is simple; objects further away appear smaller and objects closer to the camera appear larger. An example of the perspective projection can be viewed in the graph below.



The camera used in the project is called the FPS, or otherwise known as the look around camera. This camera is not confined to only moving around only one of its axis like the Steadicam camera, instead it can freely move around many of its axis. It is able to perform a pitch and a yaw around its axis, and It can also move forward, backwards, left, and right. If you ever played a first person shooting game, you would be able to visualize how this type of camera works in the world space it exists in.

Rotations and movements of the camera are controlled by both user keyboard and the user mouse. By grabbing the offset of the mouse and applying a level of sensitivity to it, we can control the pitch and the yaw. Furthermore, keyboard buttons can manipulate the camera position, and that manipulation will move the camera frontwards, backwards, left, and right. We can also apply another variable to determine the speed at which this camera moves in to these particular positions.

## Materials, animation and interaction

### Materials

As discussed earlier, we have created textures using Blender UV unwrapping and attached them to our model objects in OpenGL. Exporting the object from Blender, exports a material file with it. Materials are generally required for light to bounce of them. This is why different materials reflect light differently. For instance, a mirror would reflect light much more vividly than say a plastic. The material files exported from Blender specifies parameters on how light should be reflected on it. To be more specific the following exact parameters are specified in the materials file:

* Ka 🡪 Ambient color
* Kb 🡪 diffuse color
* Ks 🡪 specular color
* Ns 🡪 shininess
* Illum 🡪 illumination, where 1 = no specular highlights & 2 = specular highlights
* Ni 🡪 optical density or index of refraction
* D 🡪 transparency

### Light Model

The light model used in the project is simply an omnidirectional light source, otherwise known as a point light. This light model is a single source model that emits light in all directions. You can really think of this light model simply as just a light bulb. Another characteristic of this type of light is the fact that it possesses attenuation. With attenuation, the further away the object is from the light source, the darker it will appear. This is true for many cases in real world scenarios. Just imagine a candle in a dark room, the closer your hand to the candle, the brighter it appears, and the further away from the candle, the darker it appears. Point light is calculated using Phong’s light model, it calculated using the following formula:

🡪 represents the ambient lighting, which spreads equally all around with a fixed intensity and color. Where Ka is the reflection constant specified in the materials file.

🡪 represents the diffuse lighting, which is the reflection of light from a surface. Where Kd is the diffuse reflection constant specified in the materials files. represents the normal vector of the object, is the direction of vector from the surface of the object to the light source.

🡪 represents the specular reflection, which is the shiny bright color that is reflected from a surface. Where Ks is the specular reflection constant specified in the materials files. represents the direction that a perfectly reflected ray of light would take from the surface of the object. is the direction pointing towards the viewer.

Shininess 🡪 specular highlight, it is a constant also specified in the materials file.

### How scene is animated with user interaction

The scene animation can be divided into two categories: Camera Vs Slice Rotation

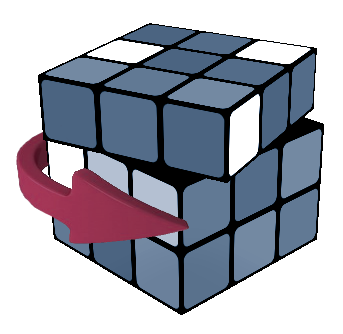
* Camera

Because we have implemented our project in such a way to use the FPS look around camera. We have to give the user the freedom of manipulating the camera position and rotation via pressing on the keyboard and moving the mouse. That means that the user now can freely move the camera around the world space and look at the Rubik’s cube for all of its direction and faces.

This is an important feature, because ideally we would like to consolidate a real world scenario of a user trying to solve the Rubik’s cube. That would involve looking at all the faces of the cube in order to analyze and efficiently work to find a solution for the current state problem.

* Slice Rotation

In a nutshell, the Slice rotation involves turning any row or column in the Rubik’s cube 90 degrees or -90 degrees around the center of that row or column. This can be illustrated in the graph below.



To break down this animation, we must first understand that the sequence of transformations in OpenGL is achieved in reverse order. With that being said, we can apply another rotation first before translating the object instances in our application. This operation allows us to rotate the entire Rubik’s cube as a whole after translation of the instances has occurred. We can later select all the instances of our object that make up every column and row and then selectively modify the rotation of those instances. This allows us to apply a rotation on a bunch of objects so that they can move as one instead of rotating independently.

## Screen effects

In the context of 3D graphics animations and gaming, screen effect is a process of performing additional operations on an already rendered result. This operation is sometimes also referred to as post-processing. This process has been vastly adopted by many 3D related industries, especially modern video games, to enhance the quality and create special scenery effects on their animations.

To perform post-processing, we must first use framebuffer objects which allows us to modify and manipulate referenced images from our model textures. Unlike the default framebuffer which is created with the context of OpenGL and allows for modification of the pixel format only, the framebuffer objects are user defined and allows use to render scenes immediately to a texture. After setting up the framebuffer object along with binding the texture and the necessary buffers, it is common use the fragment shader to actually implement the post-processing effect with the originally rendered scene as an input acting as a texture.

For my particular scene which is the Rubik’s cube, options for implementing the screen effect are not that numerous. One reason for the shortage in options is the fact that I can’t manipulate the color of the scene. The Rubik’s cube has to maintain the different colors of its faces, otherwise it is no longer called a Rubik’s cube. For that reason, I implemented a simple Blur effect on the overall scene. Blurring is done to reduce the level of detail in the scene. It is established by calculating the average color of a pixel and its surrounding pixels.

The concept behind adopting the blurry scene effect, it to simulate or to try and create an additional layer of difficulty to solving the Rubik’s cube. Now the user has to focus more, which creates a tougher challenge than the normal clear Rubik’s cube. Screenshots of the Rubik’s cube before and after implementing the blur screen effect can be found in the appendix.

Note: I was not successful in removing the transparency of the model created with the screen effect. I tried multiple methods including; manipulating the alpha color in “glClearColor”, disabling and removing Gl\_Blend, implementing “glBlendSpeparateFunc” for separate alpha values, using “glColorMask” and setting the last color to false.

## Future

There are plenty of ways to improve the quality of our demo scene using future technologies. With the Rubik’s cube, we are perhaps less interested in increasing performance or graphics quality and more interested in replacing the actual Rubik’s cube with a virtual one. This leaves us with really few options in hand. In the following paragraph I will be exploring those options and how they can be used to improve scene.

On the one hand, we can use Virtual reality to actually simulate a real Rubik’s cube in a virtual world. While this may be very effective to replace the actual Rubik’s cube which can cost money and take space in our real world, one problem with VR still exists. That problem is the fact that VR disconnects you from the real world. Unless you want to completely isolate yourself from what going on around you and focus entirely on solving the Rubik’s cube such as in a completion, no really would not want to take that approach.

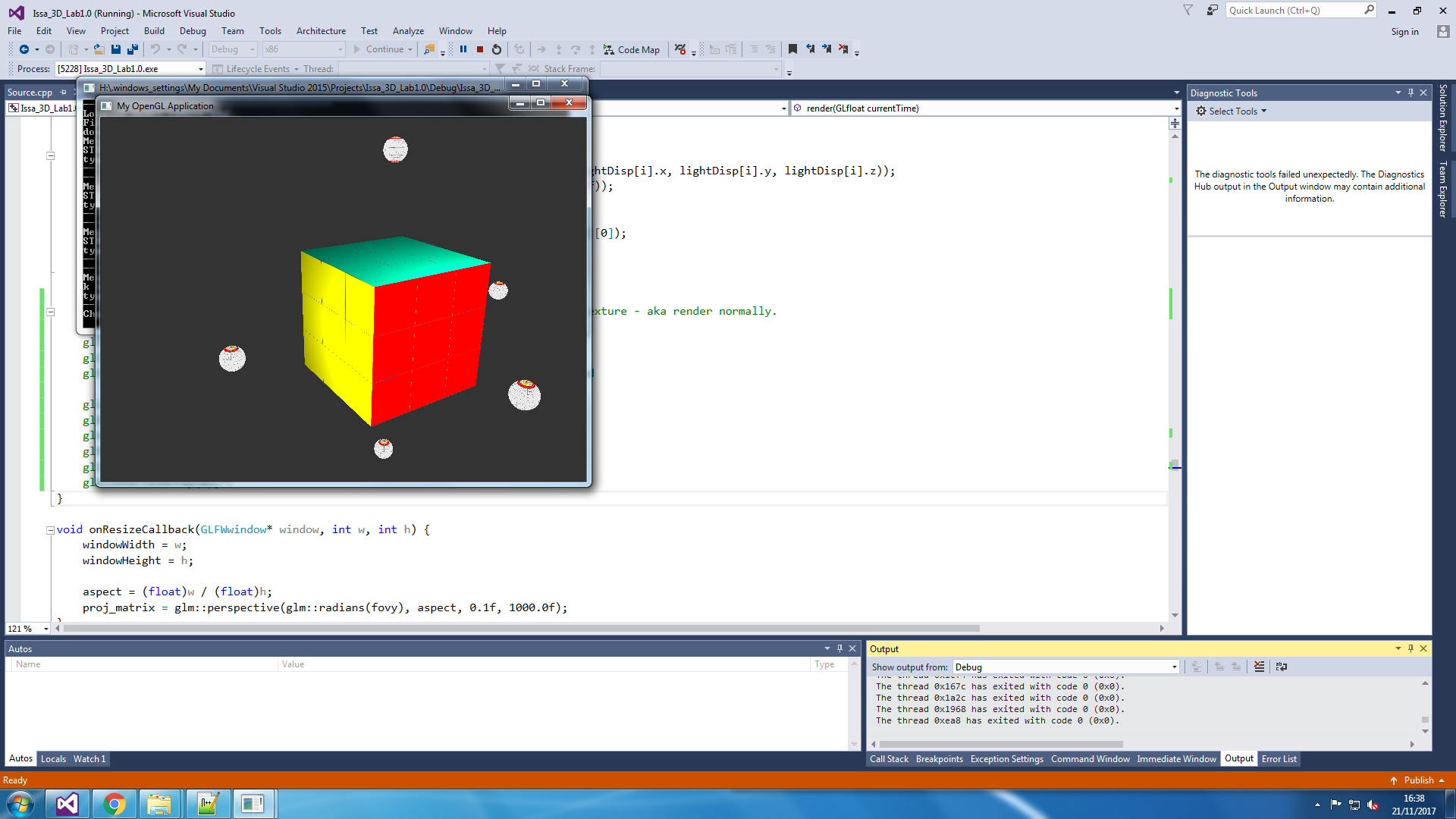
One the other hand, we can use Augmented reality, which is the process of adding a virtual element to an image of the real word. While this may be useful in keeping you connected with the real world, another issue still exists with AR. That issue is that our virtual element, or the Rubik’s will always appear on top of the real world. With that being said, the Rubik’s cube will always surface, and that usually leads it to block vision of some part of the world which isn’t always desirable.

One solution to both of the problems mentioned above is to use a technology called Mixed Reality. Mixed Reality is a slight modification to augmented reality, in the sense that it has one more additional feature. The extra feature is the fact that the virtual object not only interacts with the user, but also interacts with the real world. With that being said, we can for instance place the virtual Rubik’s cube under the table, and it will always stay there, and would never surface on top unless you physically attempt to change its location. So you will never see the Rubik’s cube unless for example you bend down and look under the table, only then will you be able to see the virtual object.

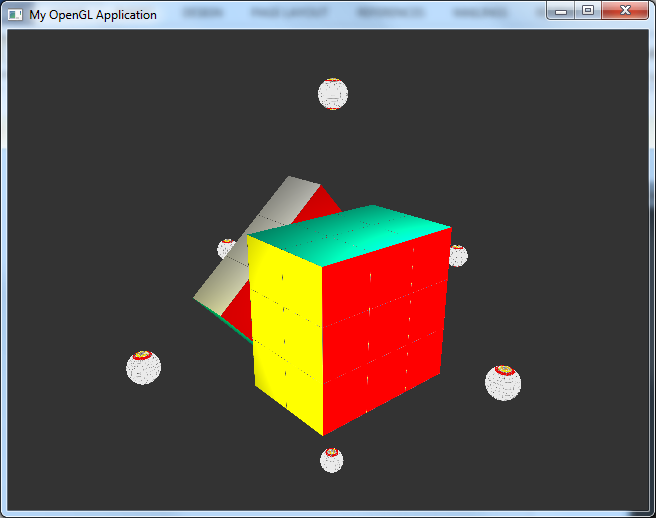
This solution will ultimately reduce space taken by the object, reduce cost because you no longer need to buy the actual cube, while keeping you connected with the real world at the same time. This leads to a win-win situation at hand.

## Appendix

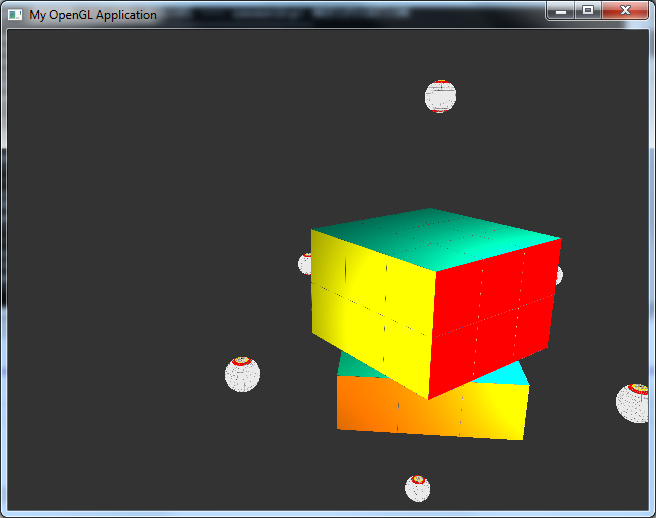
### Screenshots



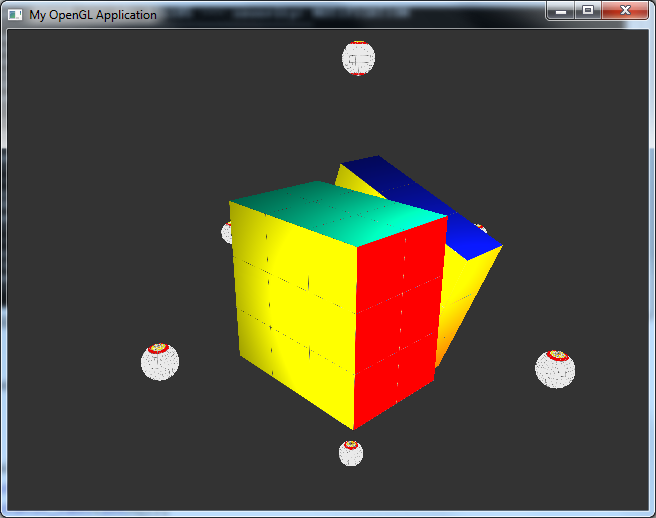
Rubik’s Cube without screen effect



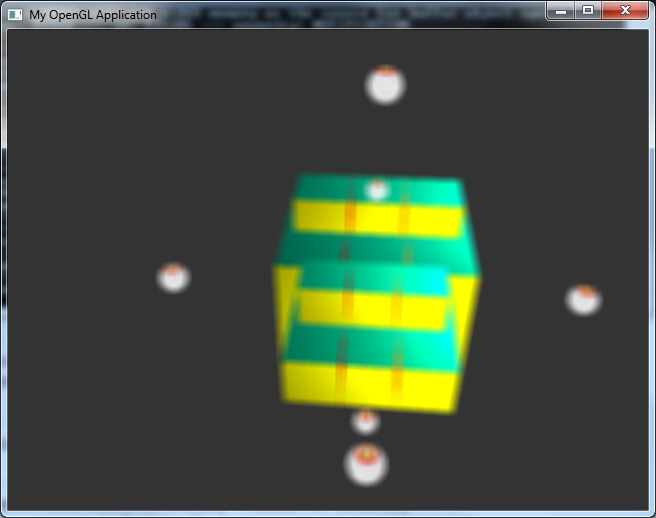
Rubik’s Cube without screen effect – Rotation = Pitch



Rubik’s Cube without screen effect – Rotation = Yaw



Rubik’s Cube without screen effect – Rotation = Roll



Rubik’s Cube with Blur screen effect

### Source Code and shader code

#### Source.cpp

1. // Multiple lights
2. //  see update function for keys

5. #pragma comment(linker, "/NODEFAULTLIB:MSVCRT")
7. #include <iostream>
8. #include <fstream>
9. #include <string>
10. #include <sstream>
11. #include <vector>
12. **using** **namespace** std;

15. #include <GL/glew.h>
16. #include <GLFW/glfw3.h>
17. #include <GLM/glm.hpp>
18. #include <GLM/gtx/transform.hpp>
19. #include <GLI/gli.hpp>
21. **void** errorCallbackGLFW(**int** error, **const** **char**\* description);
22. **void** hintsGLFW();
23. **void** endProgram();
24. **void** render(GLfloat currentTime);
25. **void** update(GLfloat currentTime);
26. **void** setupRender();
27. **void** startup();
28. **void** onResizeCallback(GLFWwindow\* window, **int** w, **int** h);
29. **void** onKeyCallback(GLFWwindow\* window, **int** key, **int** scancode, **int** action, **int** mods);
30. **void** onMouseButtonCallback(GLFWwindow\* window, **int** button, **int** action, **int** mods);
31. **void** onMouseMoveCallback(GLFWwindow\* window, **double** x, **double** y);
32. **void** onMouseWheelCallback(GLFWwindow\* window, **double** xoffset, **double** yoffset);
33. **void** debugGL();
34. **static** **void** APIENTRY openGLDebugCallback(GLenum source,
35. GLenum type,
36. GLuint id,
37. GLenum severity,
38. GLsizei length,
39. **const** GLchar\* message,
40. **const** GLvoid\* userParam);
41. string readShader(string name);
42. **void** checkErrorShader(GLuint shader);
43. **void** readObj(string name, **struct** modelObject \*obj);
44. **void** readTexture(string name, GLuint texture);

47. // VARIABLES
49. **float** modelAngle2[] = { 0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f };
50. **float** modelAngle3[] = { 0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f };
51. **float** modelAngle4[] = { 0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f };
53. **double** timer = 0;
54. **bool** rotate1[] = { **false**,**false**,**false**,**false**,**false**,**false**,**false**,**false**,**false** };
56. GLFWwindow\*     window;
57. **int**             windowWidth = 640;
58. **int**             windowHeight = 480;
59. //int               windowHeight = 480;
60. **bool**            running = **true**;
61. glm::mat4       proj\_matrix;
62. glm::vec3       modelAngle = glm::vec3(0.0f, 0.0f, 0.0f);
63. glm::vec3       modelDisp = glm::vec3(0.0f, 0.0f, 0.0f);
64. glm::vec3       cameraPosition = glm::vec3(0.0f, 0.0f, 5.0f);
65. glm::vec3       cameraFront = glm::vec3(0.0f, 0.0f, -1.0f);
66. glm::vec3       cameraUp = glm::vec3(0.0f, 1.0f, 0.0f);
67. **float**           aspect = (**float**)windowWidth / (**float**)windowHeight;
68. **float**           fovy = 45.0f;
69. **bool**            keyStatus[1024];
70. GLfloat         deltaTime = 0.0f;
71. GLfloat         lastTime = 0.0f;
73. // FPS camera variables
74. GLfloat         yaw = -90.0f;   // init pointing to inside
75. GLfloat         pitch = 0.0f;   // start centered
76. GLfloat         lastX = (GLfloat)windowWidth / 2.0f;    // start middle screen
77. GLfloat         lastY = (GLfloat)windowHeight / 2.0f;   // start middle screen
78. **bool**            firstMouse = **true**;
80. // OBJ Variables
81. **struct** modelObject {
82. std::vector < glm::vec3 > out\_vertices;
83. std::vector < glm::vec2 > out\_uvs;
84. std::vector < glm::vec3 > out\_normals;
85. GLuint\*     texture;
86. GLuint      program;
87. GLuint      vao;
88. GLuint      buffer[2];
89. GLint       mv\_location;
90. GLint       proj\_location;
91. GLint       tex\_location;
93. // extra variables for this example
94. GLuint      matColor\_location;
95. GLuint      lightColor\_location;
97. } objectModel, lightModel;
99. glm::vec3       \*modelPositions;
100. glm::vec3       \*modelRotations;
102. // Light
103. **bool**            movingLight = **true**;
104. **int**             movingLightNumber = 0;
105. glm::vec3       \*lightDisp;
106. //glm::vec3     lightDisp = glm::vec3(-1.0f, -1.0f, 0.0f);
107. glm::vec3       ia = glm::vec3(1.0f, 1.0f, 1.0f);
108. GLfloat         ka = 5.0;
109. //glm::vec3     id = glm::vec3(0.93f, 0.75f, 0.32f);
110. glm::vec3\*      id;
111. GLfloat         kd = 1.0;
112. glm::vec3       is = glm::vec3(1.0f, 1.0f, 1.0f);
113. GLfloat         ks = 1.0;

116. // framebuffer
117. GLuint          framebuffer;
118. GLuint          framebufferTexture;
119. GLuint          depthbuffer;
120. GLuint          displayVao;
121. GLuint          displayBuffer[2];
122. std::vector < glm::vec2 > displayVertices;
123. std::vector < glm::vec2 > displayUvs;
124. GLuint          displayProgram;

127. **int** main()
128. {
129. **if** (!glfwInit()) {                          // Checking for GLFW
130. cout << "Could not initialise GLFW...";
131. **return** 0;
132. }
134. glfwSetErrorCallback(errorCallbackGLFW);    // Setup a function to catch and display all GLFW errors.
136. hintsGLFW();                                // Setup glfw with various hints.
138. // Start a window using GLFW
139. string title = "My OpenGL Application";
141. // Fullscreen
142. //const GLFWvidmode \* mode = glfwGetVideoMode(glfwGetPrimaryMonitor());
143. //windowWidth = mode->width; windowHeight = mode->height;
144. //window = glfwCreateWindow(windowWidth, windowHeight, title.c\_str(), glfwGetPrimaryMonitor(), NULL);
146. // Window
147. window = glfwCreateWindow(windowWidth, windowHeight, title.c\_str(), NULL, NULL);
148. **if** (!window) {                              // Window or OpenGL context creation failed
149. cout << "Could not initialise GLFW...";
150. endProgram();
151. **return** 0;
152. }
154. glfwMakeContextCurrent(window);             // making the OpenGL context current
156. // Start GLEW (note: always initialise GLEW after creating your window context.)
157. glewExperimental = GL\_TRUE;                 // hack: catching them all - forcing newest debug callback (glDebugMessageCallback)
158. GLenum errGLEW = glewInit();
159. **if** (GLEW\_OK != errGLEW) {                   // Problems starting GLEW?
160. cout << "Could not initialise GLEW...";
161. endProgram();
162. **return** 0;
163. }
165. debugGL();                                  // Setup callback to catch openGL errors.
167. // Setup all the message loop callbacks.
168. glfwSetWindowSizeCallback(window, onResizeCallback);        // Set callback for resize
169. glfwSetKeyCallback(window, onKeyCallback);                  // Set Callback for keys
170. glfwSetMouseButtonCallback(window, onMouseButtonCallback);  // Set callback for mouse click
171. glfwSetCursorPosCallback(window, onMouseMoveCallback);      // Set callback for mouse move
172. glfwSetScrollCallback(window, onMouseWheelCallback);        // Set callback for mouse wheel.
173. //glfwSetInputMode(window, GLFW\_CURSOR, GLFW\_CURSOR\_NORMAL);    // Set mouse cursor. Fullscreen
174. glfwSetInputMode(window, GLFW\_CURSOR, GLFW\_CURSOR\_DISABLED);    // Set mouse cursor FPS.
176. setupRender();                              // setup some render variables.
177. startup();                                  // Setup all necessary information for startup (aka. load texture, shaders, models, etc).
179. **do** {                                        // run until the window is closed
180. GLfloat currentTime = (GLfloat)glfwGetTime();       // retrieve timelapse
181. deltaTime = currentTime - lastTime;     // Calculate delta time
182. lastTime = currentTime;                 // Save for next frame calculations.
183. glfwPollEvents();                       // poll callbacks
184. update(currentTime);                    // update (physics, animation, structures, etc)
185. render(currentTime);                    // call render function.
187. glfwSwapBuffers(window);                // swap buffers (avoid flickering and tearing)
189. running &= (glfwGetKey(window, GLFW\_KEY\_ESCAPE) == GLFW\_RELEASE);   // exit if escape key pressed
190. running &= (glfwWindowShouldClose(window) != GL\_TRUE);
191. } **while** (running);
193. endProgram();           // Close and clean everything up...
195. cout << "\nPress any key to continue...\n";
196. cin.ignore(); cin.get(); // delay closing console to read debugging errors.
198. **return** 0;
199. }
201. **void** errorCallbackGLFW(**int** error, **const** **char**\* description) {
202. cout << "Error GLFW: " << description << "\n";
203. }
205. **void** hintsGLFW() {
206. glfwWindowHint(GLFW\_OPENGL\_DEBUG\_CONTEXT, GL\_TRUE);         // Create context in debug mode - for debug message callback
207. glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MAJOR, 4);
208. glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR, 5);
209. }
211. **void** endProgram() {
212. glfwMakeContextCurrent(window);     // destroys window handler
213. glfwTerminate();    // destroys all windows and releases resources.
215. // tidy heap memory
216. **delete**[] objectModel.texture;
217. **delete**[] lightModel.texture;
218. **delete**[] modelPositions;
219. **delete**[] modelRotations;
220. **delete**[] lightDisp;
221. **delete**[] id;
222. }
224. **void** setupRender() {
225. glfwSwapInterval(1);    // Ony render when synced (V SYNC)
227. glfwWindowHint(GLFW\_OPENGL\_PROFILE, GLFW\_OPENGL\_CORE\_PROFILE);
228. glfwWindowHint(GLFW\_OPENGL\_FORWARD\_COMPAT, GL\_TRUE);
229. glfwWindowHint(GLFW\_SAMPLES, 0);
230. glfwWindowHint(GLFW\_STEREO, GL\_FALSE);
231. }
233. **void** startup() {
235. // Load main object model and shaders
237. // --------------Main Model---------------------
238. objectModel.program = glCreateProgram();
240. string vs\_text = readShader("vs\_model.glsl"); **static** **const** **char**\* vs\_source = vs\_text.c\_str();
241. GLuint vs = glCreateShader(GL\_VERTEX\_SHADER);
242. glShaderSource(vs, 1, &vs\_source, NULL);
243. glCompileShader(vs);
244. checkErrorShader(vs);
245. glAttachShader(objectModel.program, vs);
247. string fs\_text = readShader("fs\_model.glsl"); **static** **const** **char**\* fs\_source = fs\_text.c\_str();
248. GLuint fs = glCreateShader(GL\_FRAGMENT\_SHADER);
249. glShaderSource(fs, 1, &fs\_source, NULL);
250. glCompileShader(fs);
251. checkErrorShader(fs);
252. glAttachShader(objectModel.program, fs);
254. glLinkProgram(objectModel.program);
255. glUseProgram(objectModel.program);
257. readObj("RubiksObj.obj", &objectModel);
259. glCreateBuffers(3, objectModel.buffer);     // Create a new buffer
261. // Store the vertices
262. glNamedBufferStorage(objectModel.buffer[0], objectModel.out\_vertices.size() \* **sizeof**(glm::vec3), &objectModel.out\_vertices[0], GL\_DYNAMIC\_STORAGE\_BIT);
263. glBindBuffer(GL\_ARRAY\_BUFFER, objectModel.buffer[0]);   // Bind Buffer
265. // Store the texture coordinates
266. glNamedBufferStorage(objectModel.buffer[1], objectModel.out\_uvs.size() \* **sizeof**(glm::vec2), &objectModel.out\_uvs[0], GL\_DYNAMIC\_STORAGE\_BIT);
267. glBindBuffer(GL\_ARRAY\_BUFFER, objectModel.buffer[1]);   // Bind Buffer
269. // Store the normal Vectors
270. glNamedBufferStorage(objectModel.buffer[2], objectModel.out\_normals.size() \* **sizeof**(glm::vec3), &objectModel.out\_normals[0], GL\_DYNAMIC\_STORAGE\_BIT);
271. glBindBuffer(GL\_ARRAY\_BUFFER, objectModel.buffer[3]);   // Bind Buffer
273. glCreateVertexArrays(1, &objectModel.vao);      // Create Vertex Array Object
275. // Bind vertex position buffer to the vao and format
276. glVertexArrayVertexBuffer(objectModel.vao, 0, objectModel.buffer[0], 0, **sizeof**(GLfloat) \* 3);
277. glVertexArrayAttribFormat(objectModel.vao, 0, 3, GL\_FLOAT, GL\_FALSE, 0);
278. glEnableVertexArrayAttrib(objectModel.vao, 0);  // Enable Vertex Array Attribute
280. // Bind texture coordinate buffer to the vao and format
281. glVertexArrayVertexBuffer(objectModel.vao, 1, objectModel.buffer[1], 0, **sizeof**(GLfloat) \* 2);
282. glVertexArrayAttribFormat(objectModel.vao, 1, 2, GL\_FLOAT, GL\_FALSE, 0);
283. glEnableVertexArrayAttrib(objectModel.vao, 1);  // Enable Vertex Array Attribute
285. // Bind the normals buffer to the vao and format
286. glVertexArrayVertexBuffer(objectModel.vao, 2, objectModel.buffer[2], 0, **sizeof**(GLfloat) \* 3);
287. glVertexArrayAttribFormat(objectModel.vao, 2, 3, GL\_FLOAT, GL\_FALSE, 0);
288. glEnableVertexArrayAttrib(objectModel.vao, 2);  // Enable Vertex Array Attribute
290. glBindVertexArray(objectModel.vao);             // Bind VertexArrayObject
292. objectModel.mv\_location = glGetUniformLocation(objectModel.program, "mv\_matrix");
293. objectModel.proj\_location = glGetUniformLocation(objectModel.program, "proj\_matrix");
294. objectModel.tex\_location = glGetUniformLocation(objectModel.program, "tex");
295. objectModel.lightColor\_location = glGetUniformLocation(objectModel.program, "ia");
296. objectModel.lightColor\_location = glGetUniformLocation(objectModel.program, "ka");


300. //--------------Light Model--------------------------
301. lightModel.program = glCreateProgram();
303. string vs\_textLight = readShader("vs\_light.glsl"); **static** **const** **char**\* vs\_sourceLight = vs\_textLight.c\_str();
304. GLuint vsLight = glCreateShader(GL\_VERTEX\_SHADER);
305. glShaderSource(vsLight, 1, &vs\_sourceLight, NULL);
306. glCompileShader(vsLight);
307. checkErrorShader(vsLight);
308. glAttachShader(lightModel.program, vsLight);
310. string fs\_textLight = readShader("fs\_light.glsl"); **static** **const** **char**\* fs\_sourceLight = fs\_textLight.c\_str();
311. GLuint fsLight = glCreateShader(GL\_FRAGMENT\_SHADER);
312. glShaderSource(fsLight, 1, &fs\_sourceLight, NULL);
313. glCompileShader(fsLight);
314. checkErrorShader(fsLight);
315. glAttachShader(lightModel.program, fsLight);
317. glLinkProgram(lightModel.program);
319. readObj("sphere.obj", &lightModel);
321. glCreateBuffers(3, lightModel.buffer);      // Create a new buffer
323. // Store the vertices
324. glNamedBufferStorage(lightModel.buffer[0], lightModel.out\_vertices.size() \* **sizeof**(glm::vec3), &lightModel.out\_vertices[0], GL\_DYNAMIC\_STORAGE\_BIT);
325. glBindBuffer(GL\_ARRAY\_BUFFER, lightModel.buffer[0]);    // Bind Buffer
327. // Store the texture coordinates
328. glNamedBufferStorage(lightModel.buffer[1], lightModel.out\_uvs.size() \* **sizeof**(glm::vec2), &lightModel.out\_uvs[0], GL\_DYNAMIC\_STORAGE\_BIT);
329. glBindBuffer(GL\_ARRAY\_BUFFER, lightModel.buffer[1]);    // Bind Buffer
331. // Store the normal Vectors
332. glNamedBufferStorage(lightModel.buffer[2], lightModel.out\_normals.size() \* **sizeof**(glm::vec3), &lightModel.out\_normals[0], GL\_DYNAMIC\_STORAGE\_BIT);
333. glBindBuffer(GL\_ARRAY\_BUFFER, lightModel.buffer[3]);    // Bind Buffer
335. glCreateVertexArrays(1, &lightModel.vao);       // Create Vertex Array Object
337. // Bind vertex position buffer to the vao and format
338. glVertexArrayVertexBuffer(lightModel.vao, 0, lightModel.buffer[0], 0, **sizeof**(GLfloat) \* 3);
339. glVertexArrayAttribFormat(lightModel.vao, 0, 3, GL\_FLOAT, GL\_FALSE, 0);
340. glEnableVertexArrayAttrib(lightModel.vao, 0);   // Enable Vertex Array Attribute
342. // Bind texture coordinate buffer to the vao and format
343. glVertexArrayVertexBuffer(lightModel.vao, 1, lightModel.buffer[1], 0, **sizeof**(GLfloat) \* 2);
344. glVertexArrayAttribFormat(lightModel.vao, 1, 2, GL\_FLOAT, GL\_FALSE, 0);
345. glEnableVertexArrayAttrib(lightModel.vao, 1);   // Enable Vertex Array Attribute
347. // Bind the normals buffer to the vao and format
348. glVertexArrayVertexBuffer(lightModel.vao, 2, lightModel.buffer[2], 0, **sizeof**(GLfloat) \* 3);
349. glVertexArrayAttribFormat(lightModel.vao, 2, 3, GL\_FLOAT, GL\_FALSE, 0);
350. glEnableVertexArrayAttrib(lightModel.vao, 2);   // Enable Vertex Array Attribute
352. glBindVertexArray(lightModel.vao);              // Bind VertexArrayObject
354. lightModel.mv\_location = glGetUniformLocation(lightModel.program, "mv\_matrix");
355. lightModel.proj\_location = glGetUniformLocation(lightModel.program, "proj\_matrix");
356. lightModel.tex\_location = glGetUniformLocation(lightModel.program, "tex");

359. //--------------------------------------------
361. modelPositions = **new** glm::vec3[27];
362. modelPositions[0] = glm::vec3(0.0f, 0.0f, 0.0f);
363. modelPositions[1] = glm::vec3(0.4f, 0.0f, 0.0f);
364. modelPositions[2] = glm::vec3(-0.4f, 0.0f, 0.0f);
365. modelPositions[3] = glm::vec3(0.0f, 0.0f, 0.4f);
366. modelPositions[4] = glm::vec3(0.4f, 0.0f, 0.4f);
367. modelPositions[5] = glm::vec3(-0.4f, 0.0f, 0.4f);
368. modelPositions[6] = glm::vec3(0.0f, 0.0f, -0.4f);
369. modelPositions[7] = glm::vec3(0.4f, 0.0f, -0.4f);
370. modelPositions[8] = glm::vec3(-0.4f, 0.0f, -0.4f);
371. modelPositions[9] = glm::vec3(0.0f, 0.4f, 0.0f);
372. modelPositions[10] = glm::vec3(0.4f, 0.4f, 0.0f);
373. modelPositions[11] = glm::vec3(-0.4f, 0.4f, 0.0f);
374. modelPositions[12] = glm::vec3(0.0f, 0.4f, 0.4f);
375. modelPositions[13] = glm::vec3(0.4f, 0.4f, 0.4f);
376. modelPositions[14] = glm::vec3(-0.4f, 0.4f, 0.4f);
377. modelPositions[15] = glm::vec3(0.0f, 0.4f, -0.4f);
378. modelPositions[16] = glm::vec3(0.4f, 0.4f, -0.4f);
379. modelPositions[17] = glm::vec3(-0.4f, 0.4f, -0.4f);
380. modelPositions[18] = glm::vec3(0.0f, -0.4f, 0.0f);
381. modelPositions[19] = glm::vec3(0.4f, -0.4f, 0.0f);
382. modelPositions[20] = glm::vec3(-0.4f, -0.4f, 0.0f);
383. modelPositions[21] = glm::vec3(0.0f, -0.4f, 0.4f);
384. modelPositions[22] = glm::vec3(0.4f, -0.4f, 0.4f);
385. modelPositions[23] = glm::vec3(-0.4f, -0.4f, 0.4f);
386. modelPositions[24] = glm::vec3(0.0f, -0.4f, -0.4f);
387. modelPositions[25] = glm::vec3(0.4f, -0.4f, -0.4f);
388. modelPositions[26] = glm::vec3(-0.4f, -0.4f, -0.4f);
390. modelRotations = **new** glm::vec3[12];
391. modelRotations[0] = glm::vec3(0.0f, 0.0f, 0.0f);
392. modelRotations[1] = glm::vec3(20.0f, 10.0f, 0.0f);
393. modelRotations[2] = glm::vec3(30.0f, 40.0f, 0.0f);
394. modelRotations[3] = glm::vec3(20.0f, 50.0f, 0.0f);
395. modelRotations[4] = glm::vec3(40.0f, 50.0f, 0.0f);
396. modelRotations[5] = glm::vec3(70.0f, 60.0f, 0.0f);
397. modelRotations[6] = glm::vec3(80.0f, 60.0f, 0.0f);
398. modelRotations[7] = glm::vec3(10.0f, 20.0f, 0.0f);
399. modelRotations[8] = glm::vec3(70.0f, 30.0f, 0.0f);
400. modelRotations[9] = glm::vec3(50.0f, 40.0f, 0.0f);
401. modelRotations[10] = glm::vec3(60.0f, 80.0f, 0.0f);
402. modelRotations[11] = glm::vec3(70.0f, 80.0f, 0.0f);
404. lightDisp = **new** glm::vec3[6];
405. lightDisp[0] = glm::vec3(-1.5f, 0.0f, 0.0f);
406. lightDisp[1] = glm::vec3(0.0f, 0.0f, 1.5f);
407. lightDisp[2] = glm::vec3(1.5f, 0.0f, 0.0f);
408. lightDisp[3] = glm::vec3(0.0f, 0.0f, -1.5f);
409. lightDisp[4] = glm::vec3(0.0f, 1.5f, 0.0f);
410. lightDisp[5] = glm::vec3(0.0f, -1.5f, 0.0f);
412. id = **new** glm::vec3[4];
413. id[0] = glm::vec3(1.0f, 0.0f, 0.0f);
414. id[1] = glm::vec3(0.0f, 1.0f, 0.0f);
415. id[2] = glm::vec3(0.0f, 0.0f, 1.0f);
416. id[3] = glm::vec3(1.0f, 1.0f, 0.0f);

419. glFrontFace(GL\_CCW);
420. glCullFace(GL\_BACK);
421. glEnable(GL\_CULL\_FACE);
423. glEnable(GL\_DEPTH\_TEST);
424. glDepthFunc(GL\_LEQUAL);
426. // Calculate proj\_matrix for the first time.
427. aspect = (**float**)windowWidth / (**float**)windowHeight;
428. proj\_matrix = glm::perspective(glm::radians(fovy), aspect, 0.1f, 1000.0f);

431. // ----------Start Framebuffer---------------
432. glGenFramebuffers(1, &framebuffer);
433. glBindFramebuffer(GL\_FRAMEBUFFER, framebuffer);
435. // Create a texture for the framebuffer
436. glGenTextures(1, &framebufferTexture);
438. // Bind this texture so we can modify it.
439. glBindTexture(GL\_TEXTURE\_2D, framebufferTexture);
441. //  Start the texture and give it a size but no data- of course if you resize you need to change your texture.
442. glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGB, windowWidth, windowHeight, 0, GL\_RGB, GL\_UNSIGNED\_BYTE, 0);
444. // filtering needed - future lecture
445. glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);
446. glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);
448. // Depth buffer texture - Need to attach depth too otherwise depth testing will not be performed.
449. glGenRenderbuffers(1, &depthbuffer);
450. glBindRenderbuffer(GL\_RENDERBUFFER, depthbuffer);
451. glRenderbufferStorage(GL\_RENDERBUFFER, GL\_DEPTH\_COMPONENT, windowWidth, windowHeight);
452. glFramebufferRenderbuffer(GL\_FRAMEBUFFER, GL\_DEPTH\_ATTACHMENT, GL\_RENDERBUFFER, depthbuffer);

455. // Create a quad to display our framebuffer
456. displayVertices.push\_back(glm::vec2(-1.0f, 1.0f));
457. displayVertices.push\_back(glm::vec2(-1.0f, -1.0f));
458. displayVertices.push\_back(glm::vec2(1.0f, -1.0f));
459. displayVertices.push\_back(glm::vec2(-1.0f, 1.0f));
460. displayVertices.push\_back(glm::vec2(1.0f, -1.0f));
461. displayVertices.push\_back(glm::vec2(1.0f, 1.0f));
463. displayUvs.push\_back(glm::vec2(0.0f, 1.0f));
464. displayUvs.push\_back(glm::vec2(0.0f, 0.0f));
465. displayUvs.push\_back(glm::vec2(1.0f, 0.0f));
466. displayUvs.push\_back(glm::vec2(0.0f, 1.0f));
467. displayUvs.push\_back(glm::vec2(1.0f, 0.0f));
468. displayUvs.push\_back(glm::vec2(1.0f, 1.0f));
470. glCreateBuffers(2, displayBuffer);      // Create a new buffer
471. // Store the vertices
472. glNamedBufferStorage(displayBuffer[0], displayVertices.size() \* **sizeof**(glm::vec2), &displayVertices[0], GL\_DYNAMIC\_STORAGE\_BIT);
473. glBindBuffer(GL\_ARRAY\_BUFFER, displayBuffer[0]);    // Bind Buffer
474. // Store the texture coordinates
475. glNamedBufferStorage(displayBuffer[1], displayUvs.size() \* **sizeof**(glm::vec2), &displayUvs[0], GL\_DYNAMIC\_STORAGE\_BIT);
476. glBindBuffer(GL\_ARRAY\_BUFFER, displayBuffer[1]);    // Bind Buffer
478. glCreateVertexArrays(1, &displayVao);       // Create Vertex Array Object
479. // Bind vertex position buffer to the vao and format
480. glVertexArrayVertexBuffer(displayVao, 0, displayBuffer[0], 0, **sizeof**(GLfloat) \* 2);
481. glVertexArrayAttribFormat(displayVao, 0, 2, GL\_FLOAT, GL\_FALSE, 0);
482. glEnableVertexArrayAttrib(displayVao, 0);   // Enable Vertex Array Attribute
484. // Bind texture coordinate buffer to the vao and format
485. glVertexArrayVertexBuffer(displayVao, 1, displayBuffer[1], 0, **sizeof**(GLfloat) \* 2);
486. glVertexArrayAttribFormat(displayVao, 1, 2, GL\_FLOAT, GL\_FALSE, 0);
487. glEnableVertexArrayAttrib(displayVao, 1);   // Enable Vertex Array Attribute
489. glBindVertexArray(displayVao);              // Bind VertexArrayObject
491. //load shaders
492. displayProgram = glCreateProgram();
494. string dvs\_text = readShader("vs\_display.glsl"); **static** **const** **char**\* dvs\_source = dvs\_text.c\_str();
495. GLuint dvs = glCreateShader(GL\_VERTEX\_SHADER);
496. glShaderSource(dvs, 1, &dvs\_source, NULL);
497. glCompileShader(dvs);
498. checkErrorShader(dvs);
499. glAttachShader(displayProgram, dvs);
501. string dfs\_text = readShader("fs\_display.glsl"); **static** **const** **char**\* dfs\_source = dfs\_text.c\_str();
502. GLuint dfs = glCreateShader(GL\_FRAGMENT\_SHADER);
503. glShaderSource(dfs, 1, &dfs\_source, NULL);
504. glCompileShader(dfs);
505. checkErrorShader(dfs);
506. glAttachShader(displayProgram, dfs);
508. glLinkProgram(displayProgram);
509. glUseProgram(displayProgram);
510. }
512. **void** update(GLfloat currentTime) {
514. // calculate movement
515. GLfloat cameraSpeed = 1.0f \* deltaTime;
516. **if** (keyStatus[GLFW\_KEY\_W]) cameraPosition += cameraSpeed \* cameraFront;
517. **if** (keyStatus[GLFW\_KEY\_S]) cameraPosition -= cameraSpeed \* cameraFront;
518. **if** (keyStatus[GLFW\_KEY\_A]) cameraPosition -= glm::normalize(glm::cross(cameraFront, cameraUp)) \* cameraSpeed;
519. **if** (keyStatus[GLFW\_KEY\_D]) cameraPosition += glm::normalize(glm::cross(cameraFront, cameraUp)) \* cameraSpeed;
521. **if** (keyStatus[GLFW\_KEY\_L] && (movingLight == **false**)) {
522. cout << "Change mode to moving light...\n";
523. movingLight = **true**;
524. }
525. **if** (keyStatus[GLFW\_KEY\_M] && (movingLight == **true**)) {
526. cout << "Change mode to moving object...\n";
527. movingLight = **false**;
528. }
530. **if** (keyStatus[GLFW\_KEY\_0] && (movingLight == **true**)) {
531. movingLightNumber = 0;  cout << "Moving light 0...\n";
532. }
533. **if** (keyStatus[GLFW\_KEY\_1] && (movingLight == **true**)) {
534. movingLightNumber = 1;  cout << "Moving light 1...\n";
535. }
536. **if** (keyStatus[GLFW\_KEY\_2] && (movingLight == **true**)) {
537. movingLightNumber = 2;  cout << "Moving light 2...\n";
538. }
539. **if** (keyStatus[GLFW\_KEY\_3] && (movingLight == **true**)) {
540. movingLightNumber = 3;  cout << "Moving light 3...\n";
541. }
543. **if** (keyStatus[GLFW\_KEY\_Y])
544. rotate1[0] = **true**;
545. **if** (keyStatus[GLFW\_KEY\_U])
546. rotate1[1] = **true**;
547. **if** (keyStatus[GLFW\_KEY\_I])
548. rotate1[2] = **true**;
549. **if** (keyStatus[GLFW\_KEY\_H])
550. rotate1[3] = **true**;
551. **if** (keyStatus[GLFW\_KEY\_J])
552. rotate1[4] = **true**;
553. **if** (keyStatus[GLFW\_KEY\_K])
554. rotate1[5] = **true**;
555. **if** (keyStatus[GLFW\_KEY\_B])
556. rotate1[6] = **true**;
557. **if** (keyStatus[GLFW\_KEY\_N])
558. rotate1[7] = **true**;
559. **if** (keyStatus[GLFW\_KEY\_M])
560. rotate1[8] = **true**;
562. **if** (rotate1[0] == **true**)
563. {
564. modelAngle2[1] += 0.1;
565. modelAngle2[4] += 0.1;
566. modelAngle2[7] += 0.1;
567. modelAngle2[10] += 0.1;
568. modelAngle2[13] += 0.1;
569. modelAngle2[16] += 0.1;
570. modelAngle2[19] += 0.1;
571. modelAngle2[22] += 0.1;
572. modelAngle2[25] += 0.1;
573. timer++;
574. **if** (timer >= 63 && rotate1[0] == **true**)
575. {
576. rotate1[0] = **false**;
577. timer = 0;
578. modelAngle2[1] = 0.0;
579. modelAngle2[4] = 0.0;
580. modelAngle2[7] = 0.0;
581. modelAngle2[10] = 0.0;
582. modelAngle2[13] = 0.0;
583. modelAngle2[16] = 0.0;
584. modelAngle2[19] = 0.0;
585. modelAngle2[22] = 0.0;
586. modelAngle2[25] = 0.0;
587. }
588. }
590. **if** (rotate1[1] == **true**)
591. {
592. modelAngle2[0] += 0.1;
593. modelAngle2[3] += 0.1;
594. modelAngle2[6] += 0.1;
595. modelAngle2[9] += 0.1;
596. modelAngle2[12] += 0.1;
597. modelAngle2[15] += 0.1;
598. modelAngle2[18] += 0.1;
599. modelAngle2[21] += 0.1;
600. modelAngle2[24] += 0.1;
601. timer++;
602. **if** (timer >= 63 && rotate1[1] == **true**)
603. {
604. rotate1[1] = **false**;
605. timer = 0;
606. modelAngle2[0] = 0.0;
607. modelAngle2[3] = 0.0;
608. modelAngle2[6] = 0.0;
609. modelAngle2[9] = 0.0;
610. modelAngle2[12] = 0.0;
611. modelAngle2[15] = 0.0;
612. modelAngle2[18] = 0.0;
613. modelAngle2[21] = 0.0;
614. modelAngle2[24] = 0.0;
615. }
616. }
618. **if** (rotate1[2] == **true**)
619. {
620. modelAngle2[2] += 0.1;
621. modelAngle2[5] += 0.1;
622. modelAngle2[8] += 0.1;
623. modelAngle2[11] += 0.1;
624. modelAngle2[14] += 0.1;
625. modelAngle2[17] += 0.1;
626. modelAngle2[20] += 0.1;
627. modelAngle2[23] += 0.1;
628. modelAngle2[26] += 0.1;
629. timer++;
630. **if** (timer >= 63 && rotate1[2] == **true**)
631. {
632. rotate1[2] = **false**;
633. timer = 0;
634. modelAngle2[2] = 0.0;
635. modelAngle2[5] = 0.0;
636. modelAngle2[8] = 0.0;
637. modelAngle2[11] = 0.0;
638. modelAngle2[14] = 0.0;
639. modelAngle2[17] = 0.0;
640. modelAngle2[20] = 0.0;
641. modelAngle2[23] = 0.0;
642. modelAngle2[26] = 0.0;
643. }
644. }
646. **if** (rotate1[3] == **true**)
647. {
648. modelAngle3[18] += 0.1;
649. modelAngle3[19] += 0.1;
650. modelAngle3[20] += 0.1;
651. modelAngle3[21] += 0.1;
652. modelAngle3[22] += 0.1;
653. modelAngle3[23] += 0.1;
654. modelAngle3[24] += 0.1;
655. modelAngle3[25] += 0.1;
656. modelAngle3[26] += 0.1;
657. timer++;
658. **if** (timer >= 63 && rotate1[3] == **true**)
659. {
660. rotate1[3] = **false**;
661. timer = 0;
662. modelAngle3[18] = 0.0;
663. modelAngle3[19] = 0.0;
664. modelAngle3[20] = 0.0;
665. modelAngle3[21] = 0.0;
666. modelAngle3[22] = 0.0;
667. modelAngle3[23] = 0.0;
668. modelAngle3[24] = 0.0;
669. modelAngle3[25] = 0.0;
670. modelAngle3[26] = 0.0;
671. }
672. }



677. **if** (rotate1[4] == **true**)
678. {
679. modelAngle3[0] += 0.1;
680. modelAngle3[1] += 0.1;
681. modelAngle3[2] += 0.1;
682. modelAngle3[3] += 0.1;
683. modelAngle3[4] += 0.1;
684. modelAngle3[5] += 0.1;
685. modelAngle3[6] += 0.1;
686. modelAngle3[7] += 0.1;
687. modelAngle3[8] += 0.1;
688. timer++;
689. **if** (timer >= 63 && rotate1[4] == **true**)
690. {
691. rotate1[4] = **false**;
692. timer = 0;
693. modelAngle3[0] = 0.0;
694. modelAngle3[1] = 0.0;
695. modelAngle3[2] = 0.0;
696. modelAngle3[3] = 0.0;
697. modelAngle3[4] = 0.0;
698. modelAngle3[5] = 0.0;
699. modelAngle3[6] = 0.0;
700. modelAngle3[7] = 0.0;
701. modelAngle3[8] = 0.0;
702. }
703. }
705. **if** (rotate1[5] == **true**)
706. {
707. modelAngle3[9] += 0.1;
708. modelAngle3[10] += 0.1;
709. modelAngle3[11] += 0.1;
710. modelAngle3[12] += 0.1;
711. modelAngle3[13] += 0.1;
712. modelAngle3[14] += 0.1;
713. modelAngle3[15] += 0.1;
714. modelAngle3[16] += 0.1;
715. modelAngle3[17] += 0.1;
716. timer++;
717. **if** (timer >= 63 && rotate1[5] == **true**)
718. {
719. rotate1[5] = **false**;
720. timer = 0;
721. modelAngle3[9] = 0.0;
722. modelAngle3[10] = 0.0;
723. modelAngle3[11] = 0.0;
724. modelAngle3[12] = 0.0;
725. modelAngle3[13] = 0.0;
726. modelAngle3[14] = 0.0;
727. modelAngle3[15] = 0.0;
728. modelAngle3[16] = 0.0;
729. modelAngle3[17] = 0.0;
730. }
731. }
733. **if** (rotate1[6] == **true**)
734. {
735. modelAngle4[3] += 0.1;
736. modelAngle4[4] += 0.1;
737. modelAngle4[5] += 0.1;
738. modelAngle4[12] += 0.1;
739. modelAngle4[13] += 0.1;
740. modelAngle4[14] += 0.1;
741. modelAngle4[21] += 0.1;
742. modelAngle4[22] += 0.1;
743. modelAngle4[23] += 0.1;
744. timer++;
745. **if** (timer >= 63 && rotate1[6] == **true**)
746. {
747. rotate1[6] = **false**;
748. timer = 0;
749. modelAngle4[3] = 0.0;
750. modelAngle4[4] = 0.0;
751. modelAngle4[5] = 0.0;
752. modelAngle4[12] = 0.0;
753. modelAngle4[13] = 0.0;
754. modelAngle4[14] = 0.0;
755. modelAngle4[21] = 0.0;
756. modelAngle4[22] = 0.0;
757. modelAngle4[23] = 0.0;
758. }
759. }
761. **if** (rotate1[7] == **true**)
762. {
763. modelAngle4[0] += 0.1;
764. modelAngle4[1] += 0.1;
765. modelAngle4[2] += 0.1;
766. modelAngle4[9] += 0.1;
767. modelAngle4[10] += 0.1;
768. modelAngle4[11] += 0.1;
769. modelAngle4[18] += 0.1;
770. modelAngle4[19] += 0.1;
771. modelAngle4[20] += 0.1;
772. timer++;
773. **if** (timer >= 63 && rotate1[7] == **true**)
774. {
775. rotate1[7] = **false**;
776. timer = 0;
777. modelAngle4[0] = 0.0;
778. modelAngle4[1] = 0.0;
779. modelAngle4[2] = 0.0;
780. modelAngle4[9] = 0.0;
781. modelAngle4[10] = 0.0;
782. modelAngle4[11] = 0.0;
783. modelAngle4[18] = 0.0;
784. modelAngle4[19] = 0.0;
785. modelAngle4[20] = 0.0;
786. }
787. }
789. **if** (rotate1[8] == **true**)
790. {
791. modelAngle4[6] += 0.1;
792. modelAngle4[7] += 0.1;
793. modelAngle4[8] += 0.1;
794. modelAngle4[15] += 0.1;
795. modelAngle4[16] += 0.1;
796. modelAngle4[17] += 0.1;
797. modelAngle4[24] += 0.1;
798. modelAngle4[25] += 0.1;
799. modelAngle4[26] += 0.1;
800. timer++;
801. **if** (timer >= 63 && rotate1[8] == **true**)
802. {
803. rotate1[8] = **false**;
804. timer = 0;
805. modelAngle4[6] = 0.0;
806. modelAngle4[7] = 0.0;
807. modelAngle4[8] = 0.0;
808. modelAngle4[15] = 0.0;
809. modelAngle4[16] = 0.0;
810. modelAngle4[17] = 0.0;
811. modelAngle4[24] = 0.0;
812. modelAngle4[25] = 0.0;
813. modelAngle4[26] = 0.0;
814. }
815. }
817. **if** (movingLight == **false**) {     // moving object rotation and z displacement
818. **if** (keyStatus[GLFW\_KEY\_LEFT])           modelAngle.y += 0.05f;
819. **if** (keyStatus[GLFW\_KEY\_RIGHT])          modelAngle.y -= 0.05f;
820. **if** (keyStatus[GLFW\_KEY\_UP])             modelAngle.x += 0.05f;
821. **if** (keyStatus[GLFW\_KEY\_DOWN])           modelAngle.x -= 0.05f;
822. **if** (keyStatus[GLFW\_KEY\_KP\_ADD])         modelDisp.z += 0.10f;
823. **if** (keyStatus[GLFW\_KEY\_KP\_SUBTRACT])    modelDisp.z -= 0.10f;
824. }
825. **else** {                          // moving light displacement x y z
826. **if** (keyStatus[GLFW\_KEY\_LEFT])           lightDisp[movingLightNumber].x -= 0.05f;
827. **if** (keyStatus[GLFW\_KEY\_RIGHT])          lightDisp[movingLightNumber].x += 0.05f;
828. **if** (keyStatus[GLFW\_KEY\_UP])             lightDisp[movingLightNumber].y += 0.05f;
829. **if** (keyStatus[GLFW\_KEY\_DOWN])           lightDisp[movingLightNumber].y -= 0.05f;
830. **if** (keyStatus[GLFW\_KEY\_KP\_ADD])         lightDisp[movingLightNumber].z += 0.05f;
831. **if** (keyStatus[GLFW\_KEY\_KP\_SUBTRACT])    lightDisp[movingLightNumber].z -= 0.05f;
832. }
833. }
835. **void** render(GLfloat currentTime) {
837. //==============First Pass====================
838. //---Render framebuffer to texture
839. glBindFramebuffer(GL\_FRAMEBUFFER, framebuffer);
840. glFramebufferTexture2D(GL\_FRAMEBUFFER, GL\_COLOR\_ATTACHMENT0, GL\_TEXTURE\_2D, framebufferTexture, 0);
842. glViewport(0, 0, windowWidth, windowHeight);
844. // Clear colour buffer
845. glm::vec4 backgroundColor = glm::vec4(0.2f, 0.2f, 0.2f, 0.2f); glClearBufferfv(GL\_COLOR, 0, &backgroundColor[0]);
847. // Clear Deep buffer
848. **static** **const** GLfloat one = 1.0f; glClearBufferfv(GL\_DEPTH, 0, &one);
850. // Enable blend
851. glEnable(GL\_BLEND);
853. glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA);

856. // ----------draw main model------------
857. glUseProgram(objectModel.program);
858. glBindVertexArray(objectModel.vao);
859. glUniformMatrix4fv(objectModel.proj\_location, 1, GL\_FALSE, &proj\_matrix[0][0]);
861. glUniform4f(glGetUniformLocation(objectModel.program, "viewPosition"), cameraPosition.x, cameraPosition.y, cameraPosition.z, 1.0);
862. glUniform4f(glGetUniformLocation(objectModel.program, "ia"), ia.r, ia.g, ia.b, 1.0);
863. glUniform1f(glGetUniformLocation(objectModel.program, "ka"), ka);
865. glUniform1f(glGetUniformLocation(objectModel.program, "kd"), 1.0f);
866. glUniform4f(glGetUniformLocation(objectModel.program, "is"), is.r, is.g, is.b, 1.0);
867. glUniform1f(glGetUniformLocation(objectModel.program, "ks"), 1.0f);
868. glUniform1f(glGetUniformLocation(objectModel.program, "shininess"), 32.0f);
870. glUniform1f(glGetUniformLocation(objectModel.program, "lightConstant"), 1.0f);
871. glUniform1f(glGetUniformLocation(objectModel.program, "lightLinear"), 0.7f);
872. glUniform1f(glGetUniformLocation(objectModel.program, "lightQuadratic"), 1.8f);
874. **for** (**int** i = 0; i < 4; i++) {
875. glUniform4f(glGetUniformLocation(objectModel.program, ("lights[" + to\_string(i) + "].lightPosition").c\_str()), lightDisp[i].x, lightDisp[i].y, lightDisp[i].z, 1.0f);
876. glUniform4f(glGetUniformLocation(objectModel.program, ("lights[" + to\_string(i) + "].id").c\_str()), id[i].r, id[i].g, id[i].b, 1.0f);
877. }
879. //glUniform4f(glGetUniformLocation(objectModel.program, "lightPosition"), cameraPosition.x, cameraPosition.y, cameraPosition.z, 1.0);
880. //glUniform4f(glGetUniformLocation(objectModel.program, "lightSpotDirection"), cameraFront.x, cameraFront.y, cameraFront.z, 0.0);
881. //glUniform1f(glGetUniformLocation(objectModel.program, "lightSpotCutOff"), glm::cos(glm::radians(12.5f)));
882. //glUniform1f(glGetUniformLocation(objectModel.program, "lightSpotOuterCutOff"), glm::cos(glm::radians(15.0f)));
884. // Bind textures and samplers - using 0 as I know there is only one texture - need to extend.
885. glActiveTexture(GL\_TEXTURE0);
886. glBindTexture(GL\_TEXTURE\_2D, objectModel.texture[0]);
887. glUniform1i(objectModel.tex\_location, 0);
889. glm::mat4 viewMatrix = glm::lookAt(cameraPosition,                  // eye
890. cameraPosition + cameraFront,   // centre
891. cameraUp);                      // up
893. **for** (**int** i = 0; i < 27; i++) {
894. glm::mat4 modelMatrix = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, 0.0f, 0.0f));// modelDisp.z));
896. modelMatrix = glm::rotate(modelMatrix, modelAngle2[i], glm::vec3(1.0f, 0.0f, 0.0f));
897. modelMatrix = glm::rotate(modelMatrix, modelAngle3[i], glm::vec3(0.0f, 1.0f, 0.0f));
898. modelMatrix = glm::rotate(modelMatrix, modelAngle4[i], glm::vec3(0.0f, 0.0f, 1.0f));
900. modelMatrix = glm::translate(modelMatrix, modelPositions[i]);
901. modelMatrix = glm::rotate(modelMatrix, modelAngle.x + modelRotations[0].x, glm::vec3(1.0f, 0.0f, 0.0f));
902. modelMatrix = glm::rotate(modelMatrix, modelAngle.y + modelRotations[0].y, glm::vec3(0.0f, 1.0f, 0.0f));
903. modelMatrix = glm::scale(modelMatrix, glm::vec3(0.2f, 0.2f, 0.2f));
905. glm::mat4 mv\_matrix = viewMatrix \* modelMatrix;
907. glUniformMatrix4fv(glGetUniformLocation(objectModel.program, "model\_matrix"), 1, GL\_FALSE, &modelMatrix[0][0]);
908. glUniformMatrix4fv(glGetUniformLocation(objectModel.program, "view\_matrix"), 1, GL\_FALSE, &viewMatrix[0][0]);
910. glDrawArrays(GL\_TRIANGLES, 0, objectModel.out\_vertices.size());
911. }
913. // ----------draw light------------
914. glUseProgram(lightModel.program);
915. glBindVertexArray(lightModel.vao);
916. glUniformMatrix4fv(lightModel.proj\_location, 1, GL\_FALSE, &proj\_matrix[0][0]);
918. // Bind textures and samplers - using 0 as I know there is only one texture - need to extend.
919. glActiveTexture(GL\_TEXTURE0);
920. glBindTexture(GL\_TEXTURE\_2D, lightModel.texture[0]);
921. glUniform1i(lightModel.tex\_location, 0);
923. **for** (**int** i = 0; i < 6; i++) {
924. glm::mat4 modelMatrixLight = glm::translate(glm::mat4(1.0f), glm::vec3(lightDisp[i].x, lightDisp[i].y, lightDisp[i].z));
925. modelMatrixLight = glm::scale(modelMatrixLight, glm::vec3(0.2f, 0.2f, 0.2f));
926. glm::mat4 mv\_matrixLight = viewMatrix \* modelMatrixLight;
928. glUniformMatrix4fv(lightModel.mv\_location, 1, GL\_FALSE, &mv\_matrixLight[0][0]);
929. glDrawArrays(GL\_TRIANGLES, 0, lightModel.out\_vertices.size());
930. }
932. //==============Second Pass===================
933. glBindFramebuffer(GL\_FRAMEBUFFER, 0);   // Disable rendering framebuffer to texture - aka render normally.
935. glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
936. glClear(GL\_COLOR\_BUFFER\_BIT);
937. glDisable(GL\_DEPTH\_TEST); //not needed as we are just displaying a single quad
939. glUseProgram(displayProgram);
940. glBindVertexArray(displayVao);
941. glActiveTexture(GL\_TEXTURE0);
942. glBindTexture(GL\_TEXTURE\_2D, framebufferTexture);
943. glDrawArrays(GL\_TRIANGLES, 0, 6);
944. glBindVertexArray(0);
945. }
947. **void** onResizeCallback(GLFWwindow\* window, **int** w, **int** h) {
948. windowWidth = w;
949. windowHeight = h;
951. aspect = (**float**)w / (**float**)h;
952. proj\_matrix = glm::perspective(glm::radians(fovy), aspect, 0.1f, 1000.0f);
953. }
955. **void** onKeyCallback(GLFWwindow\* window, **int** key, **int** scancode, **int** action, **int** mods) {
956. **if** (action == GLFW\_PRESS) keyStatus[key] = **true**;
957. **else** **if** (action == GLFW\_RELEASE) keyStatus[key] = **false**;
959. **if** (key == GLFW\_KEY\_ESCAPE && action == GLFW\_PRESS)
960. glfwSetWindowShouldClose(window, GLFW\_TRUE);
961. }
963. **void** onMouseButtonCallback(GLFWwindow\* window, **int** button, **int** action, **int** mods) {
965. }
967. **void** onMouseMoveCallback(GLFWwindow\* window, **double** x, **double** y) {
968. **int** mouseX = **static\_cast**<**int**>(x);
969. **int** mouseY = **static\_cast**<**int**>(y);
971. **if** (firstMouse) {
972. lastX = (GLfloat)mouseX; lastY = (GLfloat)mouseY; firstMouse = **false**;
973. }
975. GLfloat xoffset = mouseX - lastX;
976. GLfloat yoffset = lastY - mouseY; // Reversed
977. lastX = (GLfloat)mouseX; lastY = (GLfloat)mouseY;
979. GLfloat sensitivity = 0.05f;
980. xoffset \*= sensitivity; yoffset \*= sensitivity;
982. yaw += xoffset; pitch += yoffset;
984. // check for pitch out of bounds otherwise screen gets flipped
985. **if** (pitch > 89.0f) pitch = 89.0f;
986. **if** (pitch < -89.0f) pitch = -89.0f;
988. glm::vec3 front;
989. front.x = cos(glm::radians(yaw)) \* cos(glm::radians(pitch));
990. front.y = sin(glm::radians(pitch));
991. front.z = sin(glm::radians(yaw)) \* cos(glm::radians(pitch));
993. cameraFront = glm::normalize(front);
995. }
997. **static** **void** onMouseWheelCallback(GLFWwindow\* window, **double** xoffset, **double** yoffset) {
998. **int** yoffsetInt = **static\_cast**<**int**>(yoffset);
1000. fovy += yoffsetInt;
1001. **if** (fovy >= 1.0f && fovy <= 45.0f) fovy -= (GLfloat)yoffset;
1002. **if** (fovy <= 1.0f) fovy = 1.0f;
1003. **if** (fovy >= 45.0f) fovy = 45.0f;
1004. proj\_matrix = glm::perspective(glm::radians(fovy), aspect, 0.1f, 1000.0f);
1005. }
1007. **void** debugGL() {
1008. //Output some debugging information
1009. cout << "VENDOR: " << (**char** \*)glGetString(GL\_VENDOR) << endl;
1010. cout << "VERSION: " << (**char** \*)glGetString(GL\_VERSION) << endl;
1011. cout << "RENDERER: " << (**char** \*)glGetString(GL\_RENDERER) << endl;
1013. // Enable Opengl Debug
1014. //glEnable(GL\_DEBUG\_OUTPUT);
1015. glEnable(GL\_DEBUG\_OUTPUT\_SYNCHRONOUS);
1016. glDebugMessageCallback((GLDEBUGPROC)openGLDebugCallback, nullptr);
1017. glDebugMessageControl(GL\_DONT\_CARE, GL\_DONT\_CARE, GL\_DONT\_CARE, 0, NULL, **true**);
1018. }
1020. **static** **void** APIENTRY openGLDebugCallback(GLenum source,
1021. GLenum type,
1022. GLuint id,
1023. GLenum severity,
1024. GLsizei length,
1025. **const** GLchar\* message,
1026. **const** GLvoid\* userParam) {
1028. cout << "---------------------opengl-callback------------" << endl;
1029. cout << "Message: " << message << endl;
1030. cout << "type: ";
1031. **switch** (type) {
1032. **case** GL\_DEBUG\_TYPE\_ERROR:
1033. cout << "ERROR";
1034. **break**;
1035. **case** GL\_DEBUG\_TYPE\_DEPRECATED\_BEHAVIOR:
1036. cout << "DEPRECATED\_BEHAVIOR";
1037. **break**;
1038. **case** GL\_DEBUG\_TYPE\_UNDEFINED\_BEHAVIOR:
1039. cout << "UNDEFINED\_BEHAVIOR";
1040. **break**;
1041. **case** GL\_DEBUG\_TYPE\_PORTABILITY:
1042. cout << "PORTABILITY";
1043. **break**;
1044. **case** GL\_DEBUG\_TYPE\_PERFORMANCE:
1045. cout << "PERFORMANCE";
1046. **break**;
1047. **case** GL\_DEBUG\_TYPE\_OTHER:
1048. cout << "OTHER";
1049. **break**;
1050. }
1051. cout << " --- ";
1053. cout << "id: " << id << " --- ";
1054. cout << "severity: ";
1055. **switch** (severity) {
1056. **case** GL\_DEBUG\_SEVERITY\_LOW:
1057. cout << "LOW";
1058. **break**;
1059. **case** GL\_DEBUG\_SEVERITY\_MEDIUM:
1060. cout << "MEDIUM";
1061. **break**;
1062. **case** GL\_DEBUG\_SEVERITY\_HIGH:
1063. cout << "HIGH";
1064. **break**;
1065. **case** GL\_DEBUG\_SEVERITY\_NOTIFICATION:
1066. cout << "NOTIFICATION";
1067. }
1068. cout << endl;
1069. cout << "-----------------------------------------" << endl;
1070. }
1072. string readShader(string name) {
1073. string vs\_text;
1074. std::ifstream vs\_file(name);
1076. string vs\_line;
1077. **if** (vs\_file.is\_open()) {
1079. **while** (getline(vs\_file, vs\_line)) {
1080. vs\_text += vs\_line;
1081. vs\_text += '\n';
1082. }
1083. vs\_file.close();
1084. }
1085. **return** vs\_text;
1086. }
1088. **void**  checkErrorShader(GLuint shader) {
1089. // Get log lenght
1090. GLint maxLength;
1091. glGetShaderiv(shader, GL\_INFO\_LOG\_LENGTH, &maxLength);
1093. // Init a string for it
1094. std::vector<GLchar> errorLog(maxLength);
1096. **if** (maxLength > 0) {
1097. // Get the log file
1098. glGetShaderInfoLog(shader, maxLength, &maxLength, &errorLog[0]);
1100. cout << "--------------Shader compilation error-------------\n";
1101. cout << errorLog.data();
1102. }
1104. }
1106. **void** readObj(string name, **struct** modelObject \*obj) {
1107. cout << "Loading model " << name << "\n";
1109. std::vector< unsigned **int** > vertexIndices, uvIndices, normalIndices;
1110. std::vector< string > materials, textures;
1111. std::vector< glm::vec3 > obj\_vertices;
1112. std::vector< glm::vec2 > obj\_uvs;
1113. std::vector< glm::vec3 > obj\_normals;
1115. std::ifstream dataFile(name);
1117. string rawData;     // store the raw data.
1118. **int** countLines = 0;
1119. **if** (dataFile.is\_open()) {
1120. string dataLineRaw;
1121. **while** (getline(dataFile, dataLineRaw)) {
1122. rawData += dataLineRaw;
1123. rawData += "\n";
1124. countLines++;
1125. }
1126. dataFile.close();
1127. }
1129. cout << "Finished reading model file " << name << "\n";
1131. istringstream rawDataStream(rawData);
1132. string dataLine;
1133. **int** linesDone = 0;
1134. **while** (std::getline(rawDataStream, dataLine)) {
1135. **if** (dataLine.find("v ") != string::npos) {  // does this line have a vector?
1136. glm::vec3 vertex;
1138. **int** foundStart = dataLine.find(" ");  **int** foundEnd = dataLine.find(" ", foundStart + 1);
1139. vertex.x = stof(dataLine.substr(foundStart, foundEnd - foundStart));
1141. foundStart = foundEnd; foundEnd = dataLine.find(" ", foundStart + 1);
1142. vertex.y = stof(dataLine.substr(foundStart, foundEnd - foundStart));
1144. foundStart = foundEnd; foundEnd = dataLine.find(" ", foundStart + 1);
1145. vertex.z = stof(dataLine.substr(foundStart, foundEnd - foundStart));
1147. obj\_vertices.push\_back(vertex);
1148. }
1149. **else** **if** (dataLine.find("vt ") != string::npos) {    // does this line have a uv coordinates?
1150. glm::vec2 uv;
1152. **int** foundStart = dataLine.find(" ");  **int** foundEnd = dataLine.find(" ", foundStart + 1);
1153. uv.x = stof(dataLine.substr(foundStart, foundEnd - foundStart));
1155. foundStart = foundEnd; foundEnd = dataLine.find(" ", foundStart + 1);
1156. uv.y = stof(dataLine.substr(foundStart, foundEnd - foundStart));
1158. obj\_uvs.push\_back(uv);
1159. }
1160. **else** **if** (dataLine.find("vn ") != string::npos) { // does this line have a normal coordinates?
1161. glm::vec3 normal;
1163. **int** foundStart = dataLine.find(" ");  **int** foundEnd = dataLine.find(" ", foundStart + 1);
1164. normal.x = stof(dataLine.substr(foundStart, foundEnd - foundStart));
1166. foundStart = foundEnd; foundEnd = dataLine.find(" ", foundStart + 1);
1167. normal.y = stof(dataLine.substr(foundStart, foundEnd - foundStart));
1169. foundStart = foundEnd; foundEnd = dataLine.find(" ", foundStart + 1);
1170. normal.z = stof(dataLine.substr(foundStart, foundEnd - foundStart));
1172. obj\_normals.push\_back(normal);
1173. }
1174. **else** **if** (dataLine.find("f ") != string::npos) { // does this line defines a triangle face?
1175. string parts[3];
1177. **int** foundStart = dataLine.find(" ");  **int** foundEnd = dataLine.find(" ", foundStart + 1);
1178. parts[0] = dataLine.substr(foundStart + 1, foundEnd - foundStart - 1);
1180. foundStart = foundEnd; foundEnd = dataLine.find(" ", foundStart + 1);
1181. parts[1] = dataLine.substr(foundStart + 1, foundEnd - foundStart - 1);
1183. foundStart = foundEnd; foundEnd = dataLine.find(" ", foundStart + 1);
1184. parts[2] = dataLine.substr(foundStart + 1, foundEnd - foundStart - 1);
1186. **for** (**int** i = 0; i < 3; i++) {        // for each part
1188. vertexIndices.push\_back(stoul(parts[i].substr(0, parts[i].find("/"))));
1190. **int** firstSlash = parts[i].find("/"); **int** secondSlash = parts[i].find("/", firstSlash + 1);
1192. **if** ((firstSlash + 1) != (secondSlash)) {    // there are texture coordinates.
1193. uvIndices.push\_back(stoul(parts[i].substr(firstSlash + 1, secondSlash - firstSlash + 1)));
1194. }

1197. normalIndices.push\_back(stoul(parts[i].substr(secondSlash + 1)));
1199. }
1200. }
1201. **else** **if** (dataLine.find("mtllib ") != string::npos) { // does this object have a material?
1202. materials.push\_back(dataLine.substr(dataLine.find(" ") + 1));
1203. }
1205. linesDone++;
1207. **if** (linesDone % 50000 == 0) {
1208. cout << "Parsed " << linesDone << " of " << countLines << " from model...\n";
1209. }
1211. }
1213. // Double check here in which coordinate system you exported your models - and flip or not the vertices...
1215. /\*for (unsigned int i = 0; i < vertexIndices.size(); i += 3) {
1216. (\*obj).out\_vertices.push\_back(obj\_vertices[vertexIndices[i+2] - 1]);
1217. (\*obj).out\_normals.push\_back(obj\_normals[normalIndices[i+2] - 1]);
1218. (\*obj).out\_uvs.push\_back(obj\_uvs[uvIndices[i+2] - 1]);
1220. (\*obj).out\_vertices.push\_back(obj\_vertices[vertexIndices[i+1] - 1]);
1221. (\*obj).out\_normals.push\_back(obj\_normals[normalIndices[i+1] - 1]);
1222. (\*obj).out\_uvs.push\_back(obj\_uvs[uvIndices[i + 1] - 1]);
1224. (\*obj).out\_vertices.push\_back(obj\_vertices[vertexIndices[i] - 1]);
1225. (\*obj).out\_normals.push\_back(obj\_normals[normalIndices[i] - 1]);
1226. (\*obj).out\_uvs.push\_back(obj\_uvs[uvIndices[i + 0] - 1]);
1227. }\*/

1230. **for** (unsigned **int** i = 0; i < vertexIndices.size(); i++) {
1231. (\*obj).out\_vertices.push\_back(obj\_vertices[vertexIndices[i] - 1]);
1232. (\*obj).out\_normals.push\_back(obj\_normals[normalIndices[i] - 1]);
1233. (\*obj).out\_uvs.push\_back(obj\_uvs[uvIndices[i] - 1]);
1234. }

1237. // Load Materials
1238. **for** (unsigned **int** i = 0; i < materials.size(); i++) {
1240. std::ifstream dataFileMat(materials[i]);
1242. string rawDataMat;      // store the raw data.
1243. **int** countLinesMat = 0;
1244. **if** (dataFileMat.is\_open()) {
1245. string dataLineRawMat;
1246. **while** (getline(dataFileMat, dataLineRawMat)) {
1247. rawDataMat += dataLineRawMat;
1248. rawDataMat += "\n";
1249. }
1250. dataFileMat.close();
1251. }
1253. istringstream rawDataStreamMat(rawDataMat);
1254. string dataLineMat;
1255. **while** (std::getline(rawDataStreamMat, dataLineMat)) {
1256. **if** (dataLineMat.find("map\_Kd ") != string::npos) {  // does this line have a texture map?
1257. textures.push\_back(dataLineMat.substr(dataLineMat.find(" ") + 1));
1258. }
1259. }
1260. }
1262. (\*obj).texture = **new** GLuint[textures.size()];       // Warning possible memory leak here - there is a new here, where is your delete?
1263. glCreateTextures(GL\_TEXTURE\_2D, textures.size(), (\*obj).texture);
1265. **for** (unsigned **int** i = 0; i < textures.size(); i++) {
1266. readTexture(textures[i], (\*obj).texture[i]);
1267. }
1269. cout << "done";
1270. }
1272. **void** readTexture(string name, GLuint textureName) {

1275. gli::texture tex = gli::load(name);
1276. **if** (tex.empty()) {
1277. cout << "Unable to load file " << name;
1278. }
1280. gli::gl texGl(gli::gl::PROFILE\_GL33);
1281. gli::gl::format **const** texFormat = texGl.translate(tex.format(), tex.swizzles());
1282. //GLenum texTarget = texGl.translate(tex.target());
1284. // Load and create a texture
1285. glBindTexture(GL\_TEXTURE\_2D, textureName); // All upcoming operations now have effect on this texture object
1287. glm::tvec3<GLsizei> **const** texExtent(tex.extent());
1288. GLsizei **const** texFaceTotal = **static\_cast**<GLsizei>(tex.layers() \* tex.faces());
1290. // Note: This only loads GL\_TEXTURE\_2D - for the complete code please visit >> http://gli.g-truc.net/
1291. glTexStorage2D(GL\_TEXTURE\_2D, **static\_cast**<GLint>(tex.levels()), texFormat.Internal, texExtent.x, texExtent.y);
1293. **for** (std::**size\_t** Layer = 0; Layer < tex.layers(); ++Layer) {
1294. **for** (std::**size\_t** Face = 0; Face < tex.faces(); ++Face) {
1295. **for** (std::**size\_t** Level = 0; Level < tex.levels(); ++Level) {
1296. glTextureSubImage2D(textureName, **static\_cast**<GLint>(Level),
1297. 0, 0,
1298. texExtent.x, texExtent.y,
1299. texFormat.External, texFormat.Type, tex.data(Layer, Face, Level));
1301. }
1302. }
1303. }
1305. // This only works for 2D Textures...
1306. // Set the texture wrapping parameters
1307. glTextureParameteri(textureName, GL\_TEXTURE\_WRAP\_S, GL\_REPEAT);
1308. glTextureParameteri(textureName, GL\_TEXTURE\_WRAP\_T, GL\_REPEAT);
1310. // Set texture filtering parameters
1311. glTextureParameteri(textureName, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);
1312. glTextureParameteri(textureName, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);
1314. glGenerateMipmap(GL\_TEXTURE\_2D);
1316. glBindTexture(GL\_TEXTURE\_2D, 0);// Unbind texture when done, so we won't accidentily mess up our texture.


1320. }

#### Vs\_source.txt

1. #version 450 core
3. layout (location = 0) in vec4 position;
4. layout (location = 2) in vec2 tc;
6. out VS\_OUT
7. {
8. vec2 tc;
9. } vs\_out;

12. uniform mat4 mv\_matrix;
13. uniform mat4 proj\_matrix;
15. **void** main(**void**)
16. {
17. gl\_Position = proj\_matrix \* mv\_matrix \* position;
18. vs\_out.tc = tc;
19. }

#### Fs\_source.txt

1. #version 450 core
3. out vec4 color;
5. in VS\_OUT
6. {
7. vec2 tc;
8. } fs\_in;
10. layout(binding=0) uniform sampler2D tex;
12. **void** main(**void**)
13. {
14. color = texture(tex, fs\_in.tc);
15. }

#### Vs\_display.glsl

1. #version 450 core
3. layout (location = 0) in vec2 position;
4. layout (location = 1) in vec2 texCoords;
6. out vec2 TexCoords;
8. **void** main()
9. {
10. gl\_Position = vec4(position.x, position.y, 0.0f, 1.0f);
11. TexCoords = texCoords;
12. }

#### Fs\_display.glsl

1. #version 450 core
3. in vec2 TexCoords;
4. out vec4 color;
6. uniform sampler2D screenTexture;
8. uniform **float** time;
9. **const** **float** blurSizeH = 1.0 / 500.0;
10. **const** **float** blurSizeV = 1.0 / 500.0;
12. **void** main()
13. {
15. //color = texture( screenTexture, TexCoords + 0.005\*vec2( sin(time+1024.0\*TexCoords.x),cos(time+768.0\*TexCoords.y)) ).xyz;
16. vec4 sum = vec4(0.0);
17. **for** (**int** x = -4; x <= 4; x++)
18. **for** (**int** y = -4; y <= 4; y++)
19. sum += texture(
20. screenTexture,
21. vec2(TexCoords.x + x \* blurSizeH, TexCoords.y + y \* blurSizeV)
22. ) / 81.0;
23. color = sum;
24. }

#### Vs\_model.glsl

1. #version 450 core
3. layout (location = 0) in vec4 position;
4. layout (location = 1) in vec2 tc;
5. layout (location = 2) in vec4 normals;
7. out VS\_OUT
8. {
9. vec2 tc;
10. vec4 normals;
11. vec4 fragPos;
12. } vs\_out;
14. uniform mat4 model\_matrix;
15. uniform mat4 view\_matrix;
16. uniform mat4 proj\_matrix;
18. **void** main(**void**)
19. {
20. gl\_Position = proj\_matrix \* view\_matrix \* model\_matrix \* position;
21. vs\_out.tc = tc;
23. vec3 normalsT = mat3(transpose(inverse(model\_matrix))) \* vec3(normals.xyz);
24. vs\_out.normals = vec4(normalsT, 1.0);
26. vs\_out.fragPos = model\_matrix \* position;
27. }

#### Fs\_model.glsl

1. #version 450 core
3. out vec4 color;
5. in VS\_OUT
6. {
7. vec2 tc;
8. vec4 normals;
9. vec4 fragPos;
10. } fs\_in;
12. layout(binding=0) uniform sampler2D tex;
14. uniform mat4 model\_matrix;
16. **struct** lightStruc {
17. vec4 lightPosition;
18. vec4 id;
19. };
20. #define LIGHTS 4
21. uniform lightStruc lights[LIGHTS];
23. uniform vec4 viewPosition;
25. uniform vec4 ia;        // Ambient colour
26. uniform **float** ka;       // Ambient constant
27. //uniform vec4 id;      // diffuse colour
28. uniform **float** kd;       // Diffuse constant
29. uniform vec4 is;        // specular colour
30. uniform **float** ks;       // specular constant
31. uniform **float** shininess;// shininess constant
33. uniform **float** lightConstant;
34. uniform **float** lightLinear;
35. uniform **float** lightQuadratic;
37. uniform vec4 lightSpotDirection;
38. uniform **float** lightSpotCutOff;
39. uniform **float** lightSpotOuterCutOff;

42. **void** main(**void**){

45. // Ambient
46. vec3 ambient = ka \* ia.rgb;
48. vec3 totalAmbient;
49. vec3 totalDiffuse;
50. vec3 totalSpecular;
52. **for**(**int** i = 0; i < LIGHTS; i++){
53. // Diffuse
54. vec4 lightDir = normalize(lights[i].lightPosition - fs\_in.fragPos);
55. **float** diff = max(dot(normalize(fs\_in.normals), lightDir), 0.0);
56. vec3 diffuse = kd \* lights[i].id.rgb \* diff;

59. // Specular
60. vec4 viewDir = normalize(viewPosition - fs\_in.fragPos);
61. vec4 reflectDir = reflect(-lightDir, normalize(fs\_in.normals));
62. **float** spec = pow(max(dot(viewDir, reflectDir), 0.0), shininess);
63. vec3 specular = ks \* is.rgb \* spec;

66. // Attenuation
67. **float** distance    = length(lights[i].lightPosition - fs\_in.fragPos);
68. **float** attenuation = 1.0f / (lightConstant + lightLinear \* distance + lightQuadratic \* (distance \* distance));
70. ambient  \*= attenuation;
71. diffuse  \*= attenuation;
72. specular \*= attenuation;
74. totalAmbient += ambient;
75. totalDiffuse += diffuse;
76. totalSpecular += specular;
77. }


81. // Light
82. color = vec4(totalAmbient + totalDiffuse + totalSpecular, 1.0) \* texture(tex, fs\_in.tc);
83. }

#### Vs\_light.glsl

1. #version 450 core
3. layout (location = 0) in vec4 position;
4. layout (location = 1) in vec2 tc;
6. out VS\_OUT
7. {
8. vec2 tc;
9. } vs\_out;
11. uniform mat4 mv\_matrix;
12. uniform mat4 proj\_matrix;
14. **void** main(**void**)
15. {
16. gl\_Position = proj\_matrix \* mv\_matrix \* position;
17. vs\_out.tc = tc;
18. }

#### Fs\_light.glsl

1. #version 450 core
3. out vec4 color;
5. in VS\_OUT
6. {
7. vec2 tc;
8. } fs\_in;
10. layout(binding=0) uniform sampler2D tex;
12. **void** main(**void**)
13. {
14. color = vec4( texture(tex, fs\_in.tc).xyz, 0.9);
15. }

#### Sphere.mtl

# Blender MTL File: 'None'

# Material Count: 1

newmtl Material.001

Ns 96.078431

Ka 1.000000 1.000000 1.000000

Kd 0.640000 0.640000 0.640000

Ks 0.500000 0.500000 0.500000

Ke 0.000000 0.000000 0.000000

Ni 1.000000

d 1.000000

illum 2

map\_Kd sphere\_map.ktx

#### RubiksObj.mtl

# Blender MTL File: 'None'

# Material Count: 1

newmtl Material

Ns 96.078431

Ka 1.000000 1.000000 1.000000

Kd 0.640000 0.640000 0.640000

Ks 0.500000 0.500000 0.500000

Ke 0.000000 0.000000 0.000000

Ni 1.000000

d 1.000000

illum 2

map\_Kd RubiksKTX.ktx