CS-499 Module Three Milestone Two  
3-2 Enhancement One Software Design and Engineering  
William Moore  
CS-499-Q1527 Computer Science Capstone 20EW1

Contents

[1. Questions 3](#_Toc51536863)

[A. Briefly describe the artifact. What is it? When was it created? 3](#_Toc51536864)

[B. Justify the inclusion of the artifact in your ePortfolio. Why did you select this item? 3](#_Toc51536865)

[C. Did you meet the course objectives you planned to meet with this enhancement in Module One? 5](#_Toc51536866)

[Do you have any updates to your outcome-coverage plans? 5](#_Toc51536867)

[D. Reflect on the process of enhancing and/or modifying the artifact. What did you learn as you were creating it and improving it? 5](#_Toc51536868)

[2. Original Artifact 6](#_Toc51536869)

[3. Artifact with Enhancements 22](#_Toc51536870)

[4. References 39](#_Toc51536871)

# Questions

## Briefly describe the artifact. What is it? When was it created?

The program artifact comes from CS-330 Comp Graphic and Visualization, which I took in July 2020. The program shows my ability to generate accurate representations of a 3D object using modern OpenGL and its API libraries. The application responds to input devices (keyboard, mouse movement, mouse buttons) that allow for successful navigation around the 3D object and through 3D space, and shows best practices in formatting, commenting, and creating functional logic that produced a reliable fully functional program. The program shows my selected object, which is a 4-legged table, which can be rotated in xyz space, zoomed in and out, along with side lighting.

## Justify the inclusion of the artifact in your ePortfolio. Why did you select this item?

I chose the C++ program artifact for the following reasons:

* It showcases my ability to program in C++.
* It displays my skill at developing and displaying complex 2D and 3D graphical objects on various coordinate planes, using a variety of mathematical methods.

Given that C++ programming is a language used by 4.58% of software engineers (Tiobe) and has been ranked in the top 4 languages since 1990 (Tiobe), I wanted to include a project written in a language I am familiar with. Tiobe lists programming languages and their rankings each month. As part of my self-analysis, I also wanted to show an example of a language I had used many times on projects.

**What specific components of the artifact showcase your skills and abilities in software development?**

The entire artifact showcases my expertise in programming in C++. My experience in using multiple coordinate systems is shown in the "URenderGraphics(void)" function when transforming matrixes from model coordinates to world coordinates, then from world coordinates to the camera coordinates (view matrix transformations), and finally through transformations to the homogeneous coordinates where everything is viewed through a viewport.

My use of the Phong model and how to combine the aspect, diffuse and specular lighting effects is shown in the "fragmentShaderSource" code, which is actually a macro that is compiled and processed by the graphical processing unit (GPU). The GPU is located in the video graphics chip on the computer.

Early on in the development of this program in my CS-330 class, I discovered that my code compiled and linked correctly, but did not produce the correct visual output. I knew from my analysis of the code that the GLSL macros were sent to the graphics processing unit for execution. Upon Googling the GLSL macros in the code, I discovered that there the macros were only compiled at run-time by the GPU. I found additional C++ code that detected the run-time compilation status and displayed errors in the macro compilation. Upon adding the macro compilation code, I found that I had an error in one of the two GLSL macros and after fixing the error, my program output was correct. I used the additional code in every assignment after that in the class and it saved me many hours of debugging the code.

**How was the artifact improved?**

The artifact was improved by completing the lighting functionality for two lights located around my selected object (the 4-legged table). The lights were the key light (most intense lighting effect) and the fill light (a less intense light), located at different points around the object. To make the lights distinguishable from each other, I made the key light green and the fill light red. I placed the key light in the foreground to the left of the object. I placed the fill light on the far side of the object on the right side. The three components of the lighting (ambient, diffuse, and specular) were added using the two lights and the different light intensities and colors. Finally, the Phong lighting model was created using the three components above with the object color using: 

## Did you meet the course objectives you planned to meet with this enhancement in Module One?

I have met the enhancement objectives I planned in this area. I added additional lighting to the 3D scene along with enhancing the ambient, diffuse, and specular lighting effects.

## Do you have any updates to your outcome-coverage plans?

I don't have any other updates to my outcome-coverage plan for this Software Design and Engineering component.

## Reflect on the process of enhancing and/or modifying the artifact. What did you learn as you were creating it and improving it?

I learned much more about basic lighting, which is ambient, diffuse, and specular, and the Phong lighting model. I spent time studying the theory of lighting and lighting models on several websites, along with working through examples of various techniques. Writing code for manipulating my object in various view spaces with lighting works the same as writing code to solve any other problem; there are multiple ways to implement the same effect. I experimented with various ways to create the Phong lighting, such as keeping two ambient lighting results, two diffuse results, and two specular results, then combining all six results with the object color at the "vec3 Phong = " statement. After some testing of various approaches, I decided to use the 'straightforward' approach, to have only one ambient, diffuse and specular result, then adding the lighting together and multiplying with the matrix object color at the "vec3 Phong = " statement.

**What challenges did you face?**

My challenges were in getting the two lights correctly positioned around my table object, and in getting the lighting intensities and colors correct. I spent quite a lot of time changing the relevant values and then compiling/linking the code so I could see the effect.

# Original Artifact

|  |
| --- |
| /\*  \* Name: William Moore  \* School: Southern New Hampshire University  \* Class: CS-330-J6592 Comp Graphic and Visualization 20EW6  \* Assignment: 7-1 Final Submission  \*/  /\* This program uses modern OpenGL and the API libraries to draw  \* 2D and 3D objects with rotation effects, camera locations and  \* lighting effects. In this program, a 4-leg table object is drawn,  \* rotated, and lighted with 2 cameras  \*/  // Header Inclusions  #include <iostream> // Includes C++ i/o stream  #include <GL/glew.h> // Includes glew header  #include <GL/freeglut.h> // Include the freeglut header file  // GLM Math Header inclusions  #include <glm/glm.hpp>  #include <glm/gtc/matrix\_transform.hpp>  #include <glm/gtc/type\_ptr.hpp>  // rgb color definitions for the basic colors  #define Red 1.0f, 0.0f, 0.0f  #define Green 0.0f, 1.0f, 0.0f  #define Blue 0.0f, 0.0f, 1.0f  #define Yellow 1.0f, 1.0f, 0.0f  #define Cyan 0.0f, 1.0f, 1.0f  #define Magenta 1.0f, 0.0f, 1.0f  #define White 1.0f, 1.0f, 1.0f  #define Black 0.0f, 0.0f, 0.0f  #define DullGreen 0.0f, 0.5f, 0.4f  #define Grey 0.5f, 0.5f, 0.5f  using namespace std; // use standard namespace  #define WINDOW\_TITLE "7-1 Final Project (William Moore)" // Macro for window title  // shader source macro  #ifndef GLSL  #define GLSL(Version, Source) "#version " #Version "\n" #Source  #endif  // Variable declarations for shader, window size initialization, buffer and array objects  GLint shaderProgram;  GLint lampShaderProgram;  GLint WindowWidth = 800;  GLint WindowHeight = 600;  GLuint VBO, VAO, LightVAO, EBO, texture;  GLfloat cameraSpeed = 0.0005f; // Movement speed per frame  GLchar currentKey; // Will store key pressed  int modifierKey;  bool bUsePerspectiveView = true;  // pyramid and light color, 0.6f, 0.5f, 0.75f  glm::vec3 objectColor(1.0f, 1.0f, 1.0f);  // Key Light position and scale  glm::vec3 keyLightPosition(-0.8f, 0.0f, 15.0f); // left side of pyramid in foreground  glm::vec3 keyLightScale(0.1f);  glm::vec3 keyLightColor(Green);  // Fill Light position and scale  glm::vec3 fillLightPosition(0.5f, 0.5f, -5.0f); // right back side of the pyramid  glm::vec3 fillLightScale(0.3f);  glm::vec3 fillLightColor(Red);  GLfloat lastMouseX = 400; // Locks mouse cursor at the center of the screen  GLfloat lastMouseY = 300;  GLfloat mouseXOffset; // Mouse offset, yaw and pitch variables  GLfloat mouseYOffset;  GLfloat yaw = 0.0f;  GLfloat pitch = 0.0f;  GLfloat scale\_by\_x = 2.0f;  GLfloat scale\_by\_y = 2.0f;  GLfloat scale\_by\_z = 2.0f;  GLfloat sensitivity = 0.01f; // Used for mouse / camera rotation sensitivity  bool mouseDetected = true;  bool leftClickHold = false;  bool rightClickHold = false;  const int ZOOM\_IN = 'w';  const int ZOOM\_OUT = 's';  const int PAN\_LEFT = 'a';  const int PAN\_RIGHT = 'd';  bool rotate = false;  bool checkMotion = false;  bool checkZoom = false;  // Global vector declarations  glm::vec3 cameraPosition = glm::vec3(0.0f, 0.0f, 0.0f); // Initial camera position, placed 5 units in Z  glm::vec3 CameraUpX = glm::vec3(1.0f, 0.0f, 0.0f); // Temporary x unit vector  glm::vec3 CameraUpY = glm::vec3(0.0f, 1.0f, 0.0f); // Temporary y unit vector  glm::vec3 CameraForwardZ = glm::vec3(0.0f, 0.0f, -1.0f); // Temporary z unit vector  glm::vec3 front; // Temporary z unit vector for mouse  glm::vec3 last\_front; // Temporary z unit vector for mouse  /\* User-defined function prototypes to:  \* initialize the program, set the window size  \* redraw graphics on the window when resized,  \* and render graphics on the screen  \*/  void UResizeWindow(int, int);  void URenderGraphics(void);  void UCreateShader(void);  void UCreateBuffers(void);  void UKeyboard(unsigned char key, GLint x, GLint y);  void UKeyReleased(unsigned char key, GLint x, GLint y);  void initializeMouse(void);  void initializeKeyboard(void);  void UMouseMove(int x, int y);  void OnMouseClicks(int button, int state, int x , int y);  void onMotion(int x, int y);  void UMousePressedMove(int x, int y);  void UKeyReleased(unsigned char key, GLint x, GLint y);  /\* Vertex Shader Program Source Code \*/  const GLchar \* vertexShaderSource = GLSL(330,  layout (location = 0) in vec3 position; // Vertex data from Vertex Attrib Pointer 0  layout (location = 1) in vec3 color; // Color data from Vertex Attrib Pointer 1  out vec3 FragmentPos; // for outgoing color / pixels to fragment shader  out vec3 mobileColor; // variable to transfer color data to the fragment shader  // Global variables for the transform matrices  uniform mat4 model;  uniform mat4 view;  uniform mat4 projection;  void main() {  // transforms vertices to clip coordinates  gl\_Position = projection \* view \* model \* vec4(position, 1.0f); // transforms matrices to clip coordinates  FragmentPos = vec3(model \* vec4(position, 1.0f)); // Gets fragment / pixel position in world space only, exclude view and projection  // references incoming color data  mobileColor = color; // references incoming color data  }  );  /\* Fragment Shader Program Source Code \*/  const GLchar \* fragmentShaderSource = GLSL(330,  in vec3 FragmentPos; // For incoming fragment position  in vec3 mobileColor; // Variable to hold incoming color data from vertex shader  out vec4 gpuColor; // Variable to pass color data to the GPU  // Uniform / Global variables for object color, light color, light position, and camera/view position  uniform vec3 objectColor;  uniform vec3 keyLightColor;  uniform vec3 keyLightPosition;  uniform vec3 fillLightColor;  uniform vec3 fillLightPosition;  uniform vec3 viewPosition;  uniform sampler2D uTexture; // Useful when working with multiple textures  void main() {  /\* Phong lighting model calculations to generate ambient, diffuse, and specular components \*/  // Calculate Ambient lighting  float keyAmbientStrength = 0.4f; // Set ambient or global lighting strength  float fillAmbientStrength = 0.6f; // Set ambient or global lighting strength  vec3 keyAmbient = keyAmbientStrength \* keyLightColor; // Generate ambient light color  vec3 fillAmbient = fillAmbientStrength \* fillLightColor; // Generate ambient light color  vec3 ambient = keyAmbient + fillAmbient;  // Calculate phong result  //vec3 phong = (ambient + diffuse + specular) \* objectColor;  //pyramidColor = texture(uTexture, mobileTextureCoordinate); // Sends texture to the GPU for rendering  //gpuTexture = vec4(phong, 1.0f); // Send lighting results to GPU  vec3 objectColor = mobileColor;  vec3 phong = (ambient) \* objectColor;  //vec3 phong = (diffuse) \* objectColor;  //vec3 phong = (specular) \* objectColor;  //vec3 phong = (ambient + specular) \* objectColor;  //vec3 phong = (ambient + diffuse + specular) \* objectColor;  gpuColor = vec4(mobileColor, 1.0f); //Send lighting results to GPU  //gpuColor = vec4(mobileColor, 1.0); // Sends color data to the GPU for rendering  }  );  /\* Lamp Shader Source Code \*/  const GLchar \* lampVertexShaderSource = GLSL(330,  layout (location = 0) in vec3 position; // VAP position 0 for the vertex position data  // Uniform / Global variables for the transform matrices  uniform mat4 model;  uniform mat4 view;  uniform mat4 projection;  void main() {  gl\_Position = projection \* view \* model \* vec4(position, 1.0f); // Transforms vertices into clip coordinates  }  );  /\* Lamp Fragment Shader Source Code \*/  const GLchar \* lampFragmentShaderSource = GLSL(330,  out vec4 color; // For outgoing lamp color (smaller cube) to the GPU  void main() {  color = vec4(1.0f); // Set color to white (1.0f, 1.0f, 1.0f) with alpha 1.0  }  );  // main function. Entry point to the OpenGL program  int main(int argc, char\* argv[])  {  // initialize all the glut functions  // -- set the display mode  // -- initialize the window size  // -- create the draw window with title  glutInit(&argc, argv);  glutInitDisplayMode(GLUT\_DEPTH | GLUT\_DOUBLE | GLUT\_RGBA);  glutInitWindowSize(WindowWidth, WindowHeight);  glutCreateWindow(WINDOW\_TITLE);  glutReshapeFunc(UResizeWindow);  // initialize the glew functionality  glewExperimental = GL\_TRUE;  if (glewInit() != GLEW\_OK) {  fprintf(stderr, "Failed to initialize GLEW\n");  return -2;  }  // initialize the keyboard and mouse functions  initializeKeyboard();  initializeMouse();  // Create Vertex and Fragment Shader  // Create buffers  UCreateShader();  UCreateBuffers();  // Use the Shader program  glUseProgram(shaderProgram);  glClearColor(0.0f, 0.0f, 0.0f, 1.0f); // Set background color  glutDisplayFunc(URenderGraphics);  glutMainLoop();  // Destroys Buffer objects once used  glDeleteVertexArrays(1, &VAO);  glDeleteVertexArrays(1, &LightVAO);  glDeleteBuffers(1, &VBO);  glDeleteBuffers(1, &EBO);  return 0;  }  // Resizes the window  void UResizeWindow(int w, int h) {  WindowWidth = w;  WindowHeight = h;  glViewport(0, 0, WindowWidth, WindowHeight);  }  // Implements the URenderGraphics function  void URenderGraphics(void) {  glEnable(GL\_DEPTH\_TEST); // Enable z-depth  glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT); // Clears the screen  glBindVertexArray(VAO); // Activates the Vertex Array Object before rendering and transforming them  // Camera Movement Logic  // Zoom In  if (currentKey == ZOOM\_IN) {  //increment scale values  scale\_by\_x += 0.005f;  scale\_by\_y += 0.005f;  scale\_by\_z += 0.005f;  //redisplay  glutPostRedisplay();  }  // Zoom Out  if (currentKey == ZOOM\_OUT) {  //decrement scale values  scale\_by\_x -= 0.005f;  scale\_by\_y -= 0.005f;  scale\_by\_z -= 0.005f;  // control zoom in size  if (scale\_by\_z < 0.2f) {  scale\_by\_x = 0.2f;  scale\_by\_y = 0.2f;  scale\_by\_z = 0.2f;  }  //redisplay  glutPostRedisplay();  }  // Camera moves to the left or right  if ((currentKey == PAN\_LEFT) || (currentKey == PAN\_RIGHT)) {  if (currentKey == PAN\_LEFT) {  mouseXOffset = -0.1; // move to the left  mouseYOffset = 0;  } else { // camera moves to the right  mouseXOffset = 0.1; // move to the right  mouseYOffset = 0;  }  //Applies sensitivity to mouse direction  mouseXOffset \*= sensitivity;  mouseYOffset \*= sensitivity;  // increment yaw  yaw += mouseXOffset;  front.x = 10.0f \* cos(yaw);  front.y = 10.0f \* sin(pitch);  front.z = sin(yaw) \* cos(pitch) \* 10.0f;  //cameraPosition -= (glm::normalize(glm::cross(CameraForwardZ, CameraUpY)) \* cameraSpeed);  }  CameraForwardZ = front; // Replaces camera forward vector with Radians normalized as a unit vector  // Transforms the object  glm::mat4 model;  model = glm::translate(model, glm::vec3(0.0f, 0.0f, 0.0f)); // Place the object at the center of the viewport  model = glm::rotate(model, 45.0f, glm::vec3(0.0f, 1.0f, 0.0f)); // Rotate the object 45 degrees on the X  model = glm::scale(model, glm::vec3(scale\_by\_x,scale\_by\_y,scale\_by\_z)); // Increase the object size by a scale of 2  // Transforms the camera  glm::mat4 view;  view = glm::lookAt(CameraForwardZ, cameraPosition, CameraUpY);  glm::mat4 projection;  if (bUsePerspectiveView) {  // Creates a perspective projection  projection = glm::perspective(45.0f, (GLfloat)WindowWidth / (GLfloat)WindowHeight, 0.1f, 100.0f);  } else {  projection = glm::ortho(45.0f, (GLfloat)WindowWidth / (GLfloat)WindowHeight, 0.1f, 100.0f);  }  // Retrieves and passes transform matrices to the Shader program  GLint modelLoc = glGetUniformLocation(shaderProgram, "model");  GLint viewLoc = glGetUniformLocation(shaderProgram, "view");  GLint projLoc = glGetUniformLocation(shaderProgram, "projection");  glUniformMatrix4fv(modelLoc, 1, GL\_FALSE, glm::value\_ptr(model));  glUniformMatrix4fv(viewLoc, 1, GL\_FALSE, glm::value\_ptr(view));  glUniformMatrix4fv(projLoc, 1, GL\_FALSE, glm::value\_ptr(projection));  glutPostRedisplay();  // Draw the triangles  glDrawElements(GL\_TRIANGLES, 126, GL\_UNSIGNED\_INT, 0);  glBindVertexArray(0); // Deactivate the Vertex Array Object  glutSwapBuffers(); // Flips the back buffer with the front buffer every frame. Similar to GL Flush  }  // Implements the UCreateShaders function  void UCreateShader(void) {  // Vertex shader  GLint vertexShader = glCreateShader(GL\_VERTEX\_SHADER); // Create a Vertex Shader object  glShaderSource(vertexShader, 1, &vertexShaderSource, NULL); // Attaches the Vertex shader to the source code  glCompileShader(vertexShader); // Compiles the Vertex shader  int infologLength = 0;  int charsWritten = 0;  char \*infoLog;  // verify that compilation was successful  // print error message to stderr if failed  glGetShaderiv(vertexShader, GL\_INFO\_LOG\_LENGTH, &infologLength);  if (infologLength > 0)  {  infoLog = (char \*)malloc(infologLength);  glGetShaderInfoLog(vertexShader, infologLength, &charsWritten, infoLog);  fprintf(stderr, "Compile error in vertex shader\n");  fprintf(stderr, "%s\n",infoLog);  free(infoLog);  // Exit with failure.  glDeleteShader(vertexShader); // Don't leak the shader.  return;  }  // Shader compilation is successful if here  // Fragment shader  GLint fragmentShader = glCreateShader(GL\_FRAGMENT\_SHADER); // Create a Fragment Shader object  glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL); // Attaches the Fragment shader to the source code  glCompileShader(fragmentShader); // Compiles the Fragment shader  // verify that compilation was successful  // print error message to stderr if failed  glGetShaderiv(fragmentShader, GL\_INFO\_LOG\_LENGTH, &infologLength);  if (infologLength > 0)  {  infoLog = (char \*)malloc(infologLength);  glGetShaderInfoLog(fragmentShader, infologLength, &charsWritten, infoLog);  fprintf(stderr, "Compile error in fragment shader\n");  fprintf(stderr, "%s\n",infoLog);  free(infoLog);  // Exit with failure.  glDeleteShader(vertexShader); // Don't leak the shader.  glDeleteShader(fragmentShader); // Don't leak the shader.  return;  }  // Shader compilation is successful if here  // Shader program  shaderProgram = glCreateProgram(); // Creates the Shader program and returns an id  glAttachShader(shaderProgram, vertexShader); // Attach Vertex shader to the Shader program  glAttachShader(shaderProgram, fragmentShader); // Attach Fragment shader to the Shader program  glLinkProgram(shaderProgram); // Links the shader program  // Delete the Vertex and Fragment shaders once linked  glDeleteShader(vertexShader);  glDeleteShader(fragmentShader);  // Lamp Vertex shader  GLint lampVertexShader = glCreateShader(GL\_VERTEX\_SHADER); // Creates the Vertex shader  glShaderSource(lampVertexShader, 1, &lampVertexShaderSource, NULL); // Attaches the fragment shader to the source code  glCompileShader(lampVertexShader); // Compiles the Fragment shader  glGetShaderiv(lampVertexShader, GL\_INFO\_LOG\_LENGTH, &infologLength);  if (infologLength > 0)  {  infoLog = (char \*)malloc(infologLength);  glGetShaderInfoLog(lampVertexShader, infologLength, &charsWritten, infoLog);  fprintf(stderr, "Compile error in lamp vertex shader\n");  fprintf(stderr, "%s\n",infoLog);  free(infoLog);  // Exit with failure.  glDeleteShader(vertexShader); // Don't leak the shader.  glDeleteShader(fragmentShader); // Don't leak the shader.  glDeleteShader(lampVertexShader); // Don't leak the shader.  return;  }  // Shader compilation is successful if here  // Lamp Fragment shader  GLint lampFragmentShader = glCreateShader(GL\_FRAGMENT\_SHADER); // Creates the Fragment shader  glShaderSource(lampFragmentShader, 1, &lampFragmentShaderSource, NULL); // Attaches the fragment shader to the source code  glCompileShader(lampFragmentShader); // Compiles the Fragment shader  glGetShaderiv(lampFragmentShader, GL\_INFO\_LOG\_LENGTH, &infologLength);  if (infologLength > 0)  {  infoLog = (char \*)malloc(infologLength);  glGetShaderInfoLog(lampFragmentShader, infologLength, &charsWritten, infoLog);  fprintf(stderr, "Compile error in lamp fragment shader\n");  fprintf(stderr, "%s\n",infoLog);  free(infoLog);  // Exit with failure.  glDeleteShader(vertexShader); // Don't leak the shader.  glDeleteShader(fragmentShader); // Don't leak the shader.  glDeleteShader(lampVertexShader); // Don't leak the shader.  glDeleteShader(lampFragmentShader); // Don't leak the shader.  return;  }  // Shader compilation is successful if here  // Lamp Shader Program  lampShaderProgram = glCreateProgram(); // Creates the Shader program and returns an id  glAttachShader(lampShaderProgram, lampVertexShader); // Attach Vertex shader to the Shader program  glAttachShader(lampShaderProgram, lampFragmentShader); // Attach Fragment shader to the Shader program  glLinkProgram(lampShaderProgram); // Link Vertex and Fragment shaders to Shader program  // Delete the lamp shaders once linked  glDeleteShader(lampVertexShader);  glDeleteShader(lampFragmentShader);  }  // Creates the Buffer and Array Objects  void UCreateBuffers() {  // Position and Color data  GLfloat vertices[] = {  // Vertex Positions // Colors  // Top of Table  -1.0f, 0.0f, 0.0f, Blue, // 0  -1.0f, 0.2f, 0.0f, Red, // 1  -1.0f, 0.2f, -1.0f, Red, // 2  -1.0f, 0.0f, -1.0f, Blue, // 3  1.0f, 0.2f, -1.0f, Red, // 4  1.0f, 0.0f, -1.0f, Blue, // 5  1.0f, 0.2f, 0.0f, Red, // 6  1.0f, 0.0f, 0.0f, Blue, // 7  // Leg 1  -0.8f, 0.0f, 0.0f, Blue, // 8  -0.8f, 0.0f, -0.2f, Blue, // 9  -1.0f, 0.0f, -0.2f, Green, // 10  -1.0f, -1.0f, 0.0f, Green, // 11  -0.8f, -1.0f, 0.0f, Yellow, // 12  -0.8f, -1.0f, -0.2f, Yellow, // 13  -1.0f, -1.0f, -0.2f, Magenta, // 14  // Leg 2  -1.0f, 0.0f, -0.8f, Green, // 15  -0.8f, 0.0f, -0.8f, Green, // 16  -0.8f, 0.0f, -1.0f, Yellow, // 17  -1.0f, -1.0f, -0.8f, Yellow, // 18  -0.8f, -1.0f, -0.8f, Magenta, // 19  -0.8f, -1.0f, -1.0f, Magenta, // 20  -1.0f, -1.0f, -1.0f, Red, // 21  // Leg 3  1.0f, 0.0f, -0.2f, Red, // 22  0.8f, 0.0f, -0.2f, Blue, // 23  0.8f, 0.0f, 0.0f, Blue, // 24  1.0f, -1.0f, -0.2f, Green, // 25  0.8f, -1.0f, -0.2f, Green, // 26  0.8f, -1.0f, 0.0f, Yellow, // 27  1.0f, -1.0f, 0.0f, Yellow, // 28  // Leg 4  0.8f, 0.0f, -1.0f, Blue, // 29  0.8f, 0.0f, -0.8f, Red, // 30  1.0f, 0.0f, -0.8f, Red, // 31  0.8f, -1.0f, -1.0f, Green, // 32  0.8f, -1.0f, -0.8f, Green, // 33  1.0f, -1.0f, -0.8f, Yellow, // 34  1.0f, -1.0f, -1.0f, Yellow // 35  };  // Index data to share position data  GLuint indices[] = {  // Table  1, 2, 4,  1, 6, 4,  0, 1, 2,  0, 3, 2,  3, 2, 4,  3, 5, 4,  4, 5, 7,  4, 6, 7,  0, 1, 6,  0, 7, 6,  // Leg 1  0, 8, 12,  12, 11, 0,  8, 12, 13,  13, 9, 8,  9, 10, 14,  14, 13, 9,  10, 14, 11,  11, 0, 10,  // Leg 2  3, 15, 18,  18, 21, 3,  15, 16, 19,  19, 18, 15,  16, 17, 20,  20, 19, 16,  3, 17, 20,  20, 21, 3,  // Leg 3  22, 23, 25,  23, 25, 26,  23, 24, 26,  24, 26, 27,  24, 7, 27,  27, 7, 28,  7, 22, 25,  7, 25, 28,  // Leg 4  5, 29, 35,  29, 32, 35,  29, 32, 33,  29, 30, 33,  30, 33, 34,  30, 31, 34,  31, 34, 35,  31, 5, 35  };  // Generate buffer ids  glGenVertexArrays(1, &VAO);  glGenBuffers(1, &VBO);  glGenBuffers(2, &EBO);  // Activate the Vertex Array Object before binding and setting and VBOs and Vertex Attribute Pointers  glBindVertexArray(VAO);  // Activate the VBO  glBindBuffer(GL\_ARRAY\_BUFFER, VBO);  glBufferData(GL\_ARRAY\_BUFFER, sizeof(vertices), vertices, GL\_STATIC\_DRAW); // Copy vertices to VBO  // Activate the Element Buffer Object / Indices  glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, EBO);  glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, sizeof(indices), indices, GL\_STATIC\_DRAW); // Copy indices to EBO  // Set attribute pointer 0 to hold Position data  glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 6 \* sizeof(GLfloat), (GLvoid\*)0);  glEnableVertexAttribArray(0); // Enables vertex attribute  // Set attribute pointer 1 to hold Color data  glVertexAttribPointer(1, 3, GL\_FLOAT, GL\_FALSE, 6 \* sizeof(GLfloat), (GLvoid\*)(3 \* sizeof(GLfloat)));  glEnableVertexAttribArray(1); // Enables vertex attribute  int sof = 6 \* sizeof(GLfloat);  int soui = 3 \* sizeof(GLuint);  int vsize = sizeof(vertices);  int isize = sizeof(indices);  vsize = vsize/sof;  isize = isize/soui;  glBindVertexArray(0); // Deactivates the VAO which is good practice  }  // Mouse initializations and callbacks  // occur here  void initializeMouse() {  glutPassiveMotionFunc(UMouseMove); // Detects mouse movement without any mouse buttons pushed  glutMotionFunc(onMotion); // Detects mouse movement while a mouse button is pushed  glutMouseFunc(OnMouseClicks); // Detects mouse click  }  // Keyboard initializations and callbacks  // occur here  void initializeKeyboard() {  glutKeyboardFunc(UKeyboard); // Detects key press  glutKeyboardUpFunc(UKeyReleased); // Detects key release  }  // Detects mouse movement without any mouse buttons pushed  void UMouseMove(int x, int y){  front.x = 10.0f \* cos(yaw);  front.y = 10.0f \* sin(pitch);  front.z = sin(yaw) \* cos(pitch) \* 10.0f;  }  // Implements the UKeyboard function  void UKeyboard(unsigned char key, GLint x, GLint y) {  switch(key) {  case ZOOM\_IN:  case ZOOM\_OUT:  case PAN\_LEFT:  case PAN\_RIGHT:  currentKey = key;  break;  default:  currentKey = '0';  break;  }  }  // Implements the UKeyReleased function  void UKeyReleased(unsigned char key, GLint x, GLint y) {  currentKey = '0';  }  // Detects mouse movement while a mouse button is pushed  void onMotion(int curr\_x, int curr\_y) {  //if left alt and mouse down are set  if (checkMotion) {  // gets the direction the mouse was moved  mouseXOffset = curr\_x - lastMouseX;  mouseYOffset = lastMouseY - curr\_y;  // updates with new mouse coordinates  lastMouseX = curr\_x;  lastMouseY = curr\_y;  // applies sensitivity to mouse direction  mouseXOffset \*= sensitivity;  mouseYOffset \*= sensitivity;  // get the direction of the mouse  // if there is changes in yaw, then it is moving along X  if ((yaw != yaw + mouseXOffset) && (pitch == pitch + mouseYOffset)) {  // increment yaw  yaw += mouseXOffset;  //else movement in y  } else if ((pitch != pitch+mouseYOffset) && (yaw == yaw+mouseXOffset)) {  // increment y to move vertical  pitch += mouseYOffset;  }  front.x = 10.0f \* cos(yaw);  front.y = 10.0f \* sin(pitch);  front.z = sin(yaw) \* cos(pitch) \* 10.0f;  }  // check if user is zooming, alt, right mouse button and down  if (checkZoom) {  // determine the direction of the movement, either up or down  if (lastMouseY < curr\_y) {  // mouse moving up on y  // decrement scale values, zoom in  scale\_by\_x -= 0.1f;  scale\_by\_y -= 0.1f;  scale\_by\_z -= 0.1f;  // control zoom in size  if (scale\_by\_z < 0.2f) {  scale\_by\_x = 0.2f;  scale\_by\_y = 0.2f;  scale\_by\_z = 0.2f;  }  // redisplay  glutPostRedisplay();  } else { // zoom in  // mouse down up on y  // increment scale values  scale\_by\_x += 0.1f;  scale\_by\_y += 0.1f;  scale\_by\_z += 0.1f;  glutPostRedisplay();  }  // update x and y  lastMouseY = curr\_y;  lastMouseX = curr\_x;  }  }  // Detects mouse clicks  void OnMouseClicks(int button, int state, int x, int y) {  modifierKey = glutGetModifiers(); // checks for modifier keys like alt, shif and ctrl  checkMotion = false; //set checkMotion to false  //check if button is left, and mod is alt and state is down, all should be true  if ((button == GLUT\_LEFT\_BUTTON) && (modifierKey == GLUT\_ACTIVE\_ALT) && (state == GLUT\_DOWN)) {  checkMotion = true; // set motion true  checkZoom = false; // set zoom false  } else if ((button == GLUT\_RIGHT\_BUTTON) && (modifierKey == GLUT\_ACTIVE\_ALT) && (state == GLUT\_DOWN)) {  checkMotion = false; // set motion false  checkZoom = true; // set zoom true  }  } |

# Artifact with Enhancements

|  |
| --- |
| /\*  \* Name: William Moore  \* School: Southern New Hampshire University  \* Class: CS-499-Q1527 Computer Science Capstone 20EW1  \* Assignment: 7-1 Final Submission with Enhancements  \*/  // Header Inclusions  #include <iostream> // Includes C++ i/o stream  #include <GL/glew.h> // Includes glew header  #include <GL/freeglut.h> // Include the freeglut header file  // GLM Math Header inclusions  #include <glm/glm.hpp>  #include <glm/gtc/matrix\_transform.hpp>  #include <glm/gtc/type\_ptr.hpp>  // rgb color definitions for the basic colors  #define Red 1.0f, 0.0f, 0.0f  #define Green 0.0f, 1.0f, 0.0f  #define Blue 0.0f, 0.0f, 1.0f  #define Yellow 1.0f, 1.0f, 0.0f  #define Cyan 0.0f, 1.0f, 1.0f  #define Magenta 1.0f, 0.0f, 1.0f  #define White 1.0f, 1.0f, 1.0f  #define Black 0.0f, 0.0f, 0.0f  #define DullGreen 0.0f, 0.5f, 0.4f  #define Grey 0.5f, 0.5f, 0.5f  using namespace std; // use standard namespace  #define WINDOW\_TITLE "7-1 Final Project with Enhancements (William Moore)" // Macro for window title  // shader source macro  #ifndef GLSL  #define GLSL(Version, Source) "#version " #Version "\n" #Source  #endif  // Variable declarations for shader, window size initialization, buffer and array objects  GLint shaderProgram;  GLint lampShaderProgram;  GLint WindowWidth = 800;  GLint WindowHeight = 600;  GLuint VBO, VAO, LightVAO, EBO, texture;  GLfloat cameraSpeed = 0.0005f; // Movement speed per frame  GLchar currentKey; // Will store key pressed  int modifierKey;  bool bUsePerspectiveView = true;  // pyramid and light color, 0.6f, 0.5f, 0.75f  glm::vec3 objectColor(1.0f, 1.0f, 1.0f);  // Key Light position and scale  glm::vec3 keyLightPosition(-0.8f, 0.0f, 15.0f); // left side of pyramid in foreground  glm::vec3 keyLightScale(0.1f);  glm::vec3 keyLightColor(Green);  // Fill Light position and scale  glm::vec3 fillLightPosition(0.5f, 0.5f, -5.0f); // right back side of the pyramid  glm::vec3 fillLightScale(0.3f);  glm::vec3 fillLightColor(Red);  GLfloat lastMouseX = 400; // Locks mouse cursor at the center of the screen  GLfloat lastMouseY = 300;  GLfloat mouseXOffset; // Mouse offset, yaw and pitch variables  GLfloat mouseYOffset;  GLfloat yaw = 0.0f;  GLfloat pitch = 0.0f;  GLfloat scale\_by\_x=2.0f;  GLfloat scale\_by\_y=2.0f;  GLfloat scale\_by\_z=2.0f;  GLfloat sensitivity = 0.01f; // Used for mouse / camera rotation sensitivity  bool mouseDetected = true;  bool leftClickHold = false;  bool rightClickHold = false;  const int ZOOM\_IN = 'w';  const int ZOOM\_OUT = 's';  const int PAN\_LEFT = 'a';  const int PAN\_RIGHT = 'd';  bool rotate = false;  bool checkMotion = false;  bool checkZoom = false;  // Global vector declarations  glm::vec3 cameraPosition = glm::vec3(0.0f, 0.0f, 0.0f); // Initial camera position, placed 5 units in Z  glm::vec3 CameraUpX = glm::vec3(1.0f, 0.0f, 0.0f); // Temporary x unit vector  glm::vec3 CameraUpY = glm::vec3(0.0f, 1.0f, 0.0f); // Temporary y unit vector  glm::vec3 CameraForwardZ = glm::vec3(0.0f, 0.0f, -1.0f); // Temporary z unit vector  glm::vec3 front; // Temporary z unit vector for mouse  glm::vec3 last\_front; // Temporary z unit vector for mouse  /\* User-defined function prototypes to:  \* initialize the program, set the window size  \* redraw graphics on the window when resized,  \* and render graphics on the screen  \*/  void UResizeWindow(int, int);  void URenderGraphics(void);  void UCreateShader(void);  void UCreateBuffers(void);  void UKeyboard(unsigned char key, GLint x, GLint y);  void UKeyReleased(unsigned char key, GLint x, GLint y);  void initializeMouse(void);  void initializeKeyboard(void);  //void UMouseClick(int button, int state, int x, int y);  void UMouseMove(int x, int y);  void OnMouseClicks(int button, int state, int x , int y);  void onMotion(int x, int y);  void UMousePressedMove(int x, int y);  void UKeyReleased(unsigned char key, GLint x, GLint y);  /\* Vertex Shader Program Source Code \*/  const GLchar \* vertexShaderSource = GLSL(330,  layout (location = 0) in vec3 position; // Vertex data from Vertex Attrib Pointer 0  layout (location = 1) in vec3 color; // Color data from Vertex Attrib Pointer 1  out vec3 FragmentPos; // for outgoing color / pixels to fragment shader  out vec3 mobileColor; // variable to transfer color data to the fragment shader  // Global variables for the transform matrices  uniform mat4 model;  uniform mat4 view;  uniform mat4 projection;  void main() {  // transforms vertices to clip coordinates  gl\_Position = projection \* view \* model \* vec4(position, 1.0f); // transforms matrices to clip coordinates  FragmentPos = vec3(model \* vec4(position, 1.0f)); // Gets fragment / pixel position in world space only, exclude view and projection  // references incoming color data  mobileColor = color; // references incoming color data  }  );  /\* Fragment Shader Program Source Code \*/  const GLchar \* fragmentShaderSource = GLSL(330,  in vec3 FragmentPos; // For incoming fragment position  in vec3 mobileColor; // Variable to hold incoming color data from vertex shader  out vec4 gpuColor; // Variable to pass color data to the GPU  // Uniform / Global variables for object color, light color, light position, and camera/view position  uniform vec3 objectColor;  uniform vec3 keyLightColor;  uniform vec3 keyLightPosition;  uniform vec3 fillLightColor;  uniform vec3 fillLightPosition;  uniform vec3 viewPosition;  uniform sampler2D uTexture; // Useful when working with multiple textures  void main() {  /\* Phong lighting model calculations to generate ambient, diffuse, and specular components \*/  // Calculate Ambient lighting  float keyAmbientStrength = 0.6f; // Set key ambient or global lighting strength  float fillAmbientStrength = 0.4f; // Set fill ambient or global lighting strength  vec3 keyAmbient = keyAmbientStrength \* keyLightColor; // Generate key ambient light color  vec3 fillAmbient = fillAmbientStrength \* fillLightColor; // Generate fill ambient light color  vec3 ambient = keyAmbient + fillAmbient;  // Calculate Diffuse lighting  vec3 norm = normalize(Normal); // Normalize vectors to 1 unit  vec3 keyLightDirection = normalize(keyLightPosition - FragmentPos); // Calculate distance (light direction) between light source and fragments/pixels  float impact = max(dot(norm, keyLightDirection), 0.0); // Calculate diffuse impact by generating dot product of normal and light  vec3 keyDiffuse = impact \* keyLightColor; // Generate diffuse light color  vec3 fillLightDirection = normalize(fillLightPosition - FragmentPos); // Calculate distance (light direction) between light source and fragments/pixels  impact = max(dot(norm, fillLightDirection), 0.0); // Calculate diffuse impact by generating dot product of normal and light  vec3 fillDiffuse = impact \* fillLightColor; // Generate diffuse light color  vec3 diffuse = keyDiffuse + fillDiffuse;  vec3 viewDir = normalize(viewPosition - FragmentPos); // Calculate view direction  vec3 keyReflectDir = reflect(-keyLightDirection, norm); // Calculate key reflection vector  vec3 fillReflectDir = reflect(-fillLightDirection, norm); // Calculate fill reflection vector  vec3 reflectDir = keyReflectDir + fillReflectDir;  // Calculate Specular lighting  float keySpecularIntensity = 0.6f; // Set specular key light strength  float keyHighlightSize = 1.0f; // Set specular key highlight size  float fillSpecularIntensity = 0.2f; // Set specular fill light strength  float fillHighlightSize = 0.6f; // Set specular fill highlight size  // Calculate specular component  float keySpecularComponent = pow(max(dot(viewDir, reflectDir), 0.0), keyHighlightSize);  float fillSpecularComponent = pow(max(dot(viewDir, reflectDir), 0.0), fillHighlightSize);  vec3 keySpecular = keySpecularIntensity \* keySpecularComponent \* keyLightColor;  vec3 fillSpecular = fillSpecularIntensity \* fillSpecularComponent \* fillLightColor;  vec3 specular = keySpecular + fillSpecular;  // Calculate phong result  vec3 objectColor = mobileColor;  vec3 phong = (ambient + diffuse + specular) \* objectColor;  gpuColor = vec4(phong, 1.0f); //Send lighting results to GPU  }  );  /\* Lamp Shader Source Code \*/  const GLchar \* lampVertexShaderSource = GLSL(330,  layout (location = 0) in vec3 position; // VAP position 0 for the vertex position data  // Uniform / Global variables for the transform matrices  uniform mat4 model;  uniform mat4 view;  uniform mat4 projection;  void main() {  gl\_Position = projection \* view \* model \* vec4(position, 1.0f); // Transforms vertices into clip coordinates  }  );  /\* Lamp Fragment Shader Source Code \*/  const GLchar \* lampFragmentShaderSource = GLSL(330,  out vec4 color; // For outgoing lamp color (smaller cube) to the GPU  void main() {  color = vec4(1.0f); // Set color to white (1.0f, 1.0f, 1.0f) with alpha 1.0  }  );  // main function. Entry point to the OpenGL program  int main(int argc, char\* argv[])  {  glutInit(&argc, argv);  glutInitDisplayMode(GLUT\_DEPTH | GLUT\_DOUBLE | GLUT\_RGBA);  glutInitWindowSize(WindowWidth, WindowHeight);  glutCreateWindow(WINDOW\_TITLE);  glutReshapeFunc(UResizeWindow);  glewExperimental = GL\_TRUE;  if (glewInit() != GLEW\_OK) {  fprintf(stderr, "Failed to initialize GLEW\n");  return -2;  }  initializeKeyboard();  initializeMouse();  // Create Vertex and Fragment Shader  // Create buffers  UCreateShader();  UCreateBuffers();  // Use the Shader program  glUseProgram(shaderProgram);  glClearColor(0.0f, 0.0f, 0.0f, 1.0f); // Set background color  glutDisplayFunc(URenderGraphics);  glutMainLoop();  // Destroys Buffer objects once used  glDeleteVertexArrays(1, &VAO);  glDeleteVertexArrays(1, &LightVAO);  glDeleteBuffers(1, &VBO);  glDeleteBuffers(1, &EBO);  return 0;  }  // Resizes the window  void UResizeWindow(int w, int h) {  WindowWidth = w;  WindowHeight = h;  glViewport(0, 0, WindowWidth, WindowHeight);  }  // Implements the URenderGraphics function  void URenderGraphics(void) {  glEnable(GL\_DEPTH\_TEST); // Enable z-depth  glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT); // Clears the screen  glBindVertexArray(VAO); // Activates the Vertex Array Object before rendering and transforming them  GLint objectColorLoc, viewPositionLoc;  GLint keyLightColorLoc, keyLightPositionLoc, fillLightColorLoc, fillLightPositionLoc;  // Camera Movement Logic  // Zoom In  if (currentKey == ZOOM\_IN) {  //increment scale values  scale\_by\_x += 0.005f;  scale\_by\_y += 0.005f;  scale\_by\_z += 0.005f;  //redisplay  glutPostRedisplay();  }  // Zoom Out  if (currentKey == ZOOM\_OUT) {  //decrement scale values  scale\_by\_x -= 0.005f;  scale\_by\_y -= 0.005f;  scale\_by\_z -= 0.005f;  // control zoom in size  if (scale\_by\_z < 0.2f) {  scale\_by\_x = 0.2f;  scale\_by\_y = 0.2f;  scale\_by\_z = 0.2f;  }  //redisplay  glutPostRedisplay();  }  // Camera moves to the left or right  if ((currentKey == PAN\_LEFT) || (currentKey == PAN\_RIGHT)) {  if (currentKey == PAN\_LEFT) {  mouseXOffset = -0.1; // move to the left  mouseYOffset = 0;  } else { // camera moves to the right  mouseXOffset = 0.1; // move to the right  mouseYOffset = 0;  }  //Applies sensitivity to mouse direction  mouseXOffset \*= sensitivity;  mouseYOffset \*= sensitivity;  // increment yaw  yaw += mouseXOffset;  front.x = 10.0f \* cos(yaw);  front.y = 10.0f \* sin(pitch);  front.z = sin(yaw) \* cos(pitch) \* 10.0f;  //cameraPosition -= (glm::normalize(glm::cross(CameraForwardZ, CameraUpY)) \* cameraSpeed);  }  CameraForwardZ = front; // Replaces camera forward vector with Radians normalized as a unit vector  /\* Use the pyramid Shader and activate the pyramid Vertex Array Object for rendering and transforming \*/  glUseProgram(shaderProgram);  glBindVertexArray(VAO);  // Transforms the object  glm::mat4 model;  model = glm::translate(model, glm::vec3(0.0f, 0.0f, 0.0f)); // Place the object at the center of the viewport  model = glm::rotate(model, 45.0f, glm::vec3(0.0f, 1.0f, 0.0f)); // Rotate the object 45 degrees on the X  model = glm::scale(model, glm::vec3(scale\_by\_x,scale\_by\_y,scale\_by\_z)); // Increase the object size by a scale of 2  // Transforms the camera  glm::mat4 view;  view = glm::lookAt(CameraForwardZ, cameraPosition, CameraUpY);  glm::mat4 projection;  if (bUsePerspectiveView) {  // Creates a perspective projection  projection = glm::perspective(45.0f, (GLfloat)WindowWidth / (GLfloat)WindowHeight, 0.1f, 100.0f);  } else {  projection = glm::ortho(45.0f, (GLfloat)WindowWidth / (GLfloat)WindowHeight, 0.1f, 100.0f);  }  // Retrieves and passes transform matrices to the Shader program  GLint modelLoc = glGetUniformLocation(shaderProgram, "model");  GLint viewLoc = glGetUniformLocation(shaderProgram, "view");  GLint projLoc = glGetUniformLocation(shaderProgram, "projection");  glUniformMatrix4fv(modelLoc, 1, GL\_FALSE, glm::value\_ptr(model));  glUniformMatrix4fv(viewLoc, 1, GL\_FALSE, glm::value\_ptr(view));  glUniformMatrix4fv(projLoc, 1, GL\_FALSE, glm::value\_ptr(projection));  // Reference matrix uniforms from the pyramid Shader program for the pyramid color,  // light color, light position, and camera position  objectColorLoc = glGetUniformLocation(shaderProgram, "objectColor");  keyLightColorLoc = glGetUniformLocation(shaderProgram, "keyLightColor");  keyLightPositionLoc = glGetUniformLocation(shaderProgram, "keyLightPos");  fillLightColorLoc = glGetUniformLocation(shaderProgram, "fillLightColor");  fillLightPositionLoc = glGetUniformLocation(shaderProgram, "fillLightPos");  viewPositionLoc = glGetUniformLocation(shaderProgram, "viewPosition");  // Pass color, light, and camera data to the pyramid Shader program's corresponding uniforms  glUniform3f(objectColorLoc, objectColor.r, objectColor.g, objectColor.b);  glUniform3f(keyLightColorLoc, keyLightColor.r, keyLightColor.g, keyLightColor.b);  glUniform3f(keyLightPositionLoc, keyLightPosition.x, keyLightPosition.y, keyLightPosition.z);  glUniform3f(fillLightColorLoc, fillLightColor.r, fillLightColor.g, fillLightColor.b);  glUniform3f(fillLightPositionLoc, fillLightPosition.x, fillLightPosition.y, fillLightPosition.z);  glUniform3f(viewPositionLoc, cameraPosition.x, cameraPosition.y, cameraPosition.z);  /\* Use the Lamp Shader and activate the Lamp Vertex Array Object for rendering and transforming \*/  glUseProgram(lampShaderProgram);  glBindVertexArray(LightVAO);  glm::vec3 lightPosition, lightScale;  lightPosition = keyLightPosition + fillLightPosition;  lightScale = keyLightScale + fillLightScale;  // Transform the smaller cube used as a visual que for the light source  model = glm::translate(model, lightPosition);  model = glm::scale(model, lightScale);  // Reference matrix uniforms from the Lamp Shader program  modelLoc = glGetUniformLocation(lampShaderProgram, "model");  viewLoc = glGetUniformLocation(lampShaderProgram, "view");  projLoc = glGetUniformLocation(lampShaderProgram, "projection");  // Pass matrix data to the Lamp Shader program's matrix uniform  glUniformMatrix4fv(modelLoc, 1, GL\_FALSE, glm::value\_ptr(model));  glUniformMatrix4fv(viewLoc, 1, GL\_FALSE, glm::value\_ptr(view));  glUniformMatrix4fv(projLoc, 1, GL\_FALSE, glm::value\_ptr(projection));  // Redraw the display  glutPostRedisplay();  // Draw the triangles  glDrawElements(GL\_TRIANGLES, 126, GL\_UNSIGNED\_INT, 0);  glBindVertexArray(0); // Deactivate the Vertex Array Object  /\* Use the Lamp Shader and activate the Lamp Vertex Array Object for rendering and transforming \*/  glUseProgram(lampShaderProgram);  glBindVertexArray(LightVAO);  //glm::vec3 lightPosition, lightScale;  lightPosition = keyLightPosition + fillLightPosition;  lightScale = keyLightScale + fillLightScale;  // Transform the smaller cube used as a visual que for the light source  model = glm::translate(model, lightPosition);  model = glm::scale(model, lightScale);  // Reference matrix uniforms from the Lamp Shader program  modelLoc = glGetUniformLocation(lampShaderProgram, "model");  viewLoc = glGetUniformLocation(lampShaderProgram, "view");  projLoc = glGetUniformLocation(lampShaderProgram, "projection");  // Pass matrix data to the Lamp Shader program's matrix uniforms  glUniformMatrix4fv(modelLoc, 1, GL\_FALSE, glm::value\_ptr(model));  glUniformMatrix4fv(viewLoc, 1, GL\_FALSE, glm::value\_ptr(view));  glUniformMatrix4fv(projLoc, 1, GL\_FALSE, glm::value\_ptr(projection));  // Draw the triangles  glDrawElements(GL\_TRIANGLES, 126, GL\_UNSIGNED\_INT, 0);  // Redraw the display  glutPostRedisplay();  glBindVertexArray(0); // Deactivate the Lamp Vertex Array Object  glutSwapBuffers(); // Flips the back buffer with the front buffer every frame. Similar to GL Flush  }  // Implements the UCreateShaders function  void UCreateShader(void) {  // Vertex shader  GLint vertexShader = glCreateShader(GL\_VERTEX\_SHADER); // Create a Vertex Shader object  glShaderSource(vertexShader, 1, &vertexShaderSource, NULL); // Attaches the Vertex shader to the source code  glCompileShader(vertexShader); // Compiles the Vertex shader  int infologLength = 0;  int charsWritten = 0;  char \*infoLog;  // verify that compilation was successful  // print error message to stderr if failed  glGetShaderiv(vertexShader, GL\_INFO\_LOG\_LENGTH, &infologLength);  if (infologLength > 0)  {  infoLog = (char \*)malloc(infologLength);  glGetShaderInfoLog(vertexShader, infologLength, &charsWritten, infoLog);  fprintf(stderr, "Compile error in vertex shader\n");  fprintf(stderr, "%s\n",infoLog);  free(infoLog);  // Exit with failure.  glDeleteShader(vertexShader); // Don't leak the shader.  return;  }  // Shader compilation is successful if here  // Fragment shader  GLint fragmentShader = glCreateShader(GL\_FRAGMENT\_SHADER); // Create a Fragment Shader object  glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL); // Attaches the Fragment shader to the source code  glCompileShader(fragmentShader); // Compiles the Fragment shader  // verify that compilation was successful  // print error message to stderr if failed  glGetShaderiv(fragmentShader, GL\_INFO\_LOG\_LENGTH, &infologLength);  if (infologLength > 0)  {  infoLog = (char \*)malloc(infologLength);  glGetShaderInfoLog(fragmentShader, infologLength, &charsWritten, infoLog);  fprintf(stderr, "Compile error in fragment shader\n");  fprintf(stderr, "%s\n",infoLog);  free(infoLog);  // Exit with failure.  glDeleteShader(vertexShader); // Don't leak the shader.  glDeleteShader(fragmentShader); // Don't leak the shader.  return;  }  // Shader compilation is successful if here  // Shader program  shaderProgram = glCreateProgram(); // Creates the Shader program and returns an id  glAttachShader(shaderProgram, vertexShader); // Attach Vertex shader to the Shader program  glAttachShader(shaderProgram, fragmentShader); // Attach Fragment shader to the Shader program  glLinkProgram(shaderProgram); // Links the shader program  // Delete the Vertex and Fragment shaders once linked  glDeleteShader(vertexShader);  glDeleteShader(fragmentShader);  // Lamp Vertex shader  GLint lampVertexShader = glCreateShader(GL\_VERTEX\_SHADER); // Creates the Vertex shader  glShaderSource(lampVertexShader, 1, &lampVertexShaderSource, NULL); // Attaches the fragment shader to the source code  glCompileShader(lampVertexShader); // Compiles the Fragment shader  glGetShaderiv(lampVertexShader, GL\_INFO\_LOG\_LENGTH, &infologLength);  if (infologLength > 0)  {  infoLog = (char \*)malloc(infologLength);  glGetShaderInfoLog(lampVertexShader, infologLength, &charsWritten, infoLog);  fprintf(stderr, "Compile error in lamp vertex shader\n");  fprintf(stderr, "%s\n",infoLog);  free(infoLog);  // Exit with failure.  glDeleteShader(vertexShader); // Don't leak the shader.  glDeleteShader(fragmentShader); // Don't leak the shader.  glDeleteShader(lampVertexShader); // Don't leak the shader.  return;  }  // Shader compilation is successful if here  // Lamp Fragment shader  GLint lampFragmentShader = glCreateShader(GL\_FRAGMENT\_SHADER); // Creates the Fragment shader  glShaderSource(lampFragmentShader, 1, &lampFragmentShaderSource, NULL); // Attaches the fragment shader to the source code  glCompileShader(lampFragmentShader); // Compiles the Fragment shader  glGetShaderiv(lampFragmentShader, GL\_INFO\_LOG\_LENGTH, &infologLength);  if (infologLength > 0)  {  infoLog = (char \*)malloc(infologLength);  glGetShaderInfoLog(lampFragmentShader, infologLength, &charsWritten, infoLog);  fprintf(stderr, "Compile error in lamp fragment shader\n");  fprintf(stderr, "%s\n",infoLog);  free(infoLog);  // Exit with failure.  glDeleteShader(vertexShader); // Don't leak the shader.  glDeleteShader(fragmentShader); // Don't leak the shader.  glDeleteShader(lampVertexShader); // Don't leak the shader.  glDeleteShader(lampFragmentShader); // Don't leak the shader.  return;  }  // Shader compilation is successful if here  // Lamp Shader Program  lampShaderProgram = glCreateProgram(); // Creates the Shader program and returns an id  glAttachShader(lampShaderProgram, lampVertexShader); // Attach Vertex shader to the Shader program  glAttachShader(lampShaderProgram, lampFragmentShader); // Attach Fragment shader to the Shader program  glLinkProgram(lampShaderProgram); // Link Vertex and Fragment shaders to Shader program  // Delete the lamp shaders once linked  glDeleteShader(lampVertexShader);  glDeleteShader(lampFragmentShader);  }  // Creates the Buffer and Array Objects  void UCreateBuffers() {  // Position and Color data  GLfloat vertices[] = {  // Vertex Positions // Colors  // Top of Table  -1.0f, 0.0f, 0.0f, Blue, // 0  -1.0f, 0.2f, 0.0f, Red, // 1  -1.0f, 0.2f, -1.0f, Red, // 2  -1.0f, 0.0f, -1.0f, Blue, // 3  1.0f, 0.2f, -1.0f, Red, // 4  1.0f, 0.0f, -1.0f, Blue, // 5  1.0f, 0.2f, 0.0f, Red, // 6  1.0f, 0.0f, 0.0f, Blue, // 7  // Leg 1  -0.8f, 0.0f, 0.0f, Blue, // 8  -0.8f, 0.0f, -0.2f, Blue, // 9  -1.0f, 0.0f, -0.2f, Green, // 10  -1.0f, -1.0f, 0.0f, Green, // 11  -0.8f, -1.0f, 0.0f, Yellow, // 12  -0.8f, -1.0f, -0.2f, Yellow, // 13  -1.0f, -1.0f, -0.2f, Magenta, // 14  // Leg 2  -1.0f, 0.0f, -0.8f, Green, // 15  -0.8f, 0.0f, -0.8f, Green, // 16  -0.8f, 0.0f, -1.0f, Yellow, // 17  -1.0f, -1.0f, -0.8f, Yellow, // 18  -0.8f, -1.0f, -0.8f, Magenta, // 19  -0.8f, -1.0f, -1.0f, Magenta, // 20  -1.0f, -1.0f, -1.0f, Red, // 21  // Leg 3  1.0f, 0.0f, -0.2f, Red, // 22  0.8f, 0.0f, -0.2f, Blue, // 23  0.8f, 0.0f, 0.0f, Blue, // 24  1.0f, -1.0f, -0.2f, Green, // 25  0.8f, -1.0f, -0.2f, Green, // 26  0.8f, -1.0f, 0.0f, Yellow, // 27  1.0f, -1.0f, 0.0f, Yellow, // 28  // Leg 4  0.8f, 0.0f, -1.0f, Blue, // 29  0.8f, 0.0f, -0.8f, Red, // 30  1.0f, 0.0f, -0.8f, Red, // 31  0.8f, -1.0f, -1.0f, Green, // 32  0.8f, -1.0f, -0.8f, Green, // 33  1.0f, -1.0f, -0.8f, Yellow, // 34  1.0f, -1.0f, -1.0f, Yellow // 35  };  // Index data to share position data  GLuint indices[] = {  // Table  1, 2, 4,  1, 6, 4,  0, 1, 2,  0, 3, 2,  3, 2, 4,  3, 5, 4,  4, 5, 7,  4, 6, 7,  0, 1, 6,  0, 7, 6,  // Leg 1  0, 8, 12,  12, 11, 0,  8, 12, 13,  13, 9, 8,  9, 10, 14,  14, 13, 9,  10, 14, 11,  11, 0, 10,  // Leg 2  3, 15, 18,  18, 21, 3,  15, 16, 19,  19, 18, 15,  16, 17, 20,  20, 19, 16,  3, 17, 20,  20, 21, 3,  // Leg 3  22, 23, 25,  23, 25, 26,  23, 24, 26,  24, 26, 27,  24, 7, 27,  27, 7, 28,  7, 22, 25,  7, 25, 28,  // Leg 4  5, 29, 35,  29, 32, 35,  29, 32, 33,  29, 30, 33,  30, 33, 34,  30, 31, 34,  31, 34, 35,  31, 5, 35  };  // Generate buffer ids  glGenVertexArrays(1, &VAO);  glGenBuffers(1, &VBO);  glGenBuffers(2, &EBO);  // Activate the Vertex Array Object before binding and setting and VBOs and Vertex Attribute Pointers  glBindVertexArray(VAO);  // Activate the VBO  glBindBuffer(GL\_ARRAY\_BUFFER, VBO);  glBufferData(GL\_ARRAY\_BUFFER, sizeof(vertices), vertices, GL\_STATIC\_DRAW); // Copy vertices to VBO  // Activate the Element Buffer Object / Indices  glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, EBO);  glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, sizeof(indices), indices, GL\_STATIC\_DRAW); // Copy indices to EBO  // Set attribute pointer 0 to hold Position data  glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 6 \* sizeof(GLfloat), (GLvoid\*)0);  glEnableVertexAttribArray(0); // Enables vertex attribute  // Set attribute pointer 1 to hold Color data  glVertexAttribPointer(1, 3, GL\_FLOAT, GL\_FALSE, 6 \* sizeof(GLfloat), (GLvoid\*)(3 \* sizeof(GLfloat)));  glEnableVertexAttribArray(1); // Enables vertex attribute  int sof = 6 \* sizeof(GLfloat);  int soui = 3 \* sizeof(GLuint);  int vsize = sizeof(vertices);  int isize = sizeof(indices);  vsize = vsize/sof;  isize = isize/soui;  glBindVertexArray(0); // Deactivates the VAO which is good practice  }  // Mouse initializations and callbacks  // occur here  void initializeMouse() {  glutPassiveMotionFunc(UMouseMove); // Detects mouse movement without any mouse buttons pushed  glutMotionFunc(onMotion); // Detects mouse movement while a mouse button is pushed  glutMouseFunc(OnMouseClicks); // Detects mouse click  }  // Keyboard initializations and callbacks  // occur here  void initializeKeyboard() {  glutKeyboardFunc(UKeyboard); // Detects key press  glutKeyboardUpFunc(UKeyReleased); // Detects key release  }  // Detects mouse movement without any mouse buttons pushed  void UMouseMove(int x, int y){  front.x = 10.0f \* cos(yaw);  front.y = 10.0f \* sin(pitch);  front.z = sin(yaw) \* cos(pitch) \* 10.0f;  }  // Implements the UKeyboard function  void UKeyboard(unsigned char key, GLint x, GLint y) {  switch(key) {  case ZOOM\_IN:  case ZOOM\_OUT:  case PAN\_LEFT:  case PAN\_RIGHT:  currentKey = key;  break;  default:  currentKey = '0';  break;  }  }  // Implements the UKeyReleased function  void UKeyReleased(unsigned char key, GLint x, GLint y) {  currentKey = '0';  }  // Detects mouse movement while a mouse button is pushed  void onMotion(int curr\_x, int curr\_y) {  //if left alt and mouse down are set  if (checkMotion) {  // gets the direction the mouse was moved  mouseXOffset = curr\_x - lastMouseX;  mouseYOffset = lastMouseY - curr\_y;  // updates with new mouse coordinates  lastMouseX = curr\_x;  lastMouseY = curr\_y;  // applies sensitivity to mouse direction  mouseXOffset \*= sensitivity;  mouseYOffset \*= sensitivity;  // get the direction of the mouse  // if there is changes in yaw, then it is moving along X  if ((yaw != yaw + mouseXOffset) && (pitch == pitch + mouseYOffset)) {  // increment yaw  yaw += mouseXOffset;  //else movement in y  } else if ((pitch != pitch+mouseYOffset) && (yaw == yaw+mouseXOffset)) {  // increment y to move vertical  pitch += mouseYOffset;  }  front.x = 10.0f \* cos(yaw);  front.y = 10.0f \* sin(pitch);  front.z = sin(yaw) \* cos(pitch) \* 10.0f;  }  // check if user is zooming, alt, right mouse button and down  if (checkZoom) {  // determine the direction of the movement, either up or down  if (lastMouseY < curr\_y) {  // mouse moving up on y  // decrement scale values, zoom in  scale\_by\_x -= 0.1f;  scale\_by\_y -= 0.1f;  scale\_by\_z -= 0.1f;  // control zoom in size  if (scale\_by\_z < 0.2f) {  scale\_by\_x = 0.2f;  scale\_by\_y = 0.2f;  scale\_by\_z = 0.2f;  }  // redisplay  glutPostRedisplay();  } else { // zoom in  // mouse down up on y  // increment scale values  scale\_by\_x += 0.1f;  scale\_by\_y += 0.1f;  scale\_by\_z += 0.1f;  glutPostRedisplay();  }  // update x and y  lastMouseY = curr\_y;  lastMouseX = curr\_x;  }  }  // Detects mouse clicks  void OnMouseClicks(int button, int state, int x, int y) {  modifierKey = glutGetModifiers(); // checks for modifier keys like alt, shif and ctrl  checkMotion = false; //set checkMotion to false  //check if button is left, and mod is alt and state is down, all should be true  if ((button == GLUT\_LEFT\_BUTTON) && (modifierKey == GLUT\_ACTIVE\_ALT) && (state == GLUT\_DOWN)) {  checkMotion = true; // set motion true  checkZoom = false; // set zoom false  } else if ((button == GLUT\_RIGHT\_BUTTON) && (modifierKey == GLUT\_ACTIVE\_ALT) && (state == GLUT\_DOWN)) {  checkMotion = false; // set motion false  checkZoom = true; // set zoom true  }  } |

# References

TIOBE Index for September 2020. (2020, September). Retrieved September 20, 2020, from https://www.tiobe.com/tiobe-index/