CSC 8820 Advanced Graphics Algorithms
PROJECT-1
Ashish Tiwari

```
// -----------Author-----
//Last Name: Tiwari
//First Name: Ashish

#include <fstream>
#include <iostream>
#include <GL/glew.h>
#include <GL/freeglut.h>
// ASSIMP library for loading 3D files
#include "E:/Spring18/AGA/assimp-3.3.1/include/assimp/Importer.hpp"
#include "E:/Spring18/AGA/assimp-3.3.1/include/assimp/PostProcess.h"
#include "E:/Spring18/AGA/assimp-3.3.1/include/assimp/Scene.h"
// GLM library for mathematic functions
#include <E:/Spring18/AGA/glm-0.9.8.5/glm/glm/glm.hpp>
#include <E:/Spring18/AGA/glm-0.9.8.5/glm/glm/gtx/projection.hpp>
#include <E:/Spring18/AGA/glm-0.9.8.5/glm/glm/gtc/matrix_transform.hpp>
#include <E:/Spring18/AGA/glm-0.9.8.5/glm/glm/gtx/transform2.hpp>
#include <E:/Spring18/AGA/glm-0.9.8.5/glm/glm/gtc/type_ptr.hpp>
#include <E:/Spring18/AGA/glm-0.9.8.5/glm/glm/gtc/matrix_access.hpp>
#include <E:/Spring18/AGA/glm-0.9.8.5/glm/glm/gtc/matrix_inverse.hpp>
// SOIL library for loading texture images
#include <E:\Spring18\AGA\soil\Simple OpenGL Image Library\src\SOIL.h>
// This header file contains error checking functions
#include "E:/Spring18/AGA/check_error.hpp"
// This header file contains utility functions that print out the content of
// aiScene data structure.
#include "E:/Spring18/AGA/assimp_utilities.hpp"


using namespace std;
using namespace glm;

#define BUFFER_OFFSET( offset ) ((GLvoid*) offset)

// Default folders for 3D models, images, and shaders
// Update these for your computer.
const char * defaultModelFolder = "..\\..\\..\\Models\\";
const char * defaultImageFolder = "..\\..\\..\\Images\\";
const char * defaultShaderFolder = "..\\..\\..\\code\\Shaders\\";

//------------------------
// Shader related variables

// Shader file names
const char* vShaderFilename = "Tiwari_vShader.glsl";
const char* fShaderFilename = "Tiwari_fShader.glsl";
```

```
// Index of the shader program
GLuint program;

//--------------------------
// 3D object related variable

// 3D object file name
const char * objectFileName = "object.obj";

// This is the Assimp importer object that loads the 3D file.
Assimp::Importer importer;

// Global Assimp scene object
const aiScene* scene = NULL;

// This array stores the VAO indices for each corresponding mesh in the aiScene object.
unsigned int *vaoArray = NULL;

// This is the 1D array that stores the face indices of a mesh.
unsigned int *faceArray = NULL;

//--------------------------------
// Vertex related variables

struct VertexAttributeLocations {
	GLint vPos; // Index of the in variable vPos in the vertex shader
	GLint vNormal; // Index of the in variable vNormal in the vertex shader
	GLint vTextureCoord; // Index of the in variable vTextureCoord in the vertex shader
};

VertexAttributeLocations vertexAttributeLocations;

//--------------------------------
// Transformation related variables

struct MatrixLocations {
	GLuint mvpMatrixID; // uniform attribute: model-view-projection matrix
	GLint modelMatrixID; // uniform variable: model-view matrix
	GLint normalMatrixID; // uniform variable: normal matrix for transforming normal vectors
};

MatrixLocations matrixLocations;

mat4 projMatrix; // projection matrix
mat4 viewMatrix; // view matrix

					//-----------------
					// Camera related variables
vec3 defaultCameraPosition = vec3(2.5f, 1.0f, 3.0f);
vec3 defaultCameraLookAt = vec3(0.0f, 0.0f, 0.0f);
```

```
vec3 defaultCameraUp = vec3(0.0f, 1.0f, 0.0f);

float defaultFOV = 60.0f; // in degrees
float defaultNearPlane = 0.1f;
float defaultFarPlane = 1000.0f;

int windowWidth = 600;
int windowHeight = 400;
GLfloat  tippangle = 0;



// User interactions related parameters
float rotateX = 0;
float rotateY = 0;
float scaleFactor = 1.5f;
float xTranslation = 0.0f;
float yTranslation = 0.0f;
float zTranslation = 0.0f;
float transformationStep = 1.0f;
float rotationStep = 0.5f;


// Load a 3D file
const aiScene* load3DFile(const char *filename) {

	ifstream fileIn(filename);

	// Check if the file exists.
	if (fileIn.good()) {
		fileIn.close();  // The file exists.
	}
	else {
		fileIn.close();
		cout << "Unable to open the 3D file." << endl;
		return false;
	}

	cout << "Loading 3D file " << filename << endl;

	// Load the 3D file using Assimp. The content of the 3D file is stored in an aiScene object.
	const aiScene* sceneObj = importer.ReadFile(filename, 
aiProcessPreset_TargetRealtime_Quality);

	// Check if the file is loaded successfully.
	if (!sceneObj)
	{
		// Fail to load the file
		cout << importer.GetErrorString() << endl;
		return false;
	}
```

```
	else {
		cout << "3D file " << filename << " loaded." << endl;
	}

	// Print the content of the aiScene object, if needed.
	// This function is defined in the check_error.hpp file.
	printAiSceneInfo(sceneObj, PRINT_AISCENE_SUMMARY);

	return sceneObj;
}

//-------------------
// Read a shader file
const char *readShaderFile(const char * filename) {
	ifstream shaderFile(filename);

	if (!shaderFile.is_open()) {
		cout << "Cannot open the shader file " << filename << endl;
		return NULL;
	}

	string line;
	// Must created a new string, otherwise the returned pointer
	// will be invalid
	string *shaderSourceStr = new string();

	while (getline(shaderFile, line)) {
		*shaderSourceStr += line + '\n';
	}

	const char *shaderSource = shaderSourceStr->c_str();

	shaderFile.close();

	return shaderSource;
}



// ----------------------------------
// Get only the file name out of a path
string getFileName(const string& s) {

	char sep = '/';

#ifdef _WIN32
	sep = '\\';
#endif

	size_t i = s.rfind(sep, s.length());
	if (i != string::npos) {
```

```
		return(s.substr(i + 1, s.length() - i));
	}

	return(s);
}

// ---------------------------------------
// Load and build shaders
bool prepareShaders() {

	GLuint vShaderID, fShaderID;

	// Create empty shader objects
	vShaderID = glCreateShader(GL_VERTEX_SHADER);
	checkGlCreateXError(vShaderID, "vShaderID");
	if (vShaderID == 0) {
		return false;
	}

	fShaderID = glCreateShader(GL_FRAGMENT_SHADER);
	checkGlCreateXError(fShaderID, "fShaderID");
	if (fShaderID == 0) {
		return false;
	}

	const char* vShader = readShaderFile(
		(string(defaultShaderFolder) + getFileName(string(vShaderFilename))).c_str());
	if (!vShader) {
		return false;
	}

	// OpenGL fragment shader source code
	const char* fShader = readShaderFile(
		(string(defaultShaderFolder) + getFileName(string(fShaderFilename))).c_str());
	if (!fShader) {
		return false;
	}

	// Attach shader source code the shader objects
	glShaderSource(vShaderID, 1, &vShader, NULL);
	glShaderSource(fShaderID, 1, &fShader, NULL);

	// Compile the vertex shader object
	glCompileShader(vShaderID);
	printShaderInfoLog(vShaderID);
	glCompileShader(fShaderID);
	printShaderInfoLog(fShaderID);
	program = glCreateProgram();
	checkGlCreateXError(program, "program");
	if (program == 0) {
		return false;
```

```
	}

	// Attach vertex and fragment shaders to the shader program
	glAttachShader(program, vShaderID);
	glAttachShader(program, fShaderID);

	// Link the shader program
	glLinkProgram(program);
	// Check if the shader program can run in the current OpenGL state, just for testing
purposes.
	glValidateProgram(program);
	printShaderProgramInfoLog(program); // Print error messages, if any.

	return true;
}

// -------------------------------------------
// Get shader variable locations
void getShaderVariableLocations() {
	glUseProgram(program);

	vertexAttributeLocations.vPos = glGetAttribLocation(program, "vPos");
	checkGlGetXLocationError(vertexAttributeLocations.vPos, "vPos");

	vertexAttributeLocations.vNormal = glGetAttribLocation(program, "vNormal");
	checkGlGetXLocationError(vertexAttributeLocations.vNormal, "vNormal");

	vertexAttributeLocations.vTextureCoord = glGetAttribLocation(program, "vTextureCoord");
	checkGlGetXLocationError(vertexAttributeLocations.vTextureCoord, "vTextureCoord");

	// Get the ID of the uniform matrix variable in the vertex shader.
	matrixLocations.mvpMatrixID = glGetUniformLocation(program, "mvpMatrix");
	if (matrixLocations.mvpMatrixID == -1) {
		cout << "There is an error getting the handle of GLSL uniform variable mvp_matrix."
<< endl;
	}

	matrixLocations.modelMatrixID = glGetUniformLocation(program, "modelMatrix");
	if (matrixLocations.modelMatrixID == -1) {
		cout << "There is an error getting the handle of GLSL uniform variable mvMatrix." <<
endl;
	}

	matrixLocations.normalMatrixID = glGetUniformLocation(program, "normalMatrix");
	if (matrixLocations.mvpMatrixID == -1) {
		cout << "There is an error getting the handle of GLSL uniform variable
normalMatrix." << endl;
	}

}
```

```
// Load 3D data from 3D file with Assimp


bool load3DData() {

	GLuint buffer;

	// Load the 3D file using Assimp.
	// Assume that the 3D file is stored in the default model folder.
	scene = load3DFile(
		(string(defaultModelFolder) + string(getFileName(objectFileName))).c_str());

	if (!scene) {
		return false;
	}


	// Create an array to store the VAO indices for each mesh.
	vaoArray = (unsigned int*)malloc(sizeof(unsigned int) * scene->mNumMeshes);

	// Go through each mesh stored in the aiScene object, bind it with a VAO,
	// and save the VAO index in the vaoArray.
	for (unsigned int i = 0; i < scene->mNumMeshes; i++) {
		const aiMesh* currentMesh = scene->mMeshes[i];

		// Create an empty Vertex Array Object (VAO). VAO is only available from OpenGL
3.0 or higher.
		glGenVertexArrays(1, &vaoArray[i]);
		glBindVertexArray(vaoArray[i]);

		if (currentMesh->HasPositions()) {
			// Create an empty Vertex Buffer Object (VBO)
			glGenBuffers(1, &buffer);
			glBindBuffer(GL_ARRAY_BUFFER, buffer);

			// Bind (transfer) the vertex position array (stored in aiMesh's member
variable mVertices)
			// to the VBO.
			glBufferData(GL_ARRAY_BUFFER, sizeof(float) * 3 * currentMesh-
>mNumVertices,currentMesh->mVertices, GL_STATIC_DRAW);

			glEnableVertexAttribArray(vertexAttributeLocations.vPos);
			glVertexAttribPointer(vertexAttributeLocations.vPos, 3, GL_FLOAT,
GL_FALSE, 0, BUFFER_OFFSET(0));
		}

		if (currentMesh->HasFaces()) {
			// Create an array to store the face indices (elements) of this mesh.
			faceArray = (unsigned int*)malloc(sizeof(unsigned int) * currentMesh-
>mNumFaces * currentMesh->mFaces[0].mNumIndices);
```

```
				// copy the face indices from aiScene into a 1D array faceArray.
				int faceArrayIndex = 0;
				for (unsigned int j = 0; j < currentMesh->mNumFaces; j++) {
					for (unsigned int k = 0; k < currentMesh->mFaces[j].mNumIndices;
k++) {
						faceArray[faceArrayIndex] = currentMesh-
>mFaces[j].mIndices[k];
						faceArrayIndex++;
					}
				}

				// Create an empty VBO
				glGenBuffers(1, &buffer);
	glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffer);
				glBufferData(GL_ELEMENT_ARRAY_BUFFER,
					sizeof(unsigned int) * currentMesh->mNumFaces * currentMesh-
>mFaces[0].mNumIndices,
					faceArray, GL_STATIC_DRAW);
			}

			if (currentMesh->HasNormals()) {
				// Create an empty Vertex Buffer Object (VBO)
				glGenBuffers(1, &buffer);
				glBindBuffer(GL_ARRAY_BUFFER, buffer);

				glBufferData(GL_ARRAY_BUFFER, sizeof(float) * 3 * currentMesh-
>mNumVertices,
					currentMesh->mNormals, GL_STATIC_DRAW);
				glEnableVertexAttribArray(vertexAttributeLocations.vNormal);
				glVertexAttribPointer(vertexAttributeLocations.vNormal, 3, GL_FLOAT,
GL_FALSE, 0, BUFFER_OFFSET(0));
			}



			if (currentMesh->HasTextureCoords(0)) {
				// Create an empty Vertex Buffer Object (VBO)
				glGenBuffers(1, &buffer);
				glBindBuffer(GL_ARRAY_BUFFER, buffer);
			}

			//Close the VAOs and VBOs for later use.
			glBindVertexArray(0);
			glBindBuffer(GL_ARRAY_BUFFER, 0);
			glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
		} // end for
```

```
}
//-------------------------------
//Prepare the shaders and 3D data
bool init()
{
	// Load shaders
	if (prepareShaders() == false) {
		return false;
	}

	getShaderVariableLocations();

	if (load3DData() == false) {
		return false;
	}

	// Turn on visibility test.
	glEnable(GL_DEPTH_TEST);

	// Draw the object in wire frame mode.
	glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);

	// This is the background color.
	glClearColor(0.0, 0.0, 0.0, 0.5f);

	checkOpenGLError("init()");

	return true;
}

//-----------------------------------------------
// Traverse the scene graph to find a camera, update its location and direction,
// and then create the view and projection matrices.
void nodeTreeTraversalCamera(const aiNode* node, aiMatrix4x4 matrix) {

	if (!node) {
		cout << "nodeTreeTraversalCamera(): Null node" << endl;
		return;
	}

	// Camera and lights reference a specific node by name, if any.
	string name = node->mName.C_Str();

	// Calculate this (camera) node's transformation matrix.
	aiMatrix4x4 currentTransformMatrix = matrix * node->mTransformation;

	// Check every camera on the camera list
	for (unsigned int i = 0; i < scene->mNumCameras; i++) {
		aiCamera* currentCamera = scene->mCameras[i];

		string currentCameraName = currentCamera->mName.C_Str();
```

```
            // If the current camera is the same as the camera node ...
            if (currentCameraName.compare(name) == 0) {

                // It's not clear whether we also need to multiply the camera's local matrix.
                // Maybe it's necessary for some 3D file format.
                aiMatrix4x4 cameraMatrix;
                currentCamera->GetCameraMatrix(cameraMatrix);
                //currentTransformMatrix = currentTransformMatrix *cameraMatrix;

                // Get the camera position, look-at, and up vector.
                // Don't modify aiCamera's member variables mPosition, mLookAt, mUp
directly.
                aiVector3D cameraPosition = currentCamera->mPosition;
                aiVector3D cameraLookAtPosition = currentCamera->mLookAt;
                aiVector3D cameraUpVector = currentCamera->mUp;

                // Transform the camera position, lookAt, and up vector
                cameraPosition = currentTransformMatrix * cameraPosition;
                cameraLookAtPosition = currentTransformMatrix * cameraLookAtPosition;
                cameraUpVector = currentTransformMatrix * cameraUpVector;
                cameraUpVector.Normalize(); // Remember to normalize the UP vector.

                                                    // Pass the eye
position to the shader. We'll need it for calculating
                                                    // the specular
color.
                float eyePosition[3] = { cameraPosition.x, cameraPosition.y,
cameraPosition.z };
                //glUniform3fv(lightSourceLocations.eyePosition, 1, eyePosition);

                // Build the projection and view matrices
                // It's better to use the window's aspect than using the aspect ratio from the
3D file.
                projMatrix = perspective(currentCamera->mHorizontalFOV,
                    (float)windowWidth / (float)windowHeight,
                    currentCamera->mClipPlaneNear,
                    currentCamera->mClipPlaneFar);

                // Create a view matrix
                // You specify where the camera location and orientation and GLM will
create a view matrix.
                viewMatrix = lookAt(vec3(cameraPosition.x, cameraPosition.y,
cameraPosition.z),
                    vec3(cameraLookAtPosition.x, cameraLookAtPosition.y,
cameraLookAtPosition.z),
                    vec3(cameraUpVector.x, cameraUpVector.y, cameraUpVector.z));
            } // end if camera name is the same
        } // end for

        // Recursively visit and find a camera in child nodes. This is a depth-first traversal.
```

```
	for (unsigned int j = 0; j < node->mNumChildren; j++) {
		nodeTreeTraversalCamera(node->mChildren[j], currentTransformMatrix);
	}
}

//---------------------------------------------
// Traverse the scene graph to update the locations and directions of light sources.
void nodeTreeTraversalLight(const aiNode* node, aiMatrix4x4 matrix) {

	if (!node) {
		cout << "nodeTreeTraversal(): Null node" << endl;
		return;
	}

	string nodeName = node->mName.C_Str();

	// Calculate this (light) node's transformation matrix.
	aiMatrix4x4 currentTransformMatrix = matrix * node->mTransformation;



	// Recursively visit and find a light in child nodes. This is a depth-first traversal.
	for (unsigned int j = 0; j < node->mNumChildren; j++) {
		nodeTreeTraversalLight(node->mChildren[j], currentTransformMatrix);
	}
}


//------------------------------------------------------------------------------------------
// Traverse the node tree in the aiScene object and draw the meshes associated with each node.

void nodeTreeTraversalMesh(const aiNode* node, aiMatrix4x4 matrix) {
	if (!node) {
		cout << "nodeTreeTraversal(): Null node" << endl;
		return;
	}

	// Multiply the parent's node's transformation matrix with this node's transformation
matrix.
	aiMatrix4x4 currentTransformMatrix = matrix * node->mTransformation;

	if (node->mNumMeshes > 0) {
		// If this node contains meshes, we'll caculate the model-view-projection matrix for
it.

		// Conver the transformation matrix from aiMatrix4x4 to glm:mat4 format.
		// aiMatrix4x4 is row major.
		mat4 modelMatrix = mat4(1.0);
		modelMatrix = row(modelMatrix, 0, vec4(currentTransformMatrix.a1,
			currentTransformMatrix.a2, currentTransformMatrix.a3,
currentTransformMatrix.a4));
```

```
			modelMatrix = row(modelMatrix, 1, vec4(currentTransformMatrix.b1,
				currentTransformMatrix.b2, currentTransformMatrix.b3,
currentTransformMatrix.b4));
			modelMatrix = row(modelMatrix, 2, vec4(currentTransformMatrix.c1,
				currentTransformMatrix.c2, currentTransformMatrix.c3,
currentTransformMatrix.c4));
			modelMatrix = row(modelMatrix, 3, vec4(currentTransformMatrix.d1,
				currentTransformMatrix.d2, currentTransformMatrix.d3,
currentTransformMatrix.d4));


	//***************************************************************************
********
			// Combine the model, view, and project matrix into one model-view-projection
matrix.

			// Model matrix is then multiplied with view matrix and projection matrix to create a
combined
			// model_view_projection matrix.
			// The view and projection matrices are created in the previous traversal of the
scene that processes
			// the camera data.

			// The sequence of multiplication is important here. Model matrix, view matrix, and
projection matrix
			// must be multiplied from right to left, because the vertex position is on the right
hand side.
			mat4 mvpMatrix = projMatrix * viewMatrix * modelMatrix;

			// Create a normal matrix to transform normals.
			// We don't need to include the view matrix here because the lighting is done
			// in world space.
			mat3 normalMatrix = mat3(1.0);
			normalMatrix = column(normalMatrix, 0, vec3(modelMatrix[0][0],
modelMatrix[0][1], modelMatrix[0][2]));
			normalMatrix = column(normalMatrix, 1, vec3(modelMatrix[1][0],
modelMatrix[1][1], modelMatrix[1][2]));
			normalMatrix = column(normalMatrix, 2, vec3(modelMatrix[2][0],
modelMatrix[2][1], modelMatrix[2][2]));

			normalMatrix = inverseTranspose(normalMatrix);

			// Draw all the meshes associated with the current node.
			// Certain node may have multiple meshes associated with it.
			for (unsigned int i = 0; i < node->mNumMeshes; i++) {
				// This is the index of the mesh associated with this node.
				int meshIndex = node->mMeshes[i];

				const aiMesh* currentMesh = scene->mMeshes[meshIndex];
```

```
			glUniformMatrix4fv(matrixLocations.mvpMatrixID, 1, GL_FALSE,
glm::value_ptr(mvpMatrix));

			glUniformMatrix4fv(matrixLocations.modelMatrixID, 1, GL_FALSE,
glm::value_ptr(modelMatrix));
			glUniformMatrix3fv(matrixLocations.normalMatrixID, 1, GL_FALSE,
glm::value_ptr(normalMatrix));

			// This is the material for this mesh
			unsigned int materialIndex = currentMesh->mMaterialIndex;

			glBindVertexArray(vaoArray[meshIndex]);

			// How many faces are in this mesh?
			unsigned int numFaces = currentMesh->mNumFaces;
			// How many indices are for each face?
			unsigned int numIndicesPerFace = currentMesh->mFaces[0].mNumIndices;

			glDrawElements(GL_TRIANGLES, (numFaces * numIndicesPerFace),
GL_UNSIGNED_INT, 0);

			// We are done with the current VAO. Move on to the next VAO, if any.
			glBindVertexArray(0);
		}

	} // end if (node->mNumMeshes > 0)

	for (unsigned int j = 0; j < node->mNumChildren; j++) {
		nodeTreeTraversalMesh(node->mChildren[j], currentTransformMatrix);
	}
}

//--------------------------
// Display callback function
void display() {
	// Clear the background color and the depth buffer.
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

	// Activate the shader program.
	glUseProgram(program);

	// Traverse the scene graph to update the location and direction of the camera.
	if (scene->HasCameras()) {
		nodeTreeTraversalCamera(scene->mRootNode, aiMatrix4x4());
	}
	else {
		// If there is no camera data in the file, create the default projection and view
matrices.
		projMatrix = perspective(radians(defaultFOV), (float)windowWidth /
(float)windowHeight, defaultNearPlane, defaultFarPlane);
```

```
			viewMatrix = lookAt(defaultCameraPosition, defaultCameraLookAt,
defaultCameraUp);

		}

		// Traverse the scene graph to upate the location and direction of the light sources.
		if (scene->HasLights()) {
			nodeTreeTraversalLight(scene->mRootNode, aiMatrix4x4());
		}

		aiMatrix4x4 rotationXMatrix = aiMatrix4x4::RotationX(radians(rotateX), rotationXMatrix);
		aiMatrix4x4 rotationYMatrix = aiMatrix4x4::RotationY(radians(rotateY), rotationYMatrix);
		aiMatrix4x4 scaleMatrix = aiMatrix4x4::Scaling(aiVector3D(scaleFactor, scaleFactor,
scaleFactor), scaleMatrix);
		aiMatrix4x4 translationXMatrix = aiMatrix4x4::Translation(aiVector3D(xTranslation, 0.0f,
0.0f),
			translationXMatrix);
		aiMatrix4x4 translationYMatrix = aiMatrix4x4::Translation(aiVector3D(0.0f, yTranslation,
0.0f),
			translationYMatrix);
		aiMatrix4x4 translationZMatrix = aiMatrix4x4::Translation(aiVector3D(0.0f, 0.0f,
zTranslation),
			translationZMatrix);
		aiMatrix4x4 overallTransformationMatrix =
			translationZMatrix * translationXMatrix * translationYMatrix * rotationXMatrix *
rotationYMatrix * scaleMatrix;

		// Start the node tree traversal and process each node. The overallTransformationMatrix is
passed down
		// the scene through the root node.

		if (scene->HasMeshes()) {
			nodeTreeTraversalMesh(scene->mRootNode, overallTransformationMatrix);
		}

		// Swap front and back buffers. The rendered image is now displayed.
		glutSwapBuffers();
}

//-----------------------------------------------------------------
// This function is called when the size of the window is changed.
void reshape(int width, int height)
{
		// Specify the width and height of the picture within the window
		// Creates a viewport matrix and insert it into the graphics pipeline.
		// This operation is not done in shader, but taken care of by the hardware.
		glViewport(0, 0, width, height);

		windowWidth = width;
		windowHeight = height;
}
```

```
//---------------------------------------
// Use up and down keys to move the object along the Y axis.
void specialKeys(int key, int x, int y) {
	int centerX = windowWidth / 2;
	int centerY = windowHeight / 2;

  switch (key) {
	case GLUT_KEY_UP:
		//yTranslation += transformationStep;
			rotateX += (float)(y - centerY) * 0.3f;
		break;
	case GLUT_KEY_DOWN:
		//yTranslation -= transformationStep;
		rotateX -= (float)(y - centerY) * 0.3f;
		break;
   case GLUT_KEY_LEFT:
		//yTranslation += transformationStep;
		rotateY += (float)(x - centerX) * 0.3f;
		break;
   case GLUT_KEY_RIGHT:
		rotateY -= (float)(x - centerX) * 0.3f;
		//yTranslation -= transformationStep;
		break;
	default:
		break;
	}

	// Generate a dislay event to force refreshing the window.
	glutPostRedisplay();
}



int main(int argc, char* argv[])
{
	glutInit(&argc, argv);

	// Initialize double buffer and depth buffer.
	glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH);

	glutCreateWindow(argv[0]);
	glutReshapeWindow(windowWidth, windowHeight);

	// Initialize Glew. This must be called before any OpenGL function call.
	glewInit();

	// Initialize OpenGL debug context, which is available from OpenGL 4.3+.
	// Must call this after glewInit().
```

```
	initOpenGLDebugContext(true);

	// This cannot be called before glewInit().
	cout << "OpenGL version " << glGetString(GL_VERSION) << endl;

	if (init()) {

		// Register callback functions
		glutDisplayFunc(display);

		glutReshapeFunc(reshape);


		glutSpecialFunc(specialKeys);


		glutMainLoop();

		//Release the dynamically allocated memory blocks.
		free(vaoArray);
		free(faceArray);

	}
}
```

Vertex Shader

```
#version 330

in vec3 vPos;
in vec3 vNormal;
in vec2 vTextureCoord;

uniform mat4 mvpMatrix; // model_view_project matrix
uniform mat4 modelMatrix;       // model view matrix
uniform mat3 normalMatrix; // model matrix

out vec3 N; // The normal vector is passed over to the fragment shader
out vec3 v; // Vertex position is passed over to the fragment shader
out vec2 textureCoord; // The texture coordinates are passed over to the fragment shader

// Note that there is no out color, because the pixel color is calculated
// in the fragment shader.

void main()
{
   vec4 position = vec4(vPos.xyz, 1.0);
   gl_Position = mvpMatrix * position;

   vec4 transformedPosition = modelMatrix * position;
   v = transformedPosition.xyz;
```

```
    N = normalize(normalMatrix * vNormal);

        textureCoord = vTextureCoord;
}
```

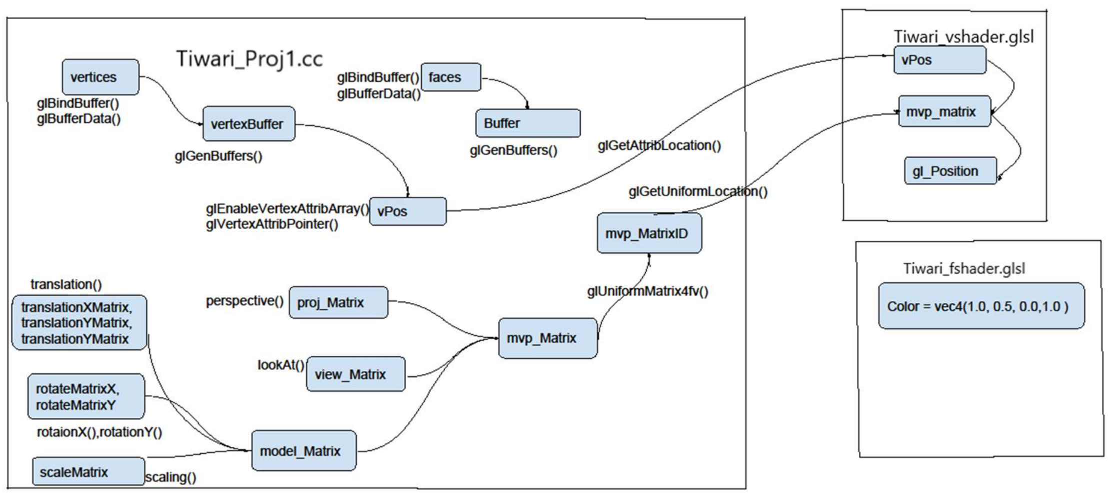
Fragment Shader

```
#version 330

in vec3 N; // interpolated normal for the pixel
in vec3 v; // interpolated position for the pixel
in vec2 textureCoord; // interpolated texture coordinate for the pixel

const int maxNumLights = 50;

uniform vec4 lightSourcePosition[maxNumLights];

// light direction
uniform vec4 lightDirection[maxNumLights];

uniform vec4 diffuseLightIntensity[maxNumLights];
uniform vec4 specularLightIntensity[maxNumLights];
uniform vec4 ambientLightIntensity[maxNumLights];

// for calculating the light attenuation
uniform float constantAttenuation[maxNumLights];
uniform float linearAttenuation[maxNumLights];
uniform float quadraticAttenuation[maxNumLights];

// Spotlight cutoff angle
uniform float spotlightOuterCone[maxNumLights];
uniform float spotlightInnerCone[maxNumLights];
uniform int lightType[maxNumLights];

uniform int numLights;

uniform vec4 Kambient;
uniform vec4 Kdiffuse;
uniform vec4 Kspecular;
uniform vec4 emission;
uniform float shininess;

uniform vec3 eyePosition;

uniform int hasTexture;
uniform sampler2D texUnit;

out vec4 color;
```

```
// This fragment shader is an example of per-pixel lighting.
void main() {

	// Now calculate the parameters for the lighting equation:
	// color = Ka * Lag + (Ka * La) + attenuation * ((Kd * (N dot L) * Ld) + (Ks * ((N dot HV) ^ 
shininess) * Ls))
	// Ka, Kd, Ks: surface material properties
	// Lag: global ambient light (not used in this example)
	// La, Ld, Ls: ambient, diffuse, and specular components of the light source
	// N: normal
	// L: light vector
	// HV: half vector
	// shininess
	// attenuation: light intensity attenuation over distance and spotlight angle

	color = vec4(1.0, 0.5, 0.0, 1.0);

	for (int i = 0; i < numLights; i++) {
		vec3 lightVector;
		float attenuation = 1.0;

		if (lightType[i] == 1) {
			// point light source
			lightVector = normalize(lightSourcePosition[i].xyz - v);

			// calculate light attenuation
			float distance = distance(lightSourcePosition[i].xyz, v);

			attenuation = 1.0 / (constantAttenuation[i] + (linearAttenuation[i] * 
distance)
				+ (quadraticAttenuation[i] * distance * distance));

		}
		else if (lightType[i] == 2) {
			// directional light source. The light position is actually the light vector.
			lightVector = lightSourcePosition[i].xyz;

			// For directional lights, there is no light attenuation.
			attenuation = 1.0;
		}
		else if (lightType[i] == 3) {
			// spotlight source
			lightVector = normalize(lightSourcePosition[i].xyz - v);

			float distance = distance(lightSourcePosition[i].xyz, v);

			float spotEffect = dot(normalize(lightDirection[i].xyz), 
normalize(lightVector));

			// spotlightInnerCone is in radians, not degrees.
			if (spotEffect > cos(spotlightInnerCone[i])) {
```

```
					// If the vertex is in the spotlight cone
					attenuation = spotEffect / (constantAttenuation[i] +
linearAttenuation[i] * distance +
						quadraticAttenuation[i] * distance * distance);
				}
				else if (spotEffect > cos(spotlightOuterCone[i])) {
					// Between inner and outer spotlight cone, make the light attenuate
sharply.
					attenuation = (pow(spotEffect, 12)) / (constantAttenuation[i] +
linearAttenuation[i] * distance +
						quadraticAttenuation[i] * distance * distance);
				}
				else {
					// If the fragment is outside of the spotlight cone, then there is no
light.
					attenuation = 0.0;
				}
			}
			else {
				attenuation = 0.0;
			}

			//calculate Diffuse Color
			float NdotL = max(dot(N, lightVector), 0.0);

			vec4 diffuseColor = Kdiffuse * diffuseLightIntensity[i] * NdotL;

			// calculate Specular color. Here we use the original Phong illumination model.
			vec3 E = normalize(eyePosition - v);

			vec3 R = normalize(-reflect(lightVector, N)); // light reflection vector

			float RdotE = max(dot(R, E), 0.0);

			vec4 specularColor = Kspecular * specularLightIntensity[i] * pow(RdotE, shininess);

			// ambient color
			vec4 ambientColor = Kambient * ambientLightIntensity[i];

			color += ambientColor + emission + attenuation * (diffuseColor + specularColor);
		}

		if (hasTexture == 1) {
			// Perform the texture mapping.
			// Retrieve the texture color from the texture image using the texture
			// coordinates.
			vec4 textureColor = texture(texUnit, textureCoord);

			// Combine the lighting color with the texture color.
			// You can use different methods to combine the two colors.
			color = mix(color, textureColor, 0.5f);
```

```
        }
}
```



Tiwari_Proj1.cc
vertices
glBindBuffer()
glBufferData()
vertexBuffer
glGenBuffers()
glBindBuffer()
glBufferData()
faces
Buffer
glGenBuffers()
glEnableVertexAttribArray()
glVertexAttribPointer()
vPos
glGetAttribLocation()
glGetUniformLocation()
mvp_MatrixID
glUniformMatrix4fv()
translation()
translationXMatrix,
translationYMatrix,
translationYMatrix
perspective()
proj_Matrix
lookAt()
view_Matrix
mvp_Matrix
rotateMatrixX,
rotateMatrixY
rotaionX(),rotationY()
model_Matrix
scaleMatrix
scaling()
Tiwari_vshader.glsl
vPos
mvp_matrix
gl_Position
Tiwari_fshader.glsl
Color = vec4(1.0, 0.5, 0.0,1.0)