**Department of Computing**

**CS-361: Computer Graphics  
  
Class: BSCS-12ABC & SE12AB**

**Lab 06: Viewing in Computer Graphics**

# CLO-02: Develop 2D and 3D graphical applications using programming libraries and tools.

# CLO-03: Implement algorithms for rendering, transformations, and animations.

**Date: 04th March 2025**

**Time: 12:00 PM – 14:25 PM**

# Instructor: Dr. Sidra Sutana

# Lab Engineer: Mr. Aftab Farooq

**Name:** Ahmed Mohiuddin Shah

**CMS ID:** 415216

**Section:** BSCS-12-A

**Lab:** 6

# Lab 06: Viewing in Computer Graphics

### **Lab Objective:**

## The objective of this lab is to help students understand and implement the concepts of Orthographic Projection and Perspective Projection in computer graphics. Students will learn how to transform 3D objects into 2D representations using these projection techniques.

### **Overview : Orthographic Projection :**

Orthographic projection is a parallel projection technique where all projection lines are orthogonal to the projection plane. It preserves the relative proportions of objects but does not provide a realistic view of depth.

**Steps:**

1. Set up the OpenGL environment.
2. Define a 3D object (e.g., a cube or pyramid).
3. Use the glOrtho() function to set up an orthographic projection matrix.
4. Render the 3D object and observe the 2D representation.

### **Perspective Projection :**

## Perspective projection simulates the way the human eye perceives objects in the real world. Objects farther away appear smaller, creating a sense of depth.

**Steps:**

1. Set up the OpenGL environment.
2. Define a 3D object (e.g., a cube or pyramid).
3. Use the gluPerspective() function to set up a perspective projection matrix.
4. Render the 3D object and observe the 2D representation with depth perception.

## 

## Tools/Software Requirement:

* **Operating System:**
  + Windows / macOS / Linux (Ubuntu recommended)
* **Development Environment:**
  + **Windows:** [Code::Blocks](http://www.codeblocks.org/) or [Visual Studio](https://visualstudio.microsoft.com/)
  + **macOS:** [Xcode](https://developer.apple.com/xcode/)
  + **Linux:** GCC and g++ compilers
* **Graphics Libraries:**
  + **OpenGL** (built-in on macOS and Linux, available in Windows IDEs)
  + **GLUT** (OpenGL Utility Toolkit)
  + **GLEW** (OpenGL Extension Wrangler Library)
* **Package Manager (for macOS/Linux):**
  + **Homebrew** (macOS): brew install freeglut glew
  + **APT** (Linux): sudo apt-get install freeglut3-dev glew-utils
* **Compilers:**
  + **Windows:** MinGW (for Code::Blocks) or Microsoft C++ Compiler (for Visual Studio)
  + **macOS/Linux:** GCC/G++
* A programming environment (e.g., Visual Studio, PyCharm, or any IDE of your choice).
* A graphics library (optional, e.g., OpenGL, SDL, or a simple image library for saving images).
* Basic knowledge of viewing in computer graphics.

## Lab Tasks :

**Task 1: Implementing Orthographic Projection**

* Write a program to implement an orthographic projection of a 3D object onto a 2D plane.

**Solution:**

I loaded a model of a teapot.

**Code:**

#define GLM\_ENABLE\_EXPERIMENTAL

#include <cmath>

#include <vector>

#include <iomanip>

#include <iostream>

#include <GL/glut.h>

#include <glm/glm.hpp>

#include <assimp/scene.h>

#include <assimp/Importer.hpp>

#include <glm/gtc/type\_ptr.hpp>

#include <assimp/postprocess.h>

#include <glm/gtx/string\_cast.hpp>

#include <glm/gtc/matrix\_transform.hpp>

*// Model transformation variables*

**float** modelX = 0.0f, modelY = 0.0f, modelZ = 0.0f; *// Model position*

**float** modelRotX = 90.0f, modelRotY = -140.0f, modelRotZ = 100.0f; *// Model rotation*

*// Light properties*

glm::vec3 lightPos = glm::vec3(70.0f, 80.0f, 80.0f); *// Light position*

glm::vec3 lightColor = glm::vec3(1.0f, 1.0f, 1.0f); *// Light color (white)*

**float** ambientIntensity = 0.2f; *// Ambient intensity*

**float** diffuseIntensity = 0.8f; *// Diffuse intensity*

**float** specularIntensity = 1.0f; *// Specular intensity*

**float** shininess = 50.0f; *// Shininess exponent*

*// Material properties (model color is red)*

glm::vec3 matAmbient = glm::vec3(0.7f, 0.0f, 0.0f); *// Ambient reflection (red)*

glm::vec3 matDiffuse = glm::vec3(0.8f, 0.0f, 0.0f); *// Diffuse reflection (red)*

glm::vec3 matSpecular = glm::vec3(0.5f, 0.5f, 0.5f); *// Specular reflection (white)*

*// Model data*

std::vector<glm::vec3> vertices;

std::vector<glm::vec3> normals;

std::vector<**unsigned** **int**> indices;

std::vector<glm::vec3> vertexColors; *// Colors for Gouraud shading*

*// Function to load a 3D model using Assimp*

**bool** loadModel(**const** std::string**&** path) {

   Assimp::Importer importer;

**const** aiScene\* scene = importer.ReadFile(path, aiProcess\_Triangulate | aiProcess\_FlipUVs | aiProcess\_GenNormals);

   if (!scene || scene->mFlags & AI\_SCENE\_FLAGS\_INCOMPLETE || !scene->mRootNode) {

       std::cerr << "Error loading model: " << importer.GetErrorString() << std::endl;

       return false;

   }

*// Process the first mesh (for simplicity)*

   aiMesh\* mesh = scene->mMeshes[0];

*// Extract vertices and normals*

   for (**unsigned** **int** i = 0; i < mesh->mNumVertices; i++) {

*// Vertices*

       vertices.push\_back(glm::vec3(mesh->mVertices[i].x, mesh->mVertices[i].y, mesh->mVertices[i].z));

*// Normals*

       normals.push\_back(glm::vec3(mesh->mNormals[i].x, mesh->mNormals[i].y, mesh->mNormals[i].z));

   }

*// Extract indices*

   for (**unsigned** **int** i = 0; i < mesh->mNumFaces; i++) {

       aiFace face = mesh->mFaces[i];

       for (**unsigned** **int** j = 0; j < face.mNumIndices; j++) {

           indices.push\_back(face.mIndices[j]);

       }

   }

   return true;

}

*// Function to compute lighting for a vertex*

glm::vec3 computeLighting(**const** glm::vec3**&** vertex, **const** glm::vec3**&** normal) {

*// Ambient component*

   glm::vec3 ambient = matAmbient \* ambientIntensity;

*// Diffuse component*

   glm::vec3 lightDir = glm::normalize(lightPos - vertex);

**float** diff = glm::max(glm::dot(normal, lightDir), 0.0f);

   glm::vec3 diffuse = matDiffuse \* diffuseIntensity \* diff;

*// Specular component*

   glm::vec3 viewDir = glm::normalize(glm::vec3(0.0f, 0.0f, 5.0f) - vertex); *// Camera at (0, 0, 5)*

   glm::vec3 reflectDir = glm::reflect(-lightDir, normal);

**float** spec = glm::pow(glm::max(glm::dot(viewDir, reflectDir), 0.0f), shininess);

   glm::vec3 specular = matSpecular \* specularIntensity \* spec;

*// Total lighting*

   return ambient + diffuse + specular;

}

*// Function to calculate vertex colors for Gouraud shading*

**void** calculateVertexColors() {

   vertexColors.clear();

   for (**size\_t** i = 0; i < vertices.size(); i++) {

       glm::vec3 color = computeLighting(vertices[i], normals[i]);

       vertexColors.push\_back(color);

   }

}

*// Function to render the model with Gouraud shading*

**void** renderModel() {

   glBegin(GL\_TRIANGLES);

   for (**size\_t** i = 0; i < indices.size(); i += 3) {

*// Get vertices and normals for the triangle*

       glm::vec3 v0 = vertices[indices[i]];

       glm::vec3 v1 = vertices[indices[i + 1]];

       glm::vec3 v2 = vertices[indices[i + 2]];

       glm::vec3 n0 = normals[indices[i]];

       glm::vec3 n1 = normals[indices[i + 1]];

       glm::vec3 n2 = normals[indices[i + 2]];

*// Compute lighting for each vertex*

       glm::vec3 color0 = computeLighting(v0, n0);

       glm::vec3 color1 = computeLighting(v1, n1);

       glm::vec3 color2 = computeLighting(v2, n2);

*// Draw the triangle with interpolated colors*

       glColor3f(color0.x, color0.y, color0.z);

       glVertex3f(v0.x, v0.y, v0.z);

       glColor3f(color1.x, color1.y, color1.z);

       glVertex3f(v1.x, v1.y, v1.z);

       glColor3f(color2.x, color2.y, color2.z);

       glVertex3f(v2.x, v2.y, v2.z);

   }

   glEnd();

}

*// Function to draw a small sphere for the light*

**void** drawLight() {

   glPushMatrix();

   glTranslatef( lightPos.x, lightPos.y, lightPos.z); *// Move to light position*

   glColor3f(1.0f, 1.0f, 1.0f); *// Light color (white)*

   glutSolidSphere(10.0f, 30, 30); *// Draw a small sphere*

   glPopMatrix();

}

**void** drawInfinitePlane() {

**const** **float** gridSize = 1000.0f; *// Size of the grid (extent in each direction)*

**const** **float** gridStep = 10.0f; *// Distance between grid lines*

*// Enable blending for transparency (optional)*

   glEnable(GL\_BLEND);

   glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA);

*// Set the color of the grid lines (light gray)*

   glColor4f(0.7f, 0.7f, 0.7f, 0.5f); *// RGBA (A = alpha for transparency)*

*// Draw the grid*

   glBegin(GL\_LINES);

   for (**float** i = -gridSize; i <= gridSize; i += gridStep) {

*// Horizontal lines*

       glVertex3f(-gridSize, i, 0.0f);

       glVertex3f(gridSize, i, 0.0f);

*// Vertical lines*

       glVertex3f(i, -gridSize, 0.0f);

       glVertex3f(i, gridSize, 0.0f);

   }

   glEnd();

*// Disable blending (if enabled)*

   glDisable(GL\_BLEND);

}

*// Function to handle keyboard input (move the light)*

**void** keyboard(**unsigned** **char** key, **int** x, **int** y) {

**float** moveAmount = 10.0f; *// Light movement speed*

      switch (key) {

      case 'w': modelRotX += moveAmount; break; *// Move light forward*

      case 's': modelRotX -= moveAmount; break; *// Move light backward*

      case 'a': modelRotY -= moveAmount; break; *// Move light left*

      case 'd': modelRotY += moveAmount; break; *// Move light right*

      case 'q': modelRotZ += moveAmount; break; *// Move light up*

      case 'e': modelRotZ -= moveAmount; break; *// Move light down*

      }

      std::cout <<  modelRotX << ", " <<  modelRotY << ", " <<  modelRotZ << ", " << std::endl;

*// Recalculate vertex colors for Gouraud shading*

   calculateVertexColors();

   glutPostRedisplay(); *// Redraw the scene*

}

*// Function to handle mouse motion (rotate model)*

**void** mouseMotion(**int** x, **int** y) {

**static** **int** lastX = x, lastY = y;

**int** deltaX = x - lastX;

**int** deltaY = y - lastY;

   modelRotY += deltaX \* 0.5f; *// Rotate around Y axis*

   modelRotX += deltaY \* 0.5f; *// Rotate around X axis*

   lastX = x;

   lastY = y;

   glutPostRedisplay(); *// Redraw the scene*

}

**void** printMatrix(glm::mat4 matrix) {

    for (**int** i = 0; i < 4; ++i) { *// Rows*

        std::cout << " [ " << "\t";

        for (**int** j = 0; j < 4; ++j) { *// Columns*

            std::cout << std::fixed << std::showpoint;

            std::cout << std::setprecision(4);

            std::cout << **float**(matrix[i][j]) << "\t\t"; *// Tab-separated values*

        }

        std::cout << " ] " << std::endl; *// Newline for each row*

    }

}

**void** printProjectionMatrix() {

*// Define the projection matrix (Perspective Projection Example)*

    glm::mat4 projection = glm::perspective(glm::radians(45.0f), 800.0f / 600.0f, 0.1f, 100.0f);

*// Print projection matrix*

    std::cout << "Projection Matrix:\n" << std::endl;

    printMatrix(projection);

}

**void** printModelViewMatrix() {

    GLfloat modelview[16];

    glGetFloatv(GL\_MODELVIEW\_MATRIX, modelview);

    glm::mat4 modelViewMat = glm::make\_mat4(modelview);

    std::cout << "ModelView Matrix:\n" << std::endl;

    printMatrix(modelViewMat);

}

*// Function to render the scene*

**void** display() {

   glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

   glLoadIdentity();

*// Set up the camera (fixed camera, move the model instead)*

   gluLookAt(10.0f, 10.0f, 0.0f, *// Camera position*

       0.0f, 0.0f, 0.0f, *// Look at point*

       0.0f, 1.0f, 0.0f); *// Up vector*

   printProjectionMatrix();

   printModelViewMatrix();

*// Apply model transformations*

   glTranslatef(modelX, modelY, modelZ); *// Move the model*

   glRotatef(modelRotX, 1.0f, 0.0f, 0.0f); *// Rotate around X axis*

   glRotatef(modelRotY, 0.0f, 1.0f, 0.0f); *// Rotate around Y axis*

   glRotatef(modelRotZ, 0.0f, 0.0f, 1.0f); *// Rotate around Z axis*

   drawInfinitePlane();

*// Render the model*

   renderModel();

*// Draw the light*

   drawLight();

   glutSwapBuffers();

}

*// Function to initialize OpenGL settings*

**void** init() {

   glEnable(GL\_DEPTH\_TEST); *// Enable depth testing*

   glClearColor(0.0, 0.0, 0.0, 1.0);

   glMatrixMode(GL\_PROJECTION);

   glLoadIdentity();

   glOrtho(-100.0, 100.0, -100.0, 100.0, -100.0, 1000.0); *// Orthographic projection setup*

*// Load the model*

   if (!loadModel("model.stl")) {

       std::cerr << "Failed to load model" << std::endl;

       exit(1);

   }

*// Calculate vertex colors for Gouraud shading*

   calculateVertexColors();

}

*// Function to handle window resizing*

**void** reshape(**int** w, **int** h) {

   glViewport(0, 0, w, h);

   glMatrixMode(GL\_PROJECTION);

   glLoadIdentity();

   glOrtho(-100.0, 100.0, -100.0, 100.0, -100.0, 1000.0);

   glMatrixMode(GL\_MODELVIEW);

}

**int** main(**int** argc, **char\*\*** argv) {

   glutInit(&argc, argv);

   glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_RGB | GLUT\_DEPTH);

   glutInitWindowSize(800, 600);

   glutCreateWindow("Task1 - Orthographic Projection");

   init();

   glutDisplayFunc(display);

   glutReshapeFunc(reshape);

   glutKeyboardFunc(keyboard);

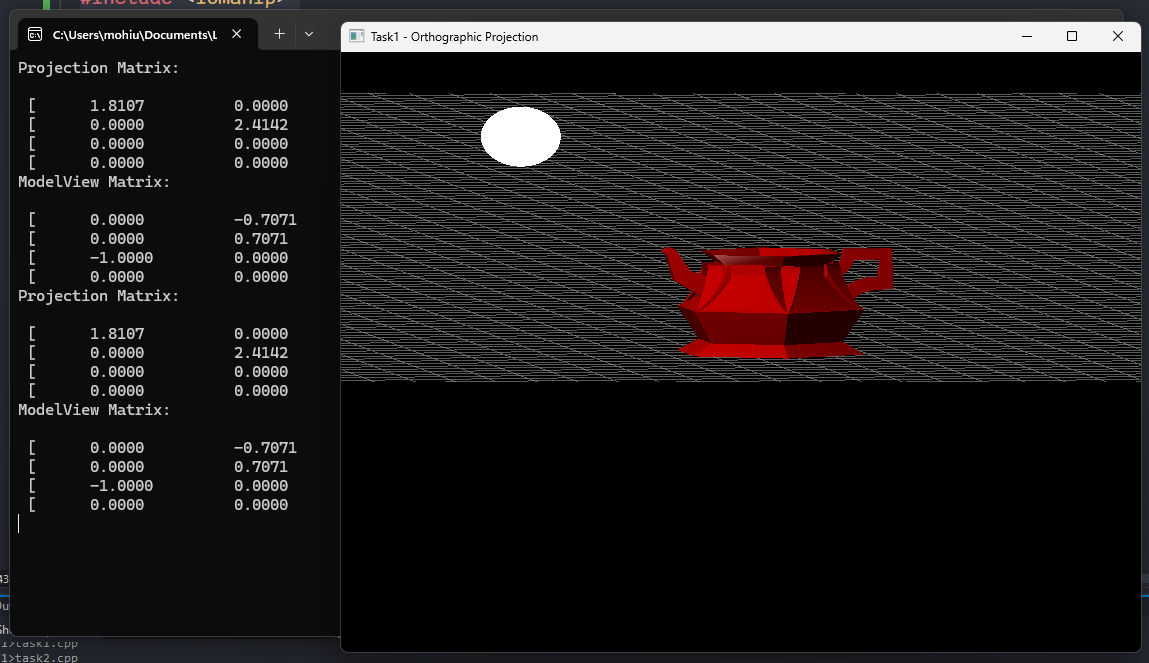
   glutMotionFunc(mouseMotion);

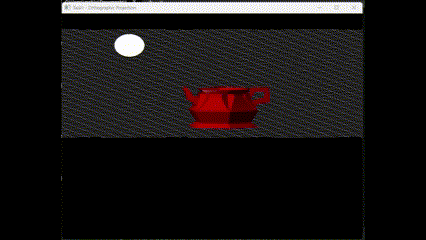
   glutMainLoop();

   return 0;

}

**Output:**

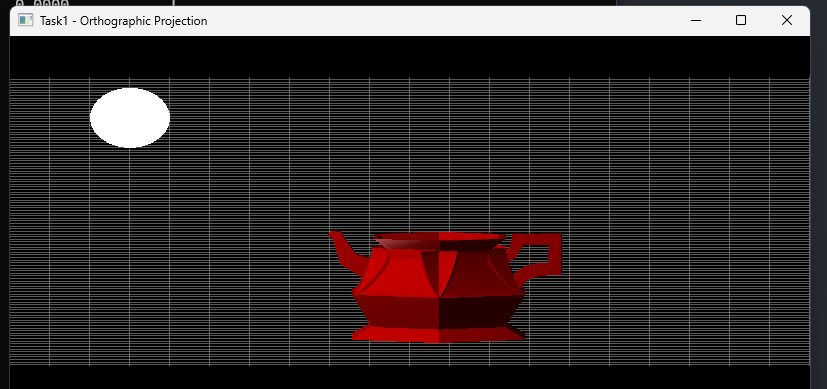
****

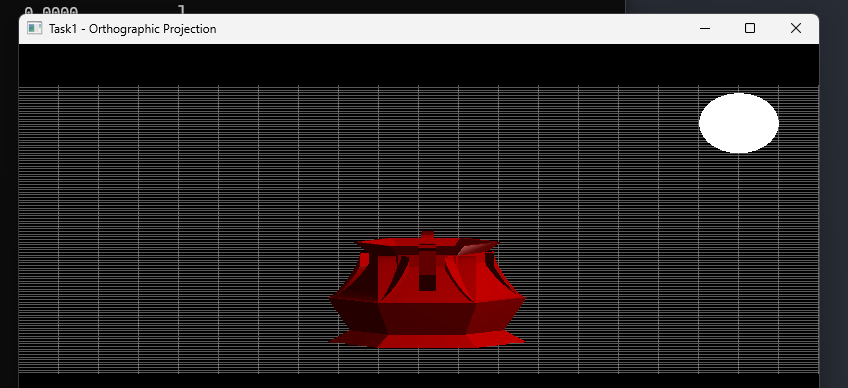
****

* Demonstrate how different viewing angles affect the output.

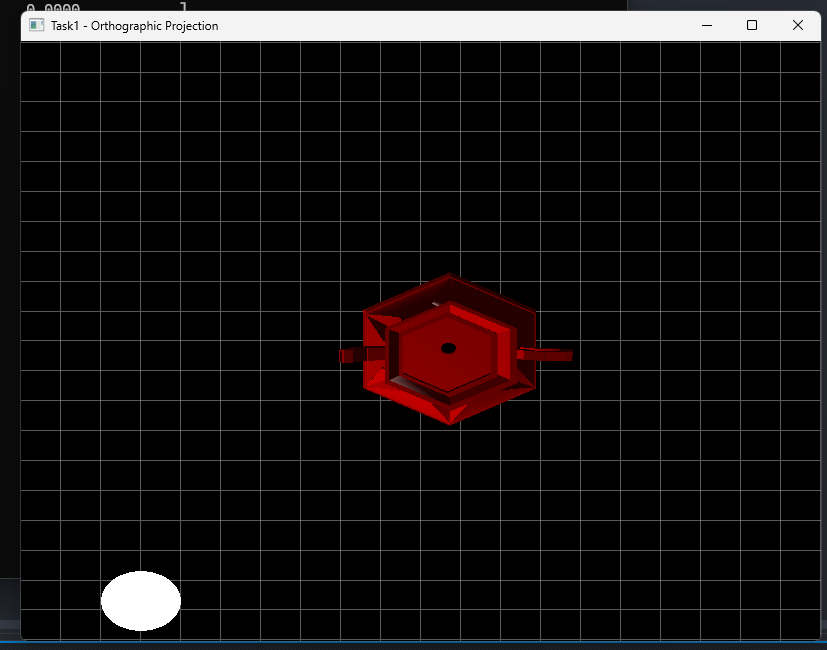
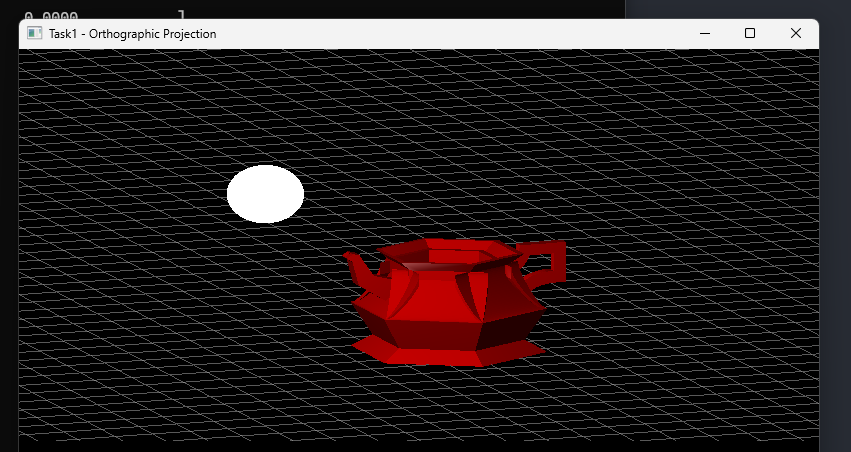
I have used WASDQE and mouse dragging to change the viewing angles of the model to show how the output is affected. In orthographic projection I observed that that the size of the object is not changing and is same at any angle and distance from the camera and also the lines stay parallel and do not seem to converge to a single point.

**Output:**

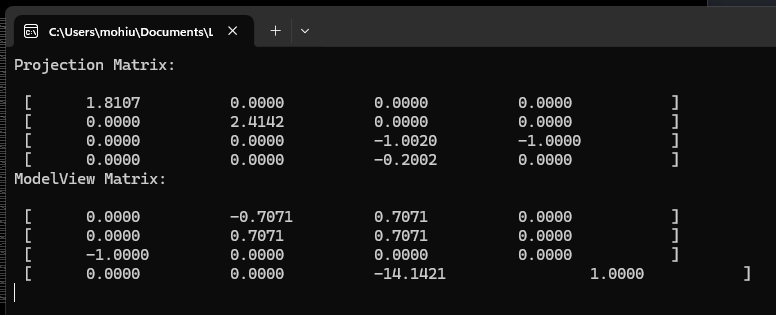


 A computer screen with a teapot

AI-generated content may be incorrect. A screen shot of a computer

AI-generated content may be incorrect.  

* Display projection matrics.

**Output:**  
 ****

**Task 2: Implementing Perspective Projection**

* Develop a program to project a 3D object using a perspective projection model.
* Apply perspective transformation matrices.

**Solution:**

**Code:**

#define GLM\_ENABLE\_EXPERIMENTAL

#include <cmath>

#include <vector>

#include <iomanip>

#include <iostream>

#include <GL/glut.h>

#include <glm/glm.hpp>

#include <assimp/scene.h>

#include <assimp/Importer.hpp>

#include <glm/gtc/type\_ptr.hpp>

#include <assimp/postprocess.h>

#include <glm/gtx/string\_cast.hpp>

#include <glm/gtc/matrix\_transform.hpp>

*// Field of View Stuff*

**float** FOV = 75.0f;

*// Model transformation variables*

**float** modelX = 0.0f, modelY = 0.0f, modelZ = 0.0f; *// Model position*

**float** modelRotX = 90.0f, modelRotY = -140.0f, modelRotZ = 100.0f; *// Model rotation*

*// Light properties*

glm::vec3 lightPos = glm::vec3(70.0f, 80.0f, 80.0f); *// Light position*

glm::vec3 lightColor = glm::vec3(1.0f, 1.0f, 1.0f); *// Light color (white)*

**float** ambientIntensity = 0.2f; *// Ambient intensity*

**float** diffuseIntensity = 0.8f; *// Diffuse intensity*

**float** specularIntensity = 1.0f; *// Specular intensity*

**float** shininess = 50.0f; *// Shininess exponent*

*// Material properties (model color is red)*

glm::vec3 matAmbient = glm::vec3(0.7f, 0.0f, 0.0f); *// Ambient reflection (red)*

glm::vec3 matDiffuse = glm::vec3(0.8f, 0.0f, 0.0f); *// Diffuse reflection (red)*

glm::vec3 matSpecular = glm::vec3(0.5f, 0.5f, 0.5f); *// Specular reflection (white)*

*// Model data*

std::vector<glm::vec3> vertices;

std::vector<glm::vec3> normals;

std::vector<**unsigned** **int**> indices;

std::vector<glm::vec3> vertexColors; *// Colors for Gouraud shading*

*// Function to load a 3D model using Assimp*

**bool** loadModel(**const** std::string**&** path) {

    Assimp::Importer importer;

**const** aiScene\* scene = importer.ReadFile(path, aiProcess\_Triangulate | aiProcess\_FlipUVs | aiProcess\_GenNormals);

    if (!scene || scene->mFlags & AI\_SCENE\_FLAGS\_INCOMPLETE || !scene->mRootNode) {

        std::cerr << "Error loading model: " << importer.GetErrorString() << std::endl;

        return false;

    }

*// Process the first mesh (for simplicity)*

    aiMesh\* mesh = scene->mMeshes[0];

*// Extract vertices and normals*

    for (**unsigned** **int** i = 0; i < mesh->mNumVertices; i++) {

*// Vertices*

        vertices.push\_back(glm::vec3(mesh->mVertices[i].x, mesh->mVertices[i].y, mesh->mVertices[i].z));

*// Normals*

        normals.push\_back(glm::vec3(mesh->mNormals[i].x, mesh->mNormals[i].y, mesh->mNormals[i].z));

    }

*// Extract indices*

    for (**unsigned** **int** i = 0; i < mesh->mNumFaces; i++) {

        aiFace face = mesh->mFaces[i];

        for (**unsigned** **int** j = 0; j < face.mNumIndices; j++) {

            indices.push\_back(face.mIndices[j]);

        }

    }

    return true;

}

*// Function to compute lighting for a vertex*

glm::vec3 computeLighting(**const** glm::vec3**&** vertex, **const** glm::vec3**&** normal) {

*// Ambient component*

    glm::vec3 ambient = matAmbient \* ambientIntensity;

*// Diffuse component*

    glm::vec3 lightDir = glm::normalize(lightPos - vertex);

**float** diff = glm::max(glm::dot(normal, lightDir), 0.0f);

    glm::vec3 diffuse = matDiffuse \* diffuseIntensity \* diff;

*// Specular component*

    glm::vec3 viewDir = glm::normalize(glm::vec3(0.0f, 0.0f, 5.0f) - vertex); *// Camera at (0, 0, 5)*

    glm::vec3 reflectDir = glm::reflect(-lightDir, normal);

**float** spec = glm::pow(glm::max(glm::dot(viewDir, reflectDir), 0.0f), shininess);

    glm::vec3 specular = matSpecular \* specularIntensity \* spec;

*// Total lighting*

    return ambient + diffuse + specular;

}

*// Function to calculate vertex colors for Gouraud shading*

**void** calculateVertexColors() {

    vertexColors.clear();

    for (size\_t i = 0; i < vertices.size(); i++) {

        glm::vec3 color = computeLighting(vertices[i], normals[i]);

        vertexColors.push\_back(color);

    }

}

*// Function to render the model with Gouraud shading*

**void** renderModel() {

    glBegin(GL\_TRIANGLES);

    for (size\_t i = 0; i < indices.size(); i += 3) {

*// Get vertices and normals for the triangle*

        glm::vec3 v0 = vertices[indices[i]];

        glm::vec3 v1 = vertices[indices[i + 1]];

        glm::vec3 v2 = vertices[indices[i + 2]];

        glm::vec3 n0 = normals[indices[i]];

        glm::vec3 n1 = normals[indices[i + 1]];

        glm::vec3 n2 = normals[indices[i + 2]];

*// Compute lighting for each vertex*

        glm::vec3 color0 = computeLighting(v0, n0);

        glm::vec3 color1 = computeLighting(v1, n1);

        glm::vec3 color2 = computeLighting(v2, n2);

*// Draw the triangle with interpolated colors*

        glColor3f(color0.x, color0.y, color0.z);

        glVertex3f(v0.x, v0.y, v0.z);

        glColor3f(color1.x, color1.y, color1.z);

        glVertex3f(v1.x, v1.y, v1.z);

        glColor3f(color2.x, color2.y, color2.z);

        glVertex3f(v2.x, v2.y, v2.z);

    }

    glEnd();

}

*// Function to draw a small sphere for the light*

**void** drawLight() {

    glPushMatrix();

    glTranslatef(lightPos.x, lightPos.y, lightPos.z); *// Move to light position*

    glColor3f(1.0f, 1.0f, 1.0f); *// Light color (white)*

    glutSolidSphere(10.0f, 30, 30); *// Draw a small sphere*

    glPopMatrix();

}

**void** drawInfinitePlane() {

**const** **float** gridSize = 1000.0f; *// Size of the grid (extent in each direction)*

**const** **float** gridStep = 10.0f; *// Distance between grid lines*

*// Enable blending for transparency (optional)*

    glEnable(GL\_BLEND);

    glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA);

*// Set the color of the grid lines (light gray)*

    glColor4f(0.7f, 0.7f, 0.7f, 0.5f); *// RGBA (A = alpha for transparency)*

*// Draw the grid*

    glBegin(GL\_LINES);

    for (**float** i = -gridSize; i <= gridSize; i += gridStep) {

*// Horizontal lines*

        glVertex3f(-gridSize, i, 0.0f);

        glVertex3f(gridSize, i, 0.0f);

*// Vertical lines*

        glVertex3f(i, -gridSize, 0.0f);

        glVertex3f(i, gridSize, 0.0f);

    }

    glEnd();

*// Disable blending (if enabled)*

    glDisable(GL\_BLEND);

}

*// Function to handle keyboard input (move the light)*

**void** keyboard(**unsigned** **char** key, **int** x, **int** y) {

**float** moveAmount = 10.0f; *// Light movement speed*

    switch (key) {

    case 'w': lightPos.x += moveAmount; break; *// Move light forward*

    case 's': lightPos.x -= moveAmount; break; *// Move light backward*

    case 'a': lightPos.y -= moveAmount; break; *// Move light left*

    case 'd': lightPos.y += moveAmount; break; *// Move light right*

    case 'q': lightPos.z += moveAmount; break; *// Move light up*

    case 'e': lightPos.z -= moveAmount; break; *// Move light down*

    case '=': FOV += 5.0; break;

    case '-': FOV -= 5.0; break;

    }

    std::cout << lightPos.x << ", " << lightPos.y << ", " << lightPos.z << std::endl;

    std::cout << "FOV: " << FOV << std::endl;

*// Recalculate vertex colors for Gouraud shading*

    calculateVertexColors();

*// Prevent extreme values*

    if (FOV < 10.0f) FOV = 10.0f;

    if (FOV > 120.0f) FOV = 120.0f;

    glutPostRedisplay(); *// Redraw the scene*

}

*// Function to handle mouse motion (rotate model)*

**void** mouseMotion(**int** x, **int** y) {

**static** **int** lastX = x, lastY = y;

**int** deltaX = x - lastX;

**int** deltaY = y - lastY;

    modelRotY += deltaX \* 0.5f; *// Rotate around Y axis*

    modelRotX += deltaY \* 0.5f; *// Rotate around X axis*

    lastX = x;

    lastY = y;

    glutPostRedisplay(); *// Redraw the scene*

    glutReshapeWindow(glutGet(GLUT\_WINDOW\_WIDTH), glutGet(GLUT\_WINDOW\_HEIGHT));

}

**void** printMatrix(glm::mat4 matrix) {

    for (**int** i = 0; i < 4; ++i) { *// Rows*

        std::cout << " [ " << "\t";

        for (**int** j = 0; j < 4; ++j) { *// Columns*

            std::cout << std::fixed << std::showpoint;

            std::cout << std::setprecision(4);

            std::cout << **float**(matrix[i][j]) << "\t\t"; *// Tab-separated values*

        }

        std::cout << " ] " << std::endl; *// Newline for each row*

    }

}

**void** printProjectionMatrix() {

*// Define the projection matrix (Perspective Projection Example)*

    glm::mat4 projection = glm::perspective(FOV, 800.0f / 600.0f, 0.1f, 100.0f);

*// Print projection matrix*

    std::cout << "Projection Matrix:\n" << std::endl;

    printMatrix(projection);

}

**void** printModelViewMatrix() {

    GLfloat modelview[16];

    glGetFloatv(GL\_MODELVIEW\_MATRIX, modelview);

    glm::mat4 modelViewMat = glm::make\_mat4(modelview);

    std::cout << "ModelView Matrix:\n" << std::endl;

    printMatrix(modelViewMat);

}

*// Function to render the scene*

**void** display() {

    glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

    glLoadIdentity();

*// Set up the camera (fixed camera, move the model instead)*

    gluLookAt(10.0f, 50.0f, 100.0f, *// Camera position*

        0.0f, 0.0f, 0.0f, *// Look at point*

        0.0f, 1.0f, 0.0f); *// Up vector*

    printProjectionMatrix();

    printModelViewMatrix();

*// Apply model transformations*

    glTranslatef(modelX, modelY, modelZ); *// Move the model*

    glRotatef(modelRotX, 1.0f, 0.0f, 0.0f); *// Rotate around X axis*

    glRotatef(modelRotY, 0.0f, 1.0f, 0.0f); *// Rotate around Y axis*

    glRotatef(modelRotZ, 0.0f, 0.0f, 1.0f); *// Rotate around Z axis*

    drawInfinitePlane();

*// Render the model*

    renderModel();

*// Draw the light*

    drawLight();

    glutSwapBuffers();

}

*// Function to initialize OpenGL settings*

**void** init() {

    glEnable(GL\_DEPTH\_TEST); *// Enable depth testing*

    glClearColor(0.0, 0.0, 0.0, 1.0);

    glMatrixMode(GL\_PROJECTION);

    glLoadIdentity();

    gluPerspective(FOV, 800/600.0, 0.1, 1000.0);

*// Load the model*

    if (!loadModel("model.stl")) {

        std::cerr << "Failed to load model" << std::endl;

        exit(1);

    }

*// Calculate vertex colors for Gouraud shading*

    calculateVertexColors();

}

*// Function to handle window resizing*

**void** reshape(**int** w, **int** h) {

    glViewport(0, 0, w, h);

    glMatrixMode(GL\_PROJECTION);

    glLoadIdentity();

    gluPerspective(FOV, 800 / 600.0, 0.1, 1000.0); *// Apply new FOV*

    glMatrixMode(GL\_MODELVIEW);

}

**int** main(**int** argc, **char\*\*** argv) {

    glutInit(&argc, argv);

    glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_RGB | GLUT\_DEPTH);

    glutInitWindowSize(800, 600);

    glutCreateWindow("Task2 - Perspective Projection");

    init();

    glutDisplayFunc(display);

    glutReshapeFunc(reshape);

    glutKeyboardFunc(keyboard);

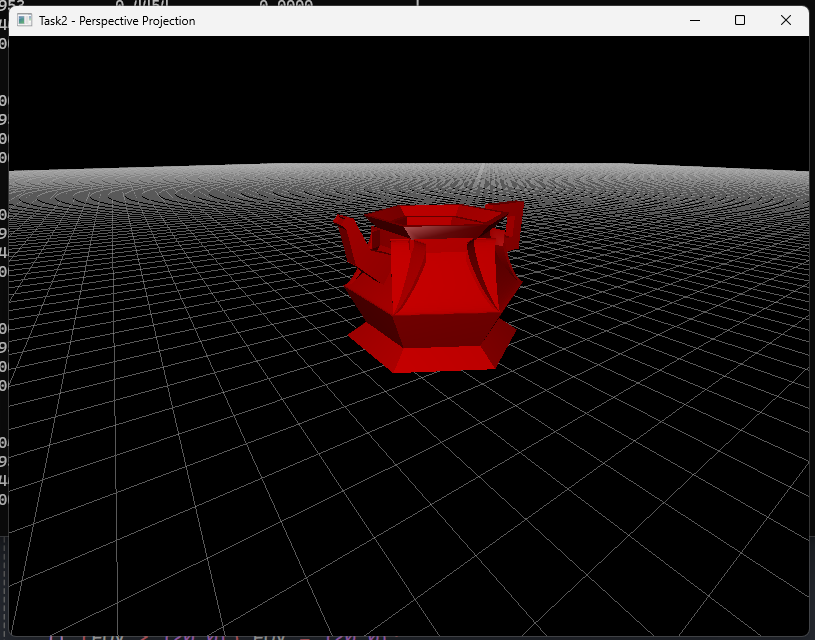
    glutMotionFunc(mouseMotion);

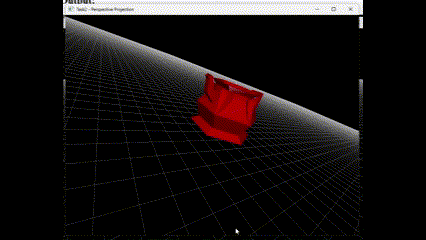
    glutMainLoop();

    return 0;

}

**Output:**

****

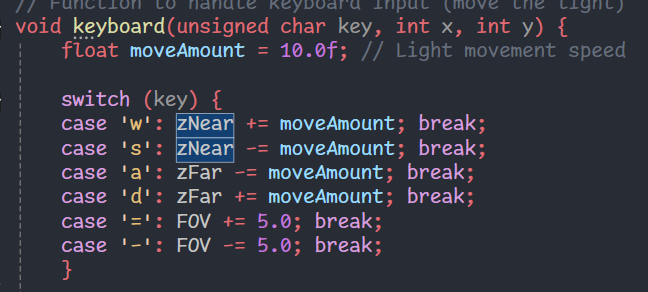
****

* Allow user interaction to change the field of view.

OpenGL was not allowing me to dynamically change the FOV or Near or Far plane Dynamically by User input so I had to change the FOV and far/Near plane manually to show their effects. I have left the appropriate code still inside the code but it isn’t working.

A number of numbers and symbols

AI-generated content may be incorrect.



**Output:**

* ⁠Demonstrate how different viewing angles affect the output along with different view angles, change near and far units

|  |  |  |  |
| --- | --- | --- | --- |
| Planes/FOV | 45 | 90 | 135 |
| Near=0.1, Far=1000.0 |  |  |  |
| Near=0.1, Far=100.0 |  |  |  |
| Near=30, Far=1000.0 |  |  |  |
| Near=30, Far=100.0 |  |  |  |

**Output:**

* Display projection matrics.

### **Output:**

A screenshot of a computer

AI-generated content may be incorrect.

### **Deliverables:**

 Compile a single word document by filling in the solution part and submit this Word file on LMS

 Include screenshots of the program outputs.

 Submit your Lab Word File and code files seperately on submission link.

# Lab Rubrics

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Lab Rubrics for (Lab 06:Viewing in Computer Graphics) | | | | | |
|  | | | | | |
| **Sr.**  **No.** | **Assessment** | **Unacceptable (0 Marks)** | **Does Not Meet Expectations (1/2 Marks)** | **Meets Expectations (3/4 Marks)** | **Exceeds Expectations (5 Marks)** |
| **1** | **Illustrating the basic understanding of semantics and syntax**  **(CLO3, PLO5)** | The student did not submit any work.  OR  The student plagiarized the solution and/or used unfair means. | The student is unable to demonstrate the understanding of syntax of C language and is unable to write an executable code.  The student is not able to understand the structure of a program at all. | The student demonstrates some understanding of syntax of C language and is able to write a code with few errors.  The student is able to understand the structure but still learning the syntax. | The student demonstrates good understanding of syntax of C language and is able to write executable code without help  The student is able to understand the structure and is able to identify problems in the code  when introduced |
| **2** | **Software Tool Usage**  **(CLO4-PLO3)** | The student demonstrates a lack of understanding of tool usage.  Implementation has syntax/semantic/runtime errors, and the student is unable to debug and correct the errors.  The code has inadequate comments and variable names and does not adhere to the coding standards.  No Error handling has been performed.  Documentation is poorly structured. | The student demonstrates some understanding of tool usage.  The codes are correct in terms of their syntax, however, the program output is not always correct in all test cases.  The code has limited comments and inconsistent variable names and may not adhere to the coding standards.  Some Error handling has been performed.  Documentation is adequately structured. | The student demonstrates a good understanding of tool usage.  Furthermore, his/her coding is complete and functional, and the program output is correct in all test cases.  The code has sufficient comments and consistent variable names and reasonably adhere to the coding standards.  Adequate Error handling has been performed.  Documentation is well structured. |