**Department of Computing**

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

**Lab 04: Demonstrating Gouraud and Phong Shading & Visualizing Lighting Vectors**  
   
CLO-02: Develop 2D and 3D graphical applications using programming libraries and tools.

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

**Date: 18th Feb 2025**

**Time: 02:00 PM – 04:50 PM**

# Instructor: Dr. Sidra Sutana

# Lab Engineer: Mr. Aftab Farooq

# Name: Ahmed Mohiuddin Shah

# CMS ID: 415216

# Section: BSCS-12-A

# Lab: 4

**Lab 04: Demonstrating Gouraud and Phong Shading & Visualizing Lighting Vectors**

### **Lab Objective:**

1. To demonstrate and compare **Gouraud** and **Phong shading** techniques on a 3D model.

## 2. To visualize how changing the direction of light sources affects shading on the same model.

## 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 3D rendering tool (Blender, Unity, or OpenGL).
* A sample 3D model (e.g., a sphere, cube, or any complex model).

## Prerequisites :

 Basic understanding of 3D graphics concepts.

##  Familiarity with a 3D rendering tool (e.g., Blender, Unity, or OpenGL).  Knowledge of lighting models and shading techniques.

## Lab Tasks :

Lab Task-01. Setup the 3D Environment: Load a 3D model into the rendering tool.

**Solution:**

This code loads an stl or obj 3d model into the 3d environment using assimp model loading library.

**Code:**

#include <GL/glut.h>

#include <assimp/Importer.hpp>

#include <assimp/scene.h>

#include <assimp/postprocess.h>

#include <vector>

#include <iostream>

// Model transformation variables-155, -40, -100,

*float* modelX = -155.0f, modelY = -40.0f, modelZ = -100.0f; // Model position

*float* modelRotX = -60.0f, modelRotY = 0.0f, modelRotZ = -30.0f; // Model rotation

// Lighting variables

GLfloat lightPos[] = { 2.0f, 2.0f, 2.0f, 1.0f }; // Point light position

GLfloat lightColor[] = { 1.0f, 1.0f, 1.0f, 1.0f }; // Light color (white)

// Material properties (for Gouraud shading)

GLfloat matAmbient[] = { 0.2f, 0.2f, 0.2f, 1.0f };

GLfloat matDiffuse[] = { 0.8f, 0.8f, 0.8f, 1.0f };

GLfloat matSpecular[] = { 1.0f, 1.0f, 1.0f, 1.0f };

GLfloat matShininess[] = { 50.0f };

// Model data

std::vector<*float*> vertices;

std::vector<*float*> normals;

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

// 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(mesh->mVertices[i].x);

        vertices.push\_back(mesh->mVertices[i].y);

        vertices.push\_back(mesh->mVertices[i].z);

        // Normals

        normals.push\_back(mesh->mNormals[i].x);

        normals.push\_back(mesh->mNormals[i].y);

        normals.push\_back(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 render the model

*void* renderModel() {

    glBegin(GL\_TRIANGLES);

    for (*unsigned* *int* i = 0; i < indices.size(); i++) {

*unsigned* *int* index = indices[i];

        glNormal3f(normals[index \* 3], normals[index \* 3 + 1], normals[index \* 3 + 2]);

        glVertex3f(vertices[index \* 3], vertices[index \* 3 + 1], vertices[index \* 3 + 2]);

    }

    glEnd();

}

// 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(0.0f, 0.0f, 5.0f,  // Camera position

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

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

    // 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

    // Render the model

    renderModel();

    glutSwapBuffers();

}

// Function to handle window resizing

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

    glViewport(0, 0, w, h);

    glMatrixMode(GL\_PROJECTION);

    glLoadIdentity();

    gluPerspective(45.0f, (*float*)w / (*float*)h, 0.1f, 10000.0f); // Set perspective projection

    glMatrixMode(GL\_MODELVIEW);

}

// Function to initialize OpenGL settings

*void* init() {

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

    glEnable(GL\_NORMALIZE);  // Normalize normals for lighting

    //setupLighting();         // Set up lighting

    // Load the model

    if (!loadModel("model.stl")) {

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

        exit(1);

    }

}

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

    glutInit(&argc, argv);

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

    glutInitWindowSize(800, 600);

    glutCreateWindow("Task1 - Loading A 3D model STL/OBJ");

    init();

    glutDisplayFunc(display);

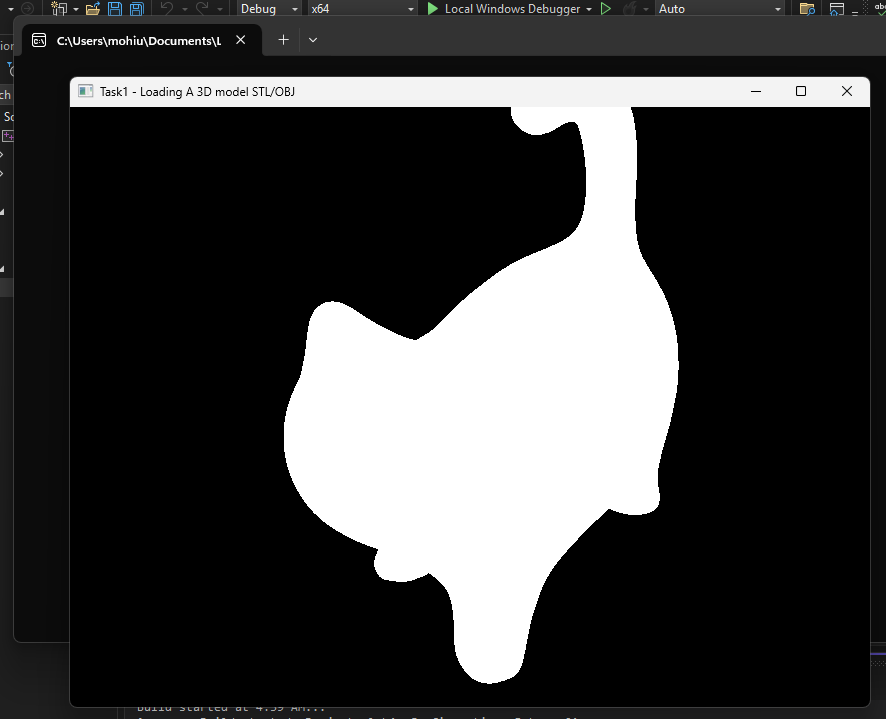
    glutReshapeFunc(reshape);

    glutMainLoop();

    return 0;

}

**Output:**

****

**I loaded a 3d model of a cat :3.**

Lab Task-02 Implement Gouraud Shading: Apply Gouraud shading to the model and observe the results.

**Solution:**

**Code:**

#include <GL/glut.h>

#include <assimp/Importer.hpp>

#include <assimp/scene.h>

#include <assimp/postprocess.h>

#include <vector>

#include <iostream>

#include <cmath>

#include <glm/glm.hpp>

// Model transformation variables-155, -40, -100,

*float* modelX = -155.0f, modelY = -40.0f, modelZ = -100.0f; // Model position

*float* modelRotX = -60.0f, modelRotY = 0.0f, modelRotZ = -30.0f; // Model rotation

// Light properties

glm::vec3 lightPos = glm::vec3(170.0f, 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

   }

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

}

// 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(0.0f, 0.0f, 5.0f,  // Camera position

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

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

   // 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

   // 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(60.0f, (*float*)*w* / (*float*)*h*, 0.1f, 10000.0f); // Set perspective projection

   glMatrixMode(GL\_MODELVIEW);

}

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

   glutInit(&*argc*, *argv*);

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

   glutInitWindowSize(800, 600);

   glutCreateWindow("Task2 - Gouraud Shading");

   init();

   glutDisplayFunc(display);

   glutReshapeFunc(reshape);

   glutKeyboardFunc(keyboard);

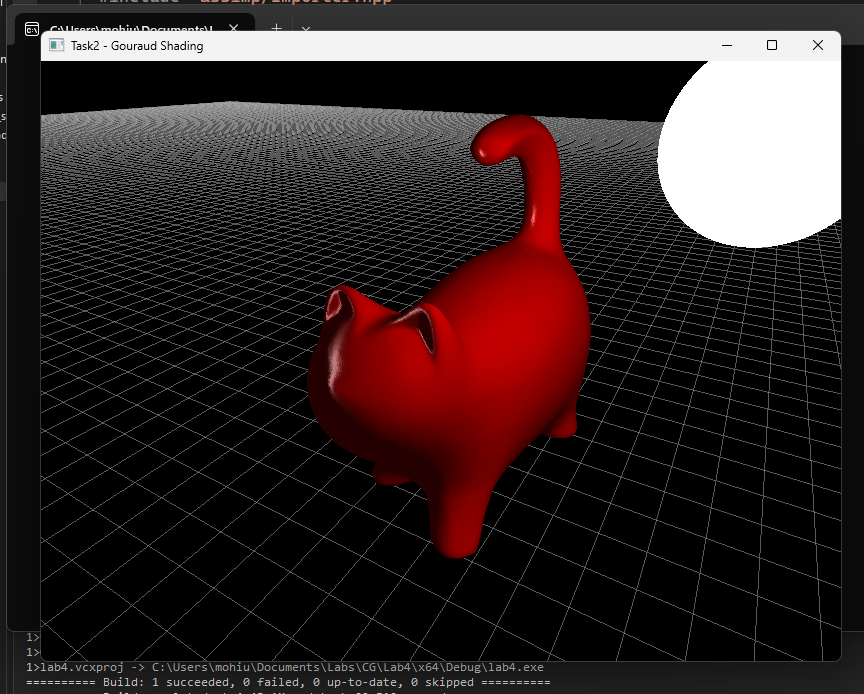
   glutMotionFunc(mouseMotion);

   glutMainLoop();

   return 0;

}

**Output:**

****

Lab Task-03 Implement Phong Shading: Apply Phong shading to the same model and compare the results with Gouraud shading.

**Solution:**

**I had to imlplement Phong Shading using shaders as it was not possible to implement in opengl without them, I searched the internet but couldn’t find a solution.**

**Code:**

#include <GL/glew.h>

#include <GL/freeglut.h>

#include <assimp/Importer.hpp>

#include <assimp/scene.h>

#include <assimp/postprocess.h>

#include <vector>

#include <iostream>

#include <cmath>

#include <glm/glm.hpp>

#include <fstream>

#include <sstream>

#include <string>

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

#include <glm/ext/matrix\_clip\_space.hpp>

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

// Model transformation variables-155, -40, -100,

*float* modelX = -155.0f, modelY = -40.0f, modelZ = -100.0f;      // Model position

*float* modelRotX = -60.0f, modelRotY = 0.0f, modelRotZ = -30.0f; // Model rotation

// Light properties

glm::vec3 lightPos = glm::vec3(170.0f, 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

GLuint shaderProgram;

// Function to read a shader file

std::string readShaderFile(const std::string &*filePath*)

{

    std::ifstream file(*filePath*);

    std::stringstream buffer;

    buffer << file.rdbuf();

    return buffer.str();

}

// Function to compile a shader

GLuint compileShader(GLenum *type*, const std::string &*source*)

{

    GLuint shader = glCreateShader(*type*);

    const *char* \*src = *source*.c\_str();

    glShaderSource(shader, 1, &src, nullptr);

    glCompileShader(shader);

    // Check for errors

    GLint success;

    glGetShaderiv(shader, GL\_COMPILE\_STATUS, &success);

    if (!success)

    {

*char* infoLog[512];

        glGetShaderInfoLog(shader, 512, nullptr, infoLog);

        std::cerr << "Shader compilation error: " << infoLog << std::endl;

        return 0;

    }

    return shader;

}

// Function to create a shader program

GLuint createShaderProgram(const std::string &*vertexShaderSource*, const std::string &*fragmentShaderSource*)

{

    GLuint vertexShader = compileShader(GL\_VERTEX\_SHADER, *vertexShaderSource*);

    GLuint fragmentShader = compileShader(GL\_FRAGMENT\_SHADER, *fragmentShaderSource*);

    GLuint shaderProgram = glCreateProgram();

    glAttachShader(shaderProgram, vertexShader);

    glAttachShader(shaderProgram, fragmentShader);

    glLinkProgram(shaderProgram);

    // Check for linking errors

    GLint success;

    glGetProgramiv(shaderProgram, GL\_LINK\_STATUS, &success);

    if (!success)

    {

*char* infoLog[512];

        glGetProgramInfoLog(shaderProgram, 512, nullptr, infoLog);

        std::cerr << "Shader program linking error: " << infoLog << std::endl;

        return 0;

    }

    glDeleteShader(vertexShader);

    glDeleteShader(fragmentShader);

    return shaderProgram;

}

// 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

        if (mesh->HasNormals())

        {

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

        }

        else

        {

            // If no normals are present, add a default normal (e.g., facing up)

            normals.push\_back(glm::vec3(0.0f, 1.0f, 0.0f));

        }

    }

    // 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;

}

GLuint vao, vbo, ebo, nbo;

*void* renderModel()

{

    // Bind VAO

    glBindVertexArray(vao);

    // Draw the model

    glDrawElements(GL\_TRIANGLES, indices.size(), GL\_UNSIGNED\_INT, 0);

    // Unbind VAO

    glBindVertexArray(0);

}

// 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

    }

    std::cout << lightPos.x << ", " << lightPos.y << ", " << lightPos.z << ", " << 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* display()

{

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

    // Use the shader program for the model

    glUseProgram(shaderProgram);

    // Set up matrices using glm

    glm::mat4 model = glm::mat4(1.0f);

    model = glm::translate(model, glm::vec3(modelX, modelY, modelZ));

    model = glm::rotate(model, glm::radians(modelRotX), glm::vec3(1.0f, 0.0f, 0.0f));

    model = glm::rotate(model, glm::radians(modelRotY), glm::vec3(0.0f, 1.0f, 0.0f));

    model = glm::rotate(model, glm::radians(modelRotZ), glm::vec3(0.0f, 0.0f, 1.0f));

    glm::mat4 view = glm::lookAt(glm::vec3(0.0f, 0.0f, 5.0f), glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3(0.0f, 1.0f, 0.0f));

    glm::mat4 projection = glm::perspective(glm::radians(60.0f), 800.0f / 600.0f, 0.1f, 100000.0f);

    // Pass matrices to shader

    glUniformMatrix4fv(glGetUniformLocation(shaderProgram, "model"), 1, GL\_FALSE, glm::value\_ptr(model));

    glUniformMatrix4fv(glGetUniformLocation(shaderProgram, "view"), 1, GL\_FALSE, glm::value\_ptr(view));

    glUniformMatrix4fv(glGetUniformLocation(shaderProgram, "projection"), 1, GL\_FALSE, glm::value\_ptr(projection));

    // Pass light and material properties

    glUniform3fv(glGetUniformLocation(shaderProgram, "lightPos"), 1, glm::value\_ptr(lightPos));

    glUniform3fv(glGetUniformLocation(shaderProgram, "lightColor"), 1, glm::value\_ptr(lightColor));

    glUniform3fv(glGetUniformLocation(shaderProgram, "viewPos"), 1, glm::value\_ptr(glm::vec3(0.0f, 0.0f, 5.0f)));

    glUniform3fv(glGetUniformLocation(shaderProgram, "matAmbient"), 1, glm::value\_ptr(matAmbient));

    glUniform3fv(glGetUniformLocation(shaderProgram, "matDiffuse"), 1, glm::value\_ptr(matDiffuse));

    glUniform3fv(glGetUniformLocation(shaderProgram, "matSpecular"), 1, glm::value\_ptr(matSpecular));

    glUniform1f(glGetUniformLocation(shaderProgram, "shininess"), shininess);

    // Render the model

    renderModel();

    // Switch to fixed-function pipeline for grid and light

    glUseProgram(0); // Disable shader program

    // Set up the fixed-function pipeline camera

    glMatrixMode(GL\_PROJECTION);

    glLoadIdentity();

    gluPerspective(60.0f, 800.0f / 600.0f, 0.1f, 100000.0f);

    glMatrixMode(GL\_MODELVIEW);

    glLoadIdentity();

    gluLookAt(0.0f, 0.0f, 5.0f,  // Camera position

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

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

    // Render the grid

    drawInfinitePlane();

    // Render the light

    drawLight();

    glutSwapBuffers();

}

// Function to initialize OpenGL settings

*void* init()

{

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

    // Load shaders

    std::string vertexShaderSource = readShaderFile("vertex\_shader.glsl");

    std::string fragmentShaderSource = readShaderFile("fragment\_shader.glsl");

    shaderProgram = createShaderProgram(vertexShaderSource, fragmentShaderSource);

    // Load the model

    if (!loadModel("model.stl"))

    {

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

        exit(1);

    }

    // Create VAO, VBO, and EBO

    glGenVertexArrays(1, &vao);

    glGenBuffers(1, &vbo);

    glGenBuffers(1, &ebo);

    glGenBuffers(1, &nbo);

    glBindVertexArray(vao);

    // Bind and fill VBO with vertex data

    glBindBuffer(GL\_ARRAY\_BUFFER, vbo);

    glBufferData(GL\_ARRAY\_BUFFER, vertices.size() \* sizeof(glm::vec3), vertices.data(), GL\_STATIC\_DRAW);

    // Bind and fill EBO with index data

    glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, ebo);

    glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, indices.size() \* sizeof(*unsigned* *int*), indices.data(), GL\_STATIC\_DRAW);

    // Set up the vertex attribute pointer for vertex positions

    glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, sizeof(glm::vec3), (*void* \*)0);

    glEnableVertexAttribArray(0);

    // Bind the normal buffer object

    glBindBuffer(GL\_ARRAY\_BUFFER, nbo); // Assuming you created a separate buffer for normals

    glBufferData(GL\_ARRAY\_BUFFER, normals.size() \* sizeof(glm::vec3), normals.data(), GL\_STATIC\_DRAW);

    // Set up the vertex attribute pointer for normals

    glVertexAttribPointer(1, 3, GL\_FLOAT, GL\_FALSE, sizeof(glm::vec3), (*void* \*)0);

    glEnableVertexAttribArray(1);

    // Bind the element buffer object (EBO) for indices

    glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, ebo);

    glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, indices.size() \* sizeof(*unsigned* *int*), indices.data(), GL\_STATIC\_DRAW);

    // Unbind the VAO

    glBindVertexArray(0);

}

// Function to handle window resizing

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

{

    glViewport(0, 0, *w*, *h*);

    glMatrixMode(GL\_PROJECTION);

    glLoadIdentity();

    gluPerspective(60.0f, (*float*)*w* / (*float*)*h*, 0.1f, 10000.0f); // Set perspective projection

    glMatrixMode(GL\_MODELVIEW);

}

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

{

    glutInit(&*argc*, *argv*);

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

    // Request a core profile OpenGL context (version 3.3 or later)

    glutInitContextVersion(3, 3);

    glutInitContextProfile(GLUT\_CORE\_PROFILE);

    glutInitWindowSize(800, 600);

    glutCreateWindow("Task2 - Phong Shading");

    // Initialize GLEW (to load modern OpenGL functions)

    glewExperimental = GL\_TRUE;

    if (glewInit() != GLEW\_OK)

    {

        std::cerr << "Failed to initialize GLEW" << std::endl;

        return -1;

    }

    // Check OpenGL version

    std::cout << "OpenGL Version: " << glGetString(GL\_VERSION) << std::endl;

    init(); // Call your initialization function

    glutDisplayFunc(display);

    glutReshapeFunc(reshape);

    glutKeyboardFunc(keyboard);

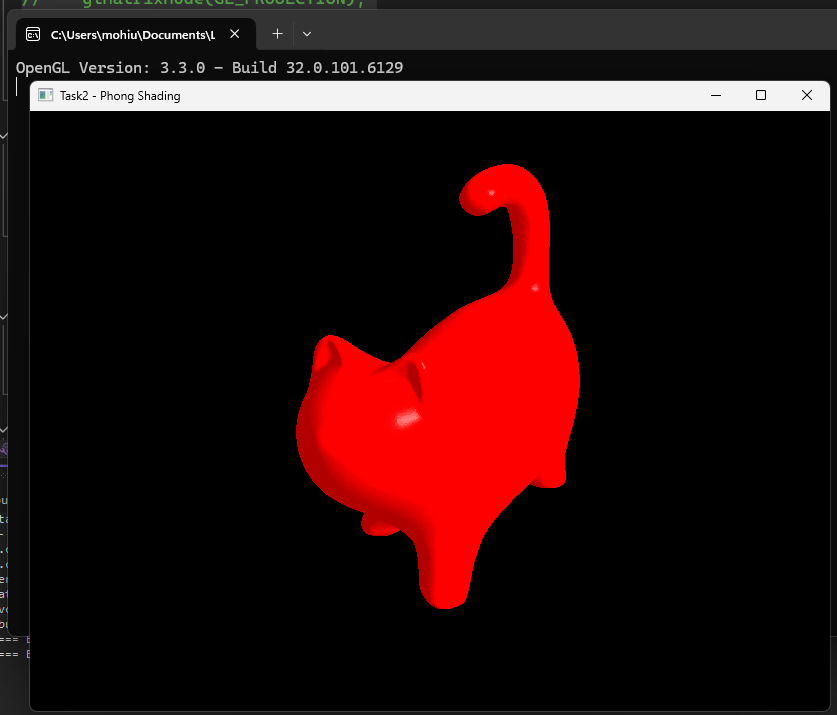
    glutMotionFunc(mouseMotion);

    glutMainLoop();

    return 0;

}

**Output:**



**Comparisons:**

**These are the things I observed:**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Gouraud Shading** | **Phong Shading** |
| **Lighting Calculation** | Per vertex | Per pixel |
| **Normal Interpolation** | None (colors are interpolated) | Normals are interpolated |
| **Highlights** | Misses highlights within triangles | Accurately renders highlights |
| **Performance** | Faster | Slower (due to per-pixel calculations) |
| **Visual Quality** | Less smooth, blocky highlights | Smoother, more accurate highlights |

Lab Task-04 Visualize Lighting Vectors: Change the light direction and observe how it affects shading.

**Solution:**

**I had implemented moving light using WASD and mouse in previous codes so thos codes are repeated here and in files.**

**Code:**

#include <GL/glut.h>

#include <assimp/Importer.hpp>

#include <assimp/scene.h>

#include <assimp/postprocess.h>

#include <vector>

#include <iostream>

#include <cmath>

#include <glm/glm.hpp>

// Model transformation variables-155, -40, -100,

*float* modelX = -155.0f, modelY = -40.0f, modelZ = -100.0f; // Model position

*float* modelRotX = -60.0f, modelRotY = 0.0f, modelRotZ = -30.0f; // Model rotation

// Light properties

glm::vec3 lightPos = glm::vec3(170.0f, 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

   }

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

}

// 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(0.0f, 0.0f, 5.0f,  // Camera position

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

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

   // 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

   // 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(60.0f, (*float*)*w* / (*float*)*h*, 0.1f, 10000.0f); // Set perspective projection

   glMatrixMode(GL\_MODELVIEW);

}

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

   glutInit(&*argc*, *argv*);

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

   glutInitWindowSize(800, 600);

   glutCreateWindow("Task2 - Gouraud Shading");

   init();

   glutDisplayFunc(display);

   glutReshapeFunc(reshape);

   glutKeyboardFunc(keyboard);

   glutMotionFunc(mouseMotion);

   glutMainLoop();

   return 0;

}

#include <GL/glew.h>

#include <GL/freeglut.h>

#include <assimp/Importer.hpp>

#include <assimp/scene.h>

#include <assimp/postprocess.h>

#include <vector>

#include <iostream>

#include <cmath>

#include <glm/glm.hpp>

#include <fstream>

#include <sstream>

#include <string>

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

#include <glm/ext/matrix\_clip\_space.hpp>

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

// Model transformation variables-155, -40, -100,

*float* modelX = -155.0f, modelY = -40.0f, modelZ = -100.0f;      // Model position

*float* modelRotX = -60.0f, modelRotY = 0.0f, modelRotZ = -30.0f; // Model rotation

// Light properties

glm::vec3 lightPos = glm::vec3(170.0f, 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

GLuint shaderProgram;

// Function to read a shader file

std::string readShaderFile(const std::string &*filePath*)

{

    std::ifstream file(*filePath*);

    std::stringstream buffer;

    buffer << file.rdbuf();

    return buffer.str();

}

// Function to compile a shader

GLuint compileShader(GLenum *type*, const std::string &*source*)

{

    GLuint shader = glCreateShader(*type*);

    const *char* \*src = *source*.c\_str();

    glShaderSource(shader, 1, &src, nullptr);

    glCompileShader(shader);

    // Check for errors

    GLint success;

    glGetShaderiv(shader, GL\_COMPILE\_STATUS, &success);

    if (!success)

    {

*char* infoLog[512];

        glGetShaderInfoLog(shader, 512, nullptr, infoLog);

        std::cerr << "Shader compilation error: " << infoLog << std::endl;

        return 0;

    }

    return shader;

}

// Function to create a shader program

GLuint createShaderProgram(const std::string &*vertexShaderSource*, const std::string &*fragmentShaderSource*)

{

    GLuint vertexShader = compileShader(GL\_VERTEX\_SHADER, *vertexShaderSource*);

    GLuint fragmentShader = compileShader(GL\_FRAGMENT\_SHADER, *fragmentShaderSource*);

    GLuint shaderProgram = glCreateProgram();

    glAttachShader(shaderProgram, vertexShader);

    glAttachShader(shaderProgram, fragmentShader);

    glLinkProgram(shaderProgram);

    // Check for linking errors

    GLint success;

    glGetProgramiv(shaderProgram, GL\_LINK\_STATUS, &success);

    if (!success)

    {

*char* infoLog[512];

        glGetProgramInfoLog(shaderProgram, 512, nullptr, infoLog);

        std::cerr << "Shader program linking error: " << infoLog << std::endl;

        return 0;

    }

    glDeleteShader(vertexShader);

    glDeleteShader(fragmentShader);

    return shaderProgram;

}

// 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

        if (mesh->HasNormals())

        {

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

        }

        else

        {

            // If no normals are present, add a default normal (e.g., facing up)

            normals.push\_back(glm::vec3(0.0f, 1.0f, 0.0f));

        }

    }

    // 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;

}

GLuint vao, vbo, ebo, nbo;

*void* renderModel()

{

    // Bind VAO

    glBindVertexArray(vao);

    // Draw the model

    glDrawElements(GL\_TRIANGLES, indices.size(), GL\_UNSIGNED\_INT, 0);

    // Unbind VAO

    glBindVertexArray(0);

}

// 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

    }

    std::cout << lightPos.x << ", " << lightPos.y << ", " << lightPos.z << ", " << 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* display()

{

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

    // Use the shader program for the model

    glUseProgram(shaderProgram);

    // Set up matrices using glm

    glm::mat4 model = glm::mat4(1.0f);

    model = glm::translate(model, glm::vec3(modelX, modelY, modelZ));

    model = glm::rotate(model, glm::radians(modelRotX), glm::vec3(1.0f, 0.0f, 0.0f));

    model = glm::rotate(model, glm::radians(modelRotY), glm::vec3(0.0f, 1.0f, 0.0f));

    model = glm::rotate(model, glm::radians(modelRotZ), glm::vec3(0.0f, 0.0f, 1.0f));

    glm::mat4 view = glm::lookAt(glm::vec3(0.0f, 0.0f, 5.0f), glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3(0.0f, 1.0f, 0.0f));

    glm::mat4 projection = glm::perspective(glm::radians(60.0f), 800.0f / 600.0f, 0.1f, 100000.0f);

    // Pass matrices to shader

    glUniformMatrix4fv(glGetUniformLocation(shaderProgram, "model"), 1, GL\_FALSE, glm::value\_ptr(model));

    glUniformMatrix4fv(glGetUniformLocation(shaderProgram, "view"), 1, GL\_FALSE, glm::value\_ptr(view));

    glUniformMatrix4fv(glGetUniformLocation(shaderProgram, "projection"), 1, GL\_FALSE, glm::value\_ptr(projection));

    // Pass light and material properties

    glUniform3fv(glGetUniformLocation(shaderProgram, "lightPos"), 1, glm::value\_ptr(lightPos));

    glUniform3fv(glGetUniformLocation(shaderProgram, "lightColor"), 1, glm::value\_ptr(lightColor));

    glUniform3fv(glGetUniformLocation(shaderProgram, "viewPos"), 1, glm::value\_ptr(glm::vec3(0.0f, 0.0f, 5.0f)));

    glUniform3fv(glGetUniformLocation(shaderProgram, "matAmbient"), 1, glm::value\_ptr(matAmbient));

    glUniform3fv(glGetUniformLocation(shaderProgram, "matDiffuse"), 1, glm::value\_ptr(matDiffuse));

    glUniform3fv(glGetUniformLocation(shaderProgram, "matSpecular"), 1, glm::value\_ptr(matSpecular));

    glUniform1f(glGetUniformLocation(shaderProgram, "shininess"), shininess);

    // Render the model

    renderModel();

    // Switch to fixed-function pipeline for grid and light

    glUseProgram(0); // Disable shader program

    // Set up the fixed-function pipeline camera

    glMatrixMode(GL\_PROJECTION);

    glLoadIdentity();

    gluPerspective(60.0f, 800.0f / 600.0f, 0.1f, 100000.0f);

    glMatrixMode(GL\_MODELVIEW);

    glLoadIdentity();

    gluLookAt(0.0f, 0.0f, 5.0f,  // Camera position

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

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

    // Render the grid

    drawInfinitePlane();

    // Render the light

    drawLight();

    glutSwapBuffers();

}

// Function to initialize OpenGL settings

*void* init()

{

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

    // Load shaders

    std::string vertexShaderSource = readShaderFile("vertex\_shader.glsl");

    std::string fragmentShaderSource = readShaderFile("fragment\_shader.glsl");

    shaderProgram = createShaderProgram(vertexShaderSource, fragmentShaderSource);

    // Load the model

    if (!loadModel("model.stl"))

    {

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

        exit(1);

    }

    // Create VAO, VBO, and EBO

    glGenVertexArrays(1, &vao);

    glGenBuffers(1, &vbo);

    glGenBuffers(1, &ebo);

    glGenBuffers(1, &nbo);

    glBindVertexArray(vao);

    // Bind and fill VBO with vertex data

    glBindBuffer(GL\_ARRAY\_BUFFER, vbo);

    glBufferData(GL\_ARRAY\_BUFFER, vertices.size() \* sizeof(glm::vec3), vertices.data(), GL\_STATIC\_DRAW);

    // Bind and fill EBO with index data

    glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, ebo);

    glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, indices.size() \* sizeof(*unsigned* *int*), indices.data(), GL\_STATIC\_DRAW);

    // Set up the vertex attribute pointer for vertex positions

    glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, sizeof(glm::vec3), (*void* \*)0);

    glEnableVertexAttribArray(0);

    // Bind the normal buffer object

    glBindBuffer(GL\_ARRAY\_BUFFER, nbo); // Assuming you created a separate buffer for normals

    glBufferData(GL\_ARRAY\_BUFFER, normals.size() \* sizeof(glm::vec3), normals.data(), GL\_STATIC\_DRAW);

    // Set up the vertex attribute pointer for normals

    glVertexAttribPointer(1, 3, GL\_FLOAT, GL\_FALSE, sizeof(glm::vec3), (*void* \*)0);

    glEnableVertexAttribArray(1);

    // Bind the element buffer object (EBO) for indices

    glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, ebo);

    glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, indices.size() \* sizeof(*unsigned* *int*), indices.data(), GL\_STATIC\_DRAW);

    // Unbind the VAO

    glBindVertexArray(0);

}

// Function to handle window resizing

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

{

    glViewport(0, 0, *w*, *h*);

    glMatrixMode(GL\_PROJECTION);

    glLoadIdentity();

    gluPerspective(60.0f, (*float*)*w* / (*float*)*h*, 0.1f, 10000.0f); // Set perspective projection

    glMatrixMode(GL\_MODELVIEW);

}

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

{

    glutInit(&*argc*, *argv*);

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

    // Request a core profile OpenGL context (version 3.3 or later)

    glutInitContextVersion(3, 3);

    glutInitContextProfile(GLUT\_CORE\_PROFILE);

    glutInitWindowSize(800, 600);

    glutCreateWindow("Task2 - Phong Shading");

    // Initialize GLEW (to load modern OpenGL functions)

    glewExperimental = GL\_TRUE;

    if (glewInit() != GLEW\_OK)

    {

        std::cerr << "Failed to initialize GLEW" << std::endl;

        return -1;

    }

    // Check OpenGL version

    std::cout << "OpenGL Version: " << glGetString(GL\_VERSION) << std::endl;

    init(); // Call your initialization function

    glutDisplayFunc(display);

    glutReshapeFunc(reshape);

    glutKeyboardFunc(keyboard);

    glutMotionFunc(mouseMotion);

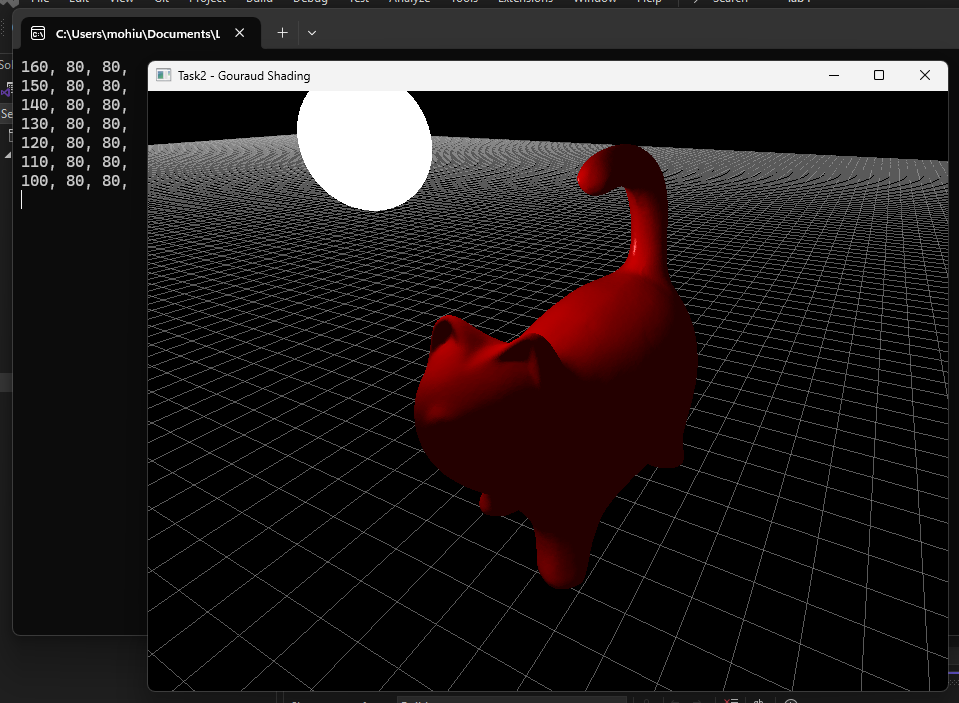
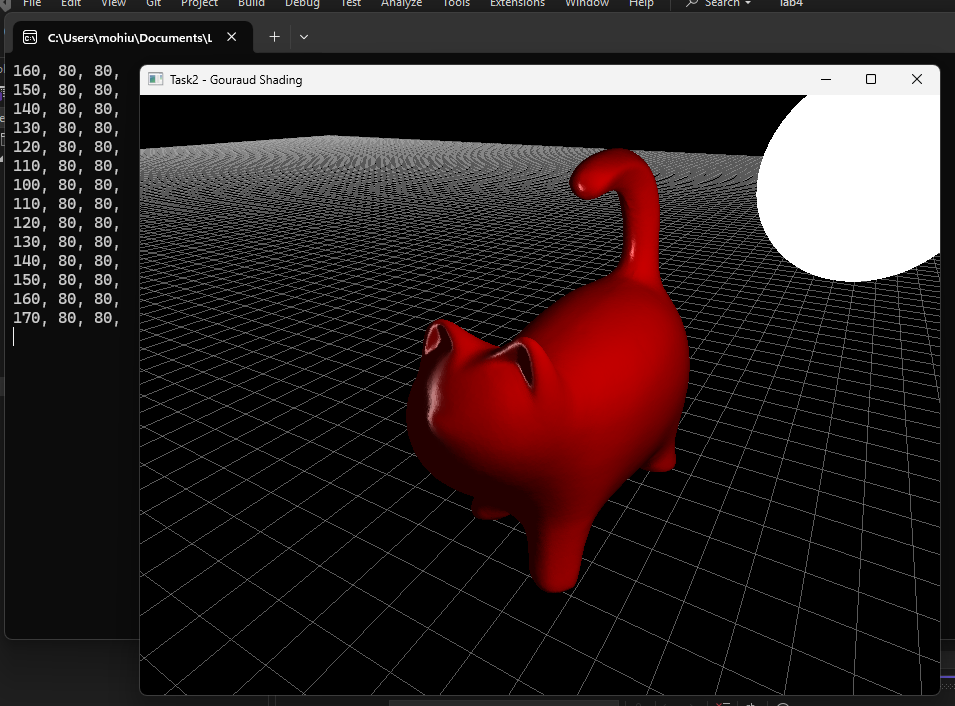
    glutMainLoop();

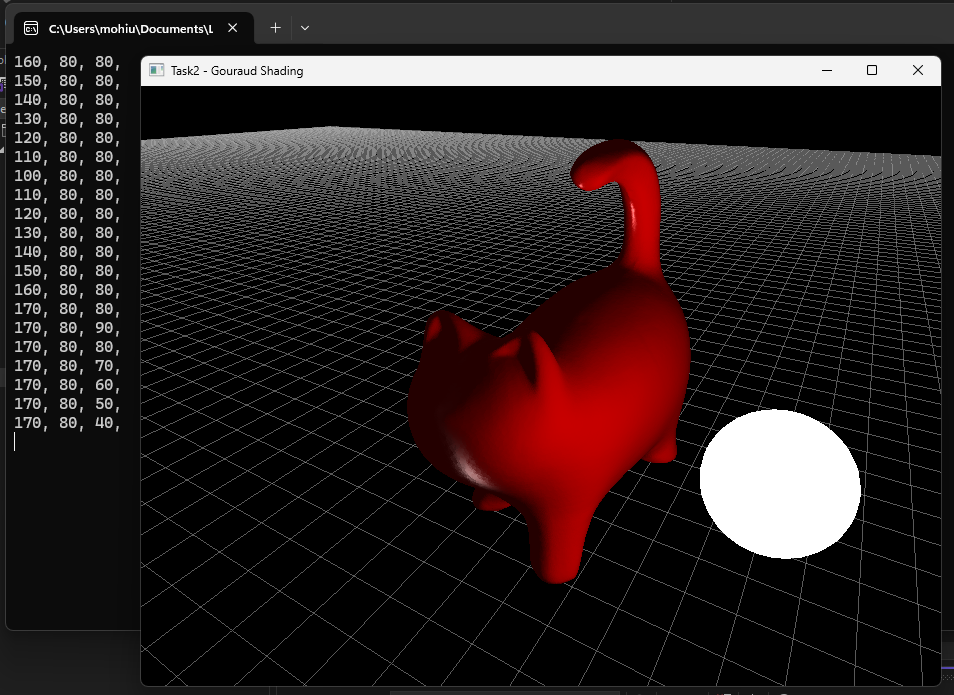
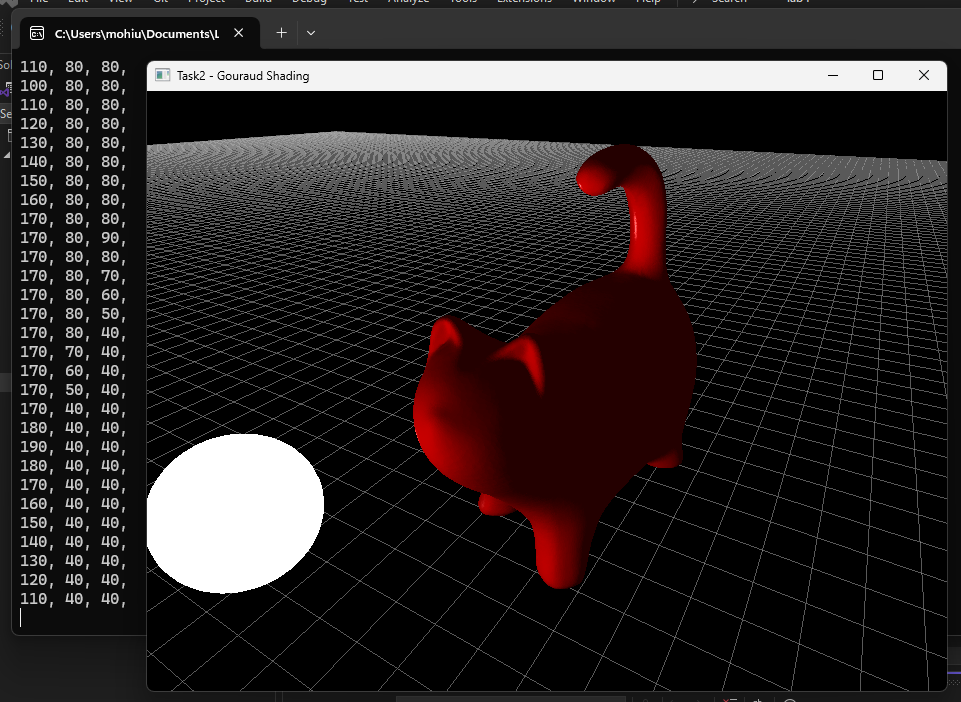
    return 0;

}

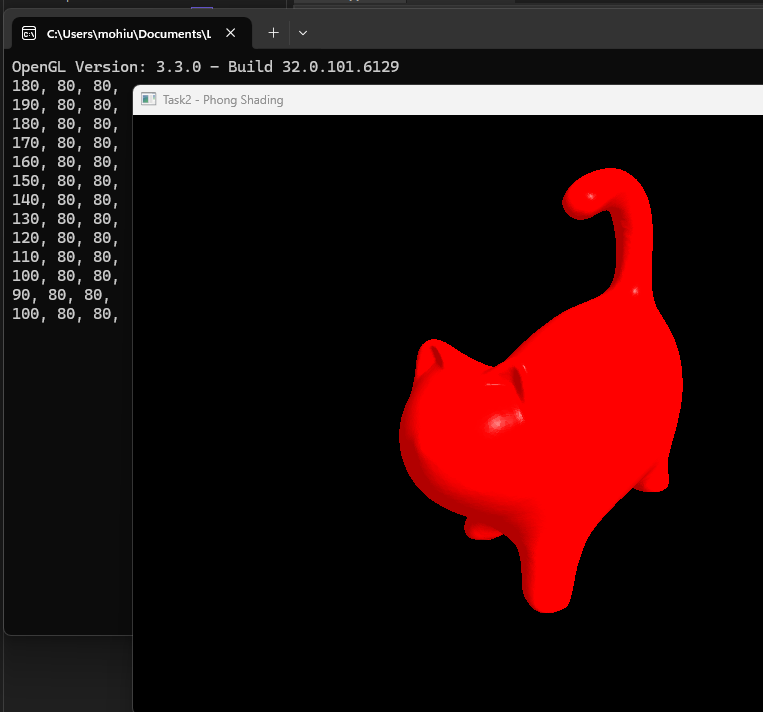
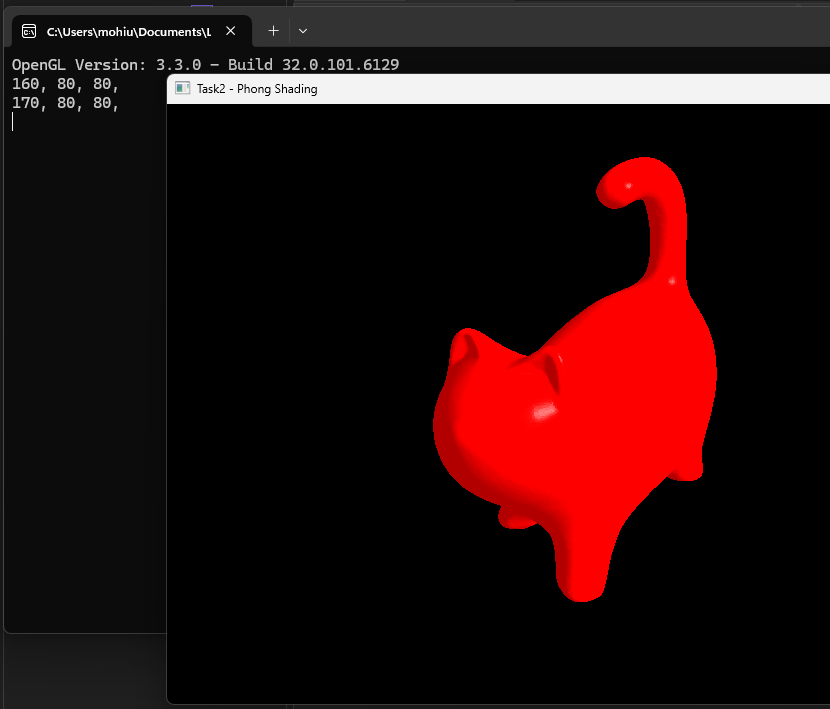
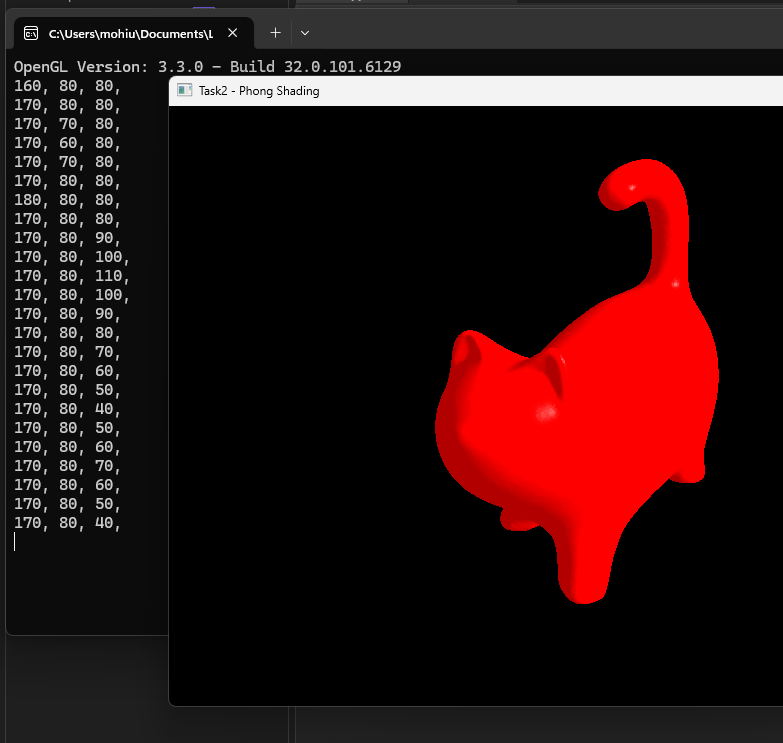
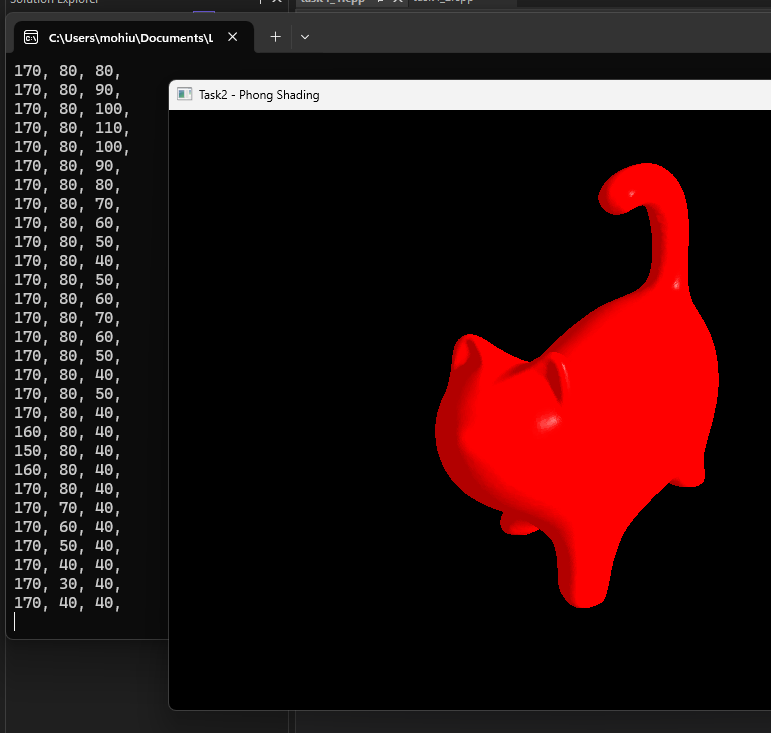
**Output:**

**Gouraud Shading:**

**Phong Shading:**

**** ****  

**What I observed between the movements:**

* **Gouraud Shading:**
  + The model had smooth color transitions, but highlights appeared blocky.
  + Moving the light caused highlights to shift abruptly between vertices.
* **Phong Shading:**
  + The model had smooth highlights and shadows.
  + Moving the light caused highlights to move smoothly across the surface.

### **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 04: Demonstrating Gouraud and Phong Shading & Visualizing Lighting Vectors) | | | | | |
|  | | | | | |
| **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. |