**Department of Computing**

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

**Lab 05: Implementing a Simple Ray Tracer**

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

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

**Date: 25th 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:** 5

**Lab 05: Implementing a Simple Ray Tracer**

### **Lab Objective:**

## The objective of this lab task is to implement a simple ray tracer that casts rays from a virtual camera, computes ray-object intersections, applies basic lighting models (Phong shading and Lambertian reflection), and renders a simple 3D scene with basic shading.

## 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 linear algebra (vectors and matrices).

## Prerequisites :

 Basic understanding of ray tracing concepts.

##  Familiarity with programming (preferably in Python, C++, or Java).  Knowledge of 3D graphics, including vectors, normals, and lighting models.

## Lab Tasks :

Lab Task-01. Set Up the Ray Tracer: Create a basic structure for the ray tracer.

**Solution:**

**Code:**

#include "helpers.h"

*int* main() {

   GLFWwindow \*window = initWindow(std::string("Task1: Basic Raytrace"));

   std::vector<Sphere> spheres;

        spheres.push\_back(Sphere(Vector3(0.0f, 0.0f, -3.0f), 1.0f)); // Center sphere

        spheres.push\_back(Sphere(Vector3(2.0f, 0.0f, -5.0f), 1.0f)); // Right sphere

        spheres.push\_back(Sphere(Vector3(-2.0f, 0.0f, -5.0f), 1.0f)); // Left sphere

   while (!glfwWindowShouldClose(window)) {

       renderBasicRaytraceScene(spheres);

       glfwSwapBuffers(window);

       glfwPollEvents();

   }

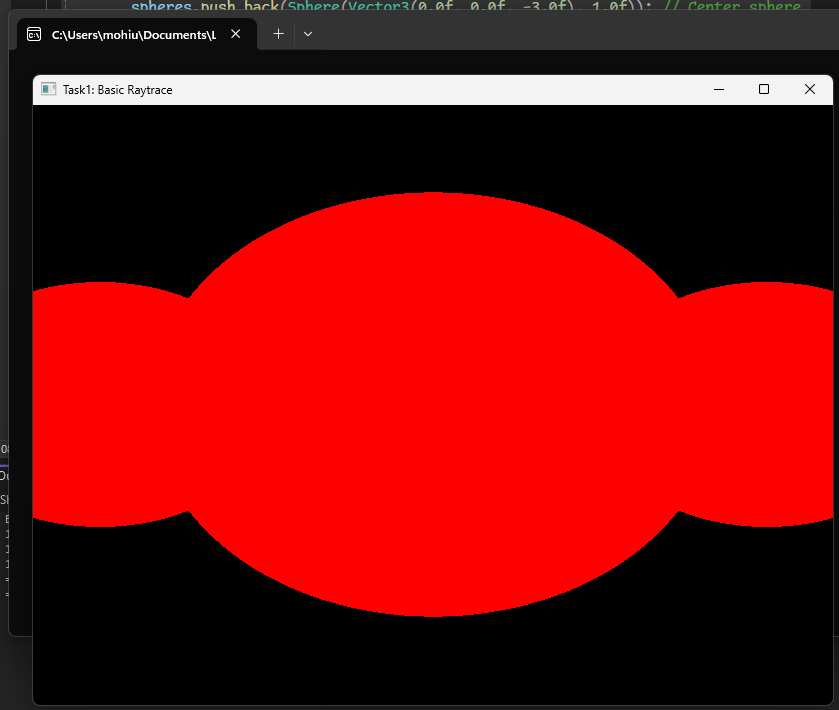
   glfwDestroyWindow(window);

   glfwTerminate();

   return 0;

}

**Output:**

****

Lab Task-02. Implement Ray Casting: Cast rays from a virtual camera into the scene.

**Solution:**

**Code:**

#include "helpers.h"

*int* main() {

    GLFWwindow \*window = initWindow(std::string("Task2: Basic Raycasting"));

    // Define spheres in the scene

    std::vector<Sphere> spheres;

    spheres.push\_back(Sphere(Vector3(0.0f, 0.0f, -3.0f), 1.0f)); // Center sphere

    spheres.push\_back(Sphere(Vector3(2.0f, 0.0f, -5.0f), 1.0f)); // Right sphere

    spheres.push\_back(Sphere(Vector3(-2.0f, 0.0f, -5.0f), 1.0f)); // Left sphere

    while (!glfwWindowShouldClose(window)) {

        renderRacastingScene(spheres);

        glfwSwapBuffers(window);

        glfwPollEvents();

    }

    glfwDestroyWindow(window);

    glfwTerminate();

    return 0;

}

**Output:**

A computer screen shot of a red circle

AI-generated content may be incorrect.

Lab Task-03 Compute Ray-Object Intersections: Implement intersection tests for basic geometric shapes (e.g., spheres, planes).

**Solution:**

**Code:**

#include "helpers.h"

*int* main() {

   GLFWwindow \*window = initWindow(std::string("Task3: Computing Ray-Object Intersections"));

   // Define spheres in the scene

   std::vector<Sphere> spheres;

   spheres.push\_back(Sphere(Vector3(0.0f, 0.0f, -3.0f), 1.0f)); // Center sphere

   spheres.push\_back(Sphere(Vector3(2.0f, 0.0f, -5.0f), 1.0f)); // Right sphere

   spheres.push\_back(Sphere(Vector3(-2.0f, 0.0f, -5.0f), 1.0f)); // Left sphere

   // Checking intersection functions

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

*float* t;

       Ray ray(Vector3(0.0f, 0.0f, 0.0f), Vector3(0.0f, 0.0f, -1.0f));

       if (spheres[i].intersect(ray, t)) {

           std::cout << "Sphere " << i << " intersected at t = " << t << std::endl;

       }

   }

   while (!glfwWindowShouldClose(window)) {

       renderRaycastingScene(spheres);

       glfwSwapBuffers(window);

       glfwPollEvents();

   }

   glfwDestroyWindow(window);

   glfwTerminate();

   return 0;

}

**Output:**

A screenshot of a computer

AI-generated content may be incorrect.

Lab Task-04 Apply Lighting Models: Implement Phong shading and Lambertian reflection.

**Solution:**

**Code:**

#include "helpers.h"

*int* main() {

  GLFWwindow \*window = initWindow(std::string("Task4: Lighting Models"));

  // Define spheres in the scene

  std::vector<Sphere> spheres;

  spheres.push\_back(Sphere(Vector3(0.0f, 0.0f, -3.0f), 1.0f)); // Center sphere

  spheres.push\_back(Sphere(Vector3(2.0f, 0.0f, -5.0f), 1.0f)); // Right sphere

  spheres.push\_back(Sphere(Vector3(-2.0f, 0.0f, -5.0f), 1.0f)); // Left sphere

  // Define planes in the scene

  std::vector<Plane> planes;

  planes.push\_back(Plane(Vector3(0.0f, 1.0f, 0.0f), -1.0f)); // Ground plane

  // Define a light source

  Light light(Vector3(2.0f, 5.0f, 0.0f), Vector3(1.0f, 1.0f, 1.0f));

  while (!glfwWindowShouldClose(window)) {

      // Function to handle keyboard input

      processInput(window, light);

      std::cout << "Light position: " << light.position.x << std::endl;

      renderPhongLambertianRayCastScene(spheres, planes, light);

      glfwSwapBuffers(window);

      glfwPollEvents();

  }

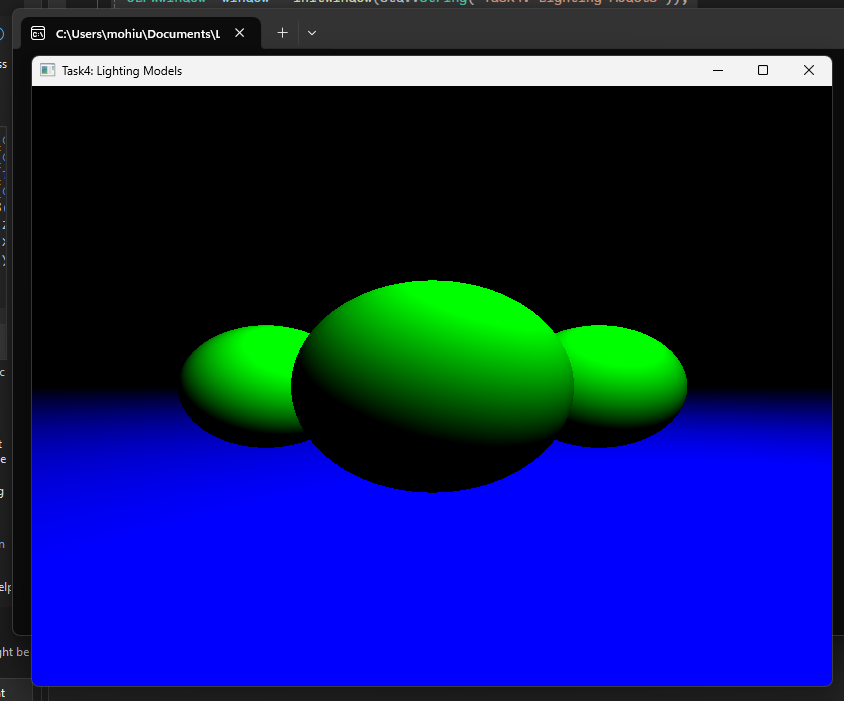
  glfwDestroyWindow(window);

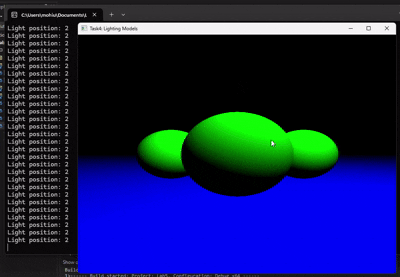
  glfwTerminate();

  return 0;

}

**Output:**





Lab Task-05 Render the Scene: Create a simple 3D scene and render it using the ray tracer.

**Solution:**

**Code:**

#include "helpers.h"

*int* main() {

  GLFWwindow \*window = initWindow(std::string("Task4: Lighting Models"));

  // Define spheres in the scene

  std::vector<Sphere> spheres;

  spheres.push\_back(Sphere(Vector3(0.0f, 0.0f, -3.0f), 1.0f)); // Center sphere

  spheres.push\_back(Sphere(Vector3(2.0f, 0.0f, -5.0f), 1.0f)); // Right sphere

  spheres.push\_back(Sphere(Vector3(-2.0f, 0.0f, -5.0f), 1.0f)); // Left sphere

  spheres.push\_back(Sphere(Vector3(-4.0f, 0.0f, -7.0f), 1.0f)); // Back sphere

  spheres.push\_back(Sphere(Vector3(4.0f, 0.0f, -9.0f), 1.0f)); // Back sphere

  // Define planes in the scene

  std::vector<Plane> planes;

  planes.push\_back(Plane(Vector3(0.0f, 1.0f, 0.0f), -1.0f)); // Ground plane

  // Define a light source

  Light light(Vector3(2.0f, 5.0f, 0.0f), Vector3(1.0f, 1.0f, 1.0f));

  while (!glfwWindowShouldClose(window)) {

      // Function to handle keyboard input

      processInput(window, light);

    std::cout << "Light position: " << light.position.x << std::endl;

      renderPhongLambertianRayCastScene(spheres, planes, light);

      glfwSwapBuffers(window);

      glfwPollEvents();

  }

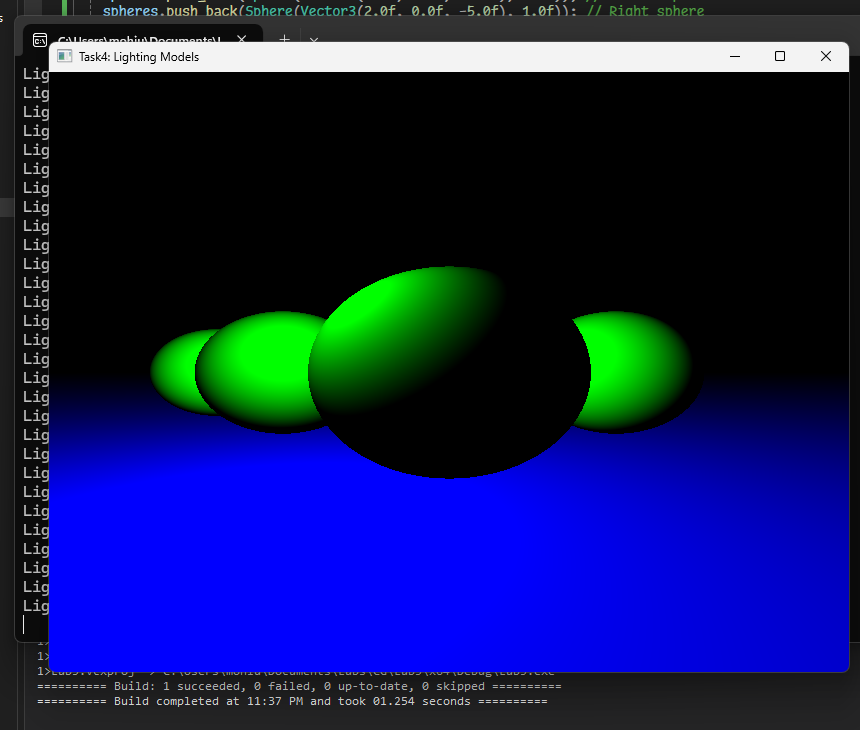
  glfwDestroyWindow(window);

  glfwTerminate();

  return 0;

}

**Output:**



In the last two tasks (4 and 5) the light can be moved with W, S, and arrow keys.

**helpers.h:**

#include <iostream>

#include <GL/glew.h>

#include <GLFW/glfw3.h>

#include <vector>

#include <cmath>

#include "shapes.h"

// Window dimensions

const *int* WIDTH = 800, HEIGHT = 600;

GLFWwindow\* initWindow(std::string *task*) {

    if (!glfwInit()) {

       std::cerr << "Failed to initialize GLFW\n";

       return NULL;

   }

   GLFWwindow\* window = glfwCreateWindow(WIDTH, HEIGHT, task.c\_str(), NULL, NULL);

   if (!window) {

       glfwTerminate();

       std::cerr << "Failed to create GLFW window\n";

       return NULL;

   }

   glfwMakeContextCurrent(window);

   glewExperimental = GL\_TRUE;

   if (glewInit() != GLEW\_OK) {

       std::cerr << "Failed to initialize GLEW\n";

       return NULL;

   }

   glViewport(0, 0, WIDTH, HEIGHT);

   glMatrixMode(GL\_PROJECTION);

   glLoadIdentity();

   glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);

   glMatrixMode(GL\_MODELVIEW);

   glLoadIdentity();

   return window;

}

// Function to render the scene

*void* renderBasicRaytraceScene(const std::vector<Sphere>& *spheres*) {

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

   glBegin(GL\_POINTS);

   for (*int* y = 0; y < HEIGHT; ++y) {

       for (*int* x = 0; x < WIDTH; ++x) {

*float* u = (*float*)x / (*float*)WIDTH;

*float* v = (*float*)y / (*float*)HEIGHT;

           Ray ray(Vector3(0, 0, 0), Vector3(u - 0.5f, v - 0.5f, -1.0f));

*float* t;

*bool* hit = false;

           for (const *auto*& sphere : spheres) {

               if (sphere.intersect(ray, t)) {

                   hit = true;

                   break;

               }

           }

           if (hit) {

               glColor3f(1.0f, 0.0f, 0.0f); // Red color for hit

           }

           else {

               glColor3f(0.0f, 0.0f, 0.0f); // Black color for no hit

           }

           glVertex2f(u, v);

       }

   }

   glEnd();

}

// Function to render the scene using ray casting

*void* renderRaycastingScene(const std::vector<Sphere>& *spheres*) {

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

    glBegin(GL\_POINTS);

    // Virtual camera position

    Vector3 cameraPosition(0.0f, 0.0f, 0.0f);

    // Loop through each pixel on the screen

    for (*int* y = 0; y < HEIGHT; ++y) {

        for (*int* x = 0; x < WIDTH; ++x) {

            // Convert pixel coordinates to normalized device coordinates (-1 to 1)

*float* ndcX = (2.0f \* x) / WIDTH - 1.0f;

*float* ndcY = 1.0f - (2.0f \* y) / HEIGHT;

            // Create a ray from the camera through the pixel

            Vector3 rayDirection(ndcX, ndcY, -1.0f); // Assume the image plane is at z = -1

            Ray ray(cameraPosition, rayDirection);

            // Check for intersections with spheres

*float* t;

*bool* hit = false;

            for (const *auto*& sphere : spheres) {

                if (sphere.intersect(ray, t)) {

                    hit = true;

                    break;

                }

            }

            // Set pixel color based on intersection

            if (hit) {

                glColor3f(1.0f, 0.0f, 0.0f); // Red color for hit

            }

            else {

                glColor3f(0.0f, 0.0f, 0.0f); // Black color for no hit

            }

            glVertex2f((*float*)x / WIDTH, (*float*)y / HEIGHT);

        }

    }

    glEnd();

}

// Function to compute Phong shading

Vector3 phongShading(const Vector3& *point*, const Vector3& *normal*, const Vector3& *viewDir*, const Light& *light*, const Vector3& *kd*, const Vector3& *ks*, *float* *shininess*) {

   // Light direction

   Vector3 lightDir = (light.position - point).normalize();

   // Diffuse component

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

   Vector3 diffuse = light.intensity \* kd \* diff;

   // Specular component

   Vector3 reflectDir = (-lightDir + normal \* 2.0f \* normal.dot(lightDir)).normalize();

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

   Vector3 specular = light.intensity \* ks \* spec;

   // Ambient component (constant)

   Vector3 ambient = light.intensity \* Vector3(0.1f, 0.1f, 0.1f);

   // Combine components

   return ambient + diffuse + specular;

}

// Function to render the scene using ray casting and Phong shading

*void* renderPhongRayCastScene(const std::vector<Sphere>& *spheres*, const std::vector<Plane>& *planes*, const Light& *light*) {

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

   glBegin(GL\_POINTS);

   // Virtual camera position

   Vector3 cameraPosition(0.0f, 0.0f, 0.0f);

   // Loop through each pixel on the screen

   for (*int* y = HEIGHT - 1; y >= 0; --y) { // Start from HEIGHT-1 and go to 0

       for (*int* x = 0; x < WIDTH; ++x) {

           // Convert pixel coordinates to normalized device coordinates (-1 to 1)

*float* ndcX = (2.0f \* x) / WIDTH - 1.0f;

*float* ndcY = 1.0f - (2.0f \* y) / HEIGHT;

           // Create a ray from the camera through the pixel

           Vector3 rayDirection(ndcX, ndcY, -1.0f); // Assume the image plane is at z = -1

           Ray ray(cameraPosition, rayDirection);

           // Variables to store intersection details

*float* t;

           Vector3 color(0.0f, 0.0f, 0.0f); // Default color (black)

*bool* hit = false;

           // Check for intersections with planes

           for (const *auto*& plane : planes) {

               if (plane.intersect(ray, t)) {

                   Vector3 point = ray.origin + ray.direction \* t;

                   Vector3 normal = plane.getNormal();

                   Vector3 viewDir = (cameraPosition - point).normalize();

                   // Apply Phong shading

                   color = phongShading(point, normal, viewDir, light, Vector3(0.2f, 0.8f, 0.2f), Vector3(1.0f, 1.0f, 1.0f), 32.0f);

                   hit = true;

                   break;

               }

           }

           // Check for intersections with spheres

           for (const *auto*& sphere : spheres) {

               if (sphere.intersect(ray, t)) {

                   Vector3 point = ray.origin + ray.direction \* t;

                   Vector3 normal = sphere.getNormal(point);

                   Vector3 viewDir = (cameraPosition - point).normalize();

                   // Apply Phong shading

                   color = phongShading(point, normal, viewDir, light, Vector3(0.8f, 0.2f, 0.2f), Vector3(1.0f, 1.0f, 1.0f), 32.0f);

                   hit = true;

                   break;

               }

           }

           // Set pixel color

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

           glVertex2f((*float*)x / WIDTH, (*float*)(HEIGHT - 1 - y) / HEIGHT); // Flip Y-axis here

       }

   }

   glEnd();

}

// Function to compute Lambertian reflection

Vector3 lambertianReflection(const Vector3& *point*, const Vector3& *normal*, const Light& *light*, const Vector3& *kd*) {

    // Light direction

    Vector3 lightDir = (light.position - point).normalize();

    // Diffuse component

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

    Vector3 diffuse = light.intensity \* kd \* diff;

    return diffuse;

}

// Function to render the scene using ray casting and Phong shading with Lambertian reflection

*void* renderPhongLambertianRayCastScene(const std::vector<Sphere>& *spheres*, const std::vector<Plane>& *planes*, const Light& *light*) {

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

    glBegin(GL\_POINTS);

    // Virtual camera position

    Vector3 cameraPosition(0.0f, 0.0f, 0.0f);

    // Loop through each pixel on the screen

    for (*int* y = HEIGHT - 1; y >= 0; --y) { // Start from HEIGHT-1 and go to 0

        for (*int* x = 0; x < WIDTH; ++x) {

            // Convert pixel coordinates to normalized device coordinates (-1 to 1)

*float* ndcX = (2.0f \* x) / WIDTH - 1.0f;

*float* ndcY = 1.0f - (2.0f \* y) / HEIGHT;

            // Create a ray from the camera through the pixel

            Vector3 rayDirection(ndcX, ndcY, -1.0f); // Assume the image plane is at z = -1

            Ray ray(cameraPosition, rayDirection);

            // Variables to store intersection details

*float* t;

            Vector3 color(0.0f, 0.0f, 0.0f); // Default color (black)

*bool* hit = false;

            // Check for intersections with planes

            for (const *auto*& plane : planes) {

                if (plane.intersect(ray, t)) {

                    Vector3 point = ray.origin + ray.direction \* t;

                    Vector3 normal = plane.getNormal();

                    Vector3 viewDir = (cameraPosition - point).normalize();

                    // Apply Phong shading

                    Vector3 phongColor = phongShading(point, normal, viewDir, light, Vector3(0.2f, 0.8f, 0.2f), Vector3(1.0f, 1.0f, 1.0f), 32.0f);

                    // Apply Lambertian reflection

                    Vector3 lambertianColor = lambertianReflection(point, normal, light, Vector3(0.2f, 0.8f, 0.2f));

                    color = phongColor + lambertianColor;

                    hit = true;

                    break;

                }

            }

            // Check for intersections with spheres

            for (const *auto*& sphere : spheres) {

                if (sphere.intersect(ray, t)) {

                    Vector3 point = ray.origin + ray.direction \* t;

                    Vector3 normal = sphere.getNormal(point);

                    Vector3 viewDir = (cameraPosition - point).normalize();

                    // Apply Phong shading

                    Vector3 phongColor = phongShading(point, normal, viewDir, light, Vector3(0.8f, 0.2f, 0.2f), Vector3(1.0f, 1.0f, 1.0f), 32.0f);

                    // Apply Lambertian reflection

                    Vector3 lambertianColor = lambertianReflection(point, normal, light, Vector3(0.8f, 0.2f, 0.2f));

                    color = phongColor + lambertianColor;

                    hit = true;

                    break;

                }

            }

            // Set pixel color

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

            glVertex2f((*float*)x / WIDTH, (*float*)(HEIGHT - 1 - y) / HEIGHT); // Flip Y-axis here

        }

    }

    glEnd();

}

**shapes.h:**

#include <iostream>

#include <GL/glew.h>

#include <GLFW/glfw3.h>

#include <vector>

#include <cmath>

// Vector3 class for handling 3D vectors

*class* Vector3 {

*public:*

*float* x, y, z;

   Vector3() : x(0), y(0), z(0) {}

   Vector3(*float* *x*, *float* *y*, *float* *z*) : x(x), y(y), z(z) {}

   Vector3 operator+(const Vector3& *v*) const { return Vector3(x + v.x, y + v.y, z + v.z); }

   Vector3 operator-(const Vector3& *v*) const { return Vector3(x - v.x, y - v.y, z - v.z); }

   Vector3 operator-() const { return Vector3(-x, -y, -z); }

   Vector3 operator\*(*float* *f*) const { return Vector3(x \* f, y \* f, z \* f); }

   Vector3 operator\*(const Vector3& *v*) const {

       return Vector3(

           y \* v.z - z \* v.y,

           z \* v.x - x \* v.z,

           x \* v.y - y \* v.x

       );

   }

*float* dot(const Vector3& *v*) const { return x \* v.x + y \* v.y + z \* v.z; }

   Vector3 cross(const Vector3& *v*) const {

       return Vector3(y \* v.z - z \* v.y, z \* v.x - x \* v.z, x \* v.y - y \* v.x);

   }

   Vector3 normalize() const {

*float* mag = sqrt(x \* x + y \* y + z \* z);

       return Vector3(x / mag, y / mag, z / mag);

   }

};

// Ray class

*class* Ray {

*public:*

   Vector3 origin;

   Vector3 direction;

   Ray(const Vector3& *origin*, const Vector3& *direction*) : origin(origin), direction(direction.normalize()) {}

};

// Sphere class

*class* Sphere {

*public:*

   Vector3 center;

*float* radius;

   Sphere(const Vector3& *center*, *float* *radius*) : center(center), radius(radius) {}

*bool* intersect(const Ray& *ray*, *float*& *t*) const {

       Vector3 oc = ray.origin - center;

*float* a = ray.direction.dot(ray.direction);

*float* b = 2.0f \* oc.dot(ray.direction);

*float* c = oc.dot(oc) - radius \* radius;

*float* discriminant = b \* b - 4 \* a \* c;

       if (discriminant < 0) {

           return false;

       }

       else {

           t = (-b - sqrt(discriminant)) / (2.0f \* a);

           return true;

       }

   }

   Vector3 getNormal(const Vector3& *point*) const {

       return (point - center).normalize();

   }

};

// Plane class

*class* Plane {

*public:*

   Vector3 normal;

*float* distance;

   Plane(const Vector3& *normal*, *float* *distance*) : normal(normal.normalize()), distance(distance) {}

*bool* intersect(const Ray& *ray*, *float*& *t*) const {

*float* denom = normal.dot(ray.direction);

       if (abs(denom) > 1e-6) { // Avoid division by zero

           Vector3 p0 = normal \* distance; // A point on the plane

           t = (p0 - ray.origin).dot(normal) / denom;

           return (t >= 0);

       }

       return false;

   }

   Vector3 getNormal() const {

       return normal;

   }

};

// Light class

*class* Light {

*public:*

   Vector3 position;

   Vector3 intensity;

   Light(const Vector3& *position*, const Vector3& *intensity*) : position(position), intensity(intensity) {}

};

// Function to handle keyboard input

*void* processInput(GLFWwindow\* *window*, Light& *light*) {

*float* moveSpeed = 1.0f; // Speed of light movement

    if (glfwGetKey(window, GLFW\_KEY\_UP) == GLFW\_PRESS) {

        light.position.y += moveSpeed; // Move light up

    }

    if (glfwGetKey(window, GLFW\_KEY\_DOWN) == GLFW\_PRESS) {

        light.position.y -= moveSpeed; // Move light down

    }

    if (glfwGetKey(window, GLFW\_KEY\_LEFT) == GLFW\_PRESS) {

        light.position.x -= moveSpeed; // Move light left

    }

    if (glfwGetKey(window, GLFW\_KEY\_RIGHT) == GLFW\_PRESS) {

        light.position.x += moveSpeed; // Move light right

    }

    if (glfwGetKey(window, GLFW\_KEY\_W) == GLFW\_PRESS) {

        light.position.z -= moveSpeed; // Move light forward

    }

    if (glfwGetKey(window, GLFW\_KEY\_S) == GLFW\_PRESS) {

        light.position.z += moveSpeed; // Move light backward

    }

}

### **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 05: Implementing a Simple Ray Tracer) | | | | | |
|  | | | | | |
| **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. |