Name: Ahmed Mohiuddin Shah
CMSID: 415216
Section: BSCS-12-A
Dated: 23-02-2025
Submitted to: Dr. Sidra Sultana

Assignment # 1

Question No. 1: Understanding Lighting Vectors:

Light Source Point $L = \begin{bmatrix} 3 \\ 4 \\ 5 \end{bmatrix}$

Surface Point $P = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$

Light direction vector is from surface to light source:

$$\vec{PL} = L - P = \begin{bmatrix} 3 \\ 4 \\ 5 \end{bmatrix} - \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 3-1 \\ 4-1 \\ 5-1 \end{bmatrix}$$

$$\vec{PL} = \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix}$$

Magnitude of $\vec{PL} = |\vec{PL}| = \sqrt{2^2 + 3^2 + 4^2} = \sqrt{4+9+16}$

$$|\vec{PL}| = \sqrt{29}$$

unit vector of $\vec{PL}$ is $\left(\frac{2}{\sqrt{29}}, \frac{3}{\sqrt{29}}, \frac{4}{\sqrt{29}}\right)$

Light direction vector $\vec{PL} = (2, 3, 4)$

$$|\vec{PL}| = \sqrt{29}$$

unit vector $= \left(\frac{2}{\sqrt{29}}, \frac{3}{\sqrt{29}}, \frac{4}{\sqrt{29}}\right)$

Question 2: Gouraud Shading:

Gouraud shading computes colors at vertices and interpolates the color across the surface of the shape. The process af interpolation occurrs as follows:

1. Compute Vertex Colors:

At each vertex we calculate the color based on lighting calculations depending on normal vector at the vertex, the light source direction and material properties (diffuse, ambient and specular intensity). For the given triangle with vertices $A(1,1,1)$, $B(2,2,2)$, $C(3,1,0)$ and normals $N_A=(0,0,1)$, $N_B=(0,1,0)$ and $N_C=(1,0,0)$ respectively, the lighting calculations would yield colors C_A, C_B and C_C for vertices A, B and C respectively.

2. Interpolate Colors Along Edges:

For each scanline, we interpolate the colors along the edges af the triangle. we find the intersection points af the scanline with the edges af the triangle and interpolate the colors at these intersection points based on the colors at the vertices. Interpolation is typically done through barycentric coordinates or linear interpolation based on the distance along the edge.

for example:

Along edge AB, interpolate C_A and C_B.

Along edge AC, interpolate C_A and C_C.

3. Interpolate colors along the Scanlines:

Interpolate the color between two edge intersection points for each pixel along the scanline. This is done through linear interpolation:

$$C_{pixel} = C_{start} + t \cdot (C_{end} - C_{start})$$

where

C_{start} and C_{end} are colors at start and end af scanline segment and t is the fractional distance af the pixel along the scanline segment.

4. Repeat for all Scanlines:

Repeat for every scanline that intersects the triangle.

Example:

We suppose computed colors at each vertex are:

$C_A(1,0,0)$ (red), $C_B(0,1,0)$ (green), $C_C(0,0,1)$ (Blue).

Interpolate along edge AB and AC:

At midpoint of AB, $C_{AB} = \frac{C_A + C_B}{2} = (0.5, 0.5, 0)$

At midpoint of AC, $C_{AC} = \frac{C_A + C_C}{2} = (0.5, 0, 0.5)$

Interpolate accross scanline:

for a scanline intersecting AB and AC, interpolate between C_{AB} and C_{AC}

and so on...

Question 3: Phong Shading:

Normal Point $N = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$

Light Vector $L = \begin{bmatrix} 0.5 \\ 0.5 \\ 0.5 \end{bmatrix}$

View Vector $V = \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix}$

Reflection vector $R = ?$

using formula $R = 2(N \cdot L)N - L$

$$R = 2\left(\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 0.5 \\ 0.5 \\ 0.5 \end{bmatrix}\right)\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} - \begin{bmatrix} 0.5 \\ 0.5 \\ 0.5 \end{bmatrix}$$

$$= 2(0 + 0 + 0.5)\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} - \begin{bmatrix} 0.5 \\ 0.5 \\ 0.5 \end{bmatrix}$$

$$= 2(0.5)\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} - \begin{bmatrix} 0.5 \\ 0.5 \\ 0.5 \end{bmatrix}$$

$$= 1\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} - \begin{bmatrix} 0.5 \\ 0.5 \\ 0.5 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} - \begin{bmatrix} 0.5 \\ 0.5 \\ 0.5 \end{bmatrix} = \begin{bmatrix} 0 - 0.5 \\ 0 - 0.5 \\ 1 - 0.5 \end{bmatrix}$$

$$R = \begin{bmatrix} -0.5 \\ -0.5 \\ 0.5 \end{bmatrix}$$

Magnitude of $|R| = \sqrt{(-0.5)^2 + (-0.5)^2 + (0.5)^2} = \sqrt{0.25 + 0.25 + 0.25}$.

$$|R| = \sqrt{0.75}$$

unit vector $\hat{R} = \left(\frac{-0.5}{\sqrt{0.75}}, \frac{-0.5}{\sqrt{0.75}}, \frac{0.5}{\sqrt{0.75}}\right)$

So the reflection Vector is:

$$R = (-0.5, -0.5, 0.5)$$

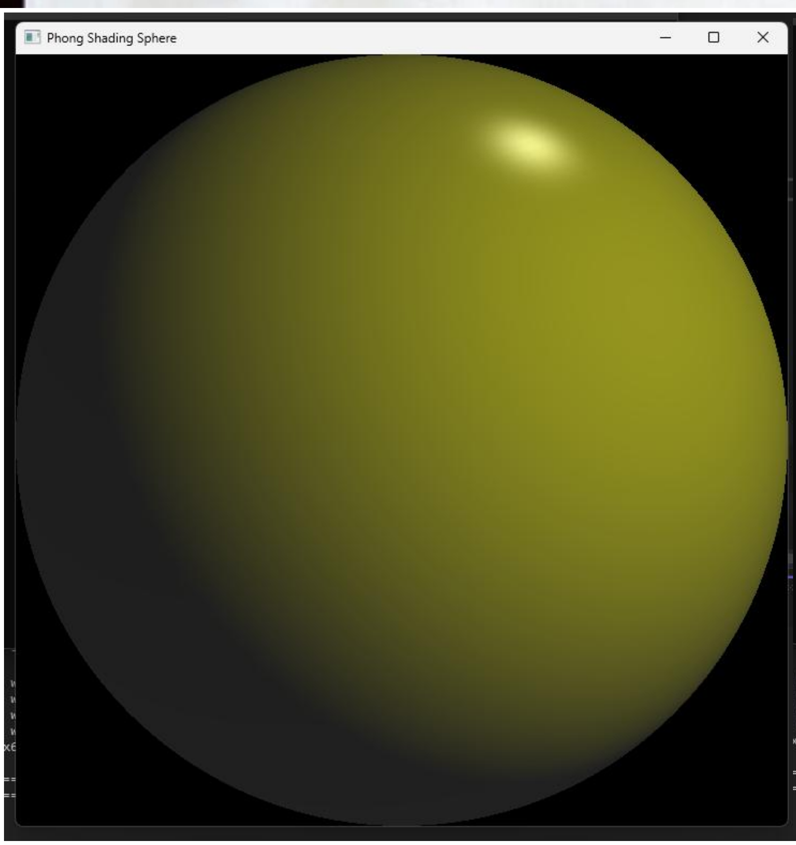
$$|R| = \sqrt{0.75}$$

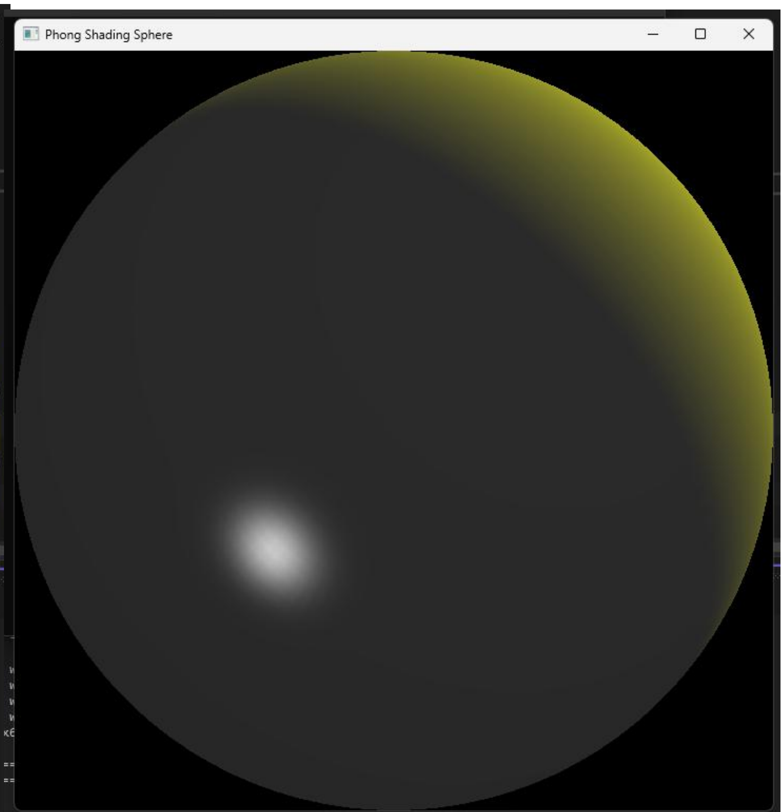
$$\hat{R} = \left(\frac{-0.5}{\sqrt{0.75}}, \frac{-0.5}{\sqrt{0.75}}, \frac{0.5}{\sqrt{0.75}} \right)$$

Question: 4: Coding Implementation:

Code: Code is attached or uploaded with submission.

Output:





Use WASD to rotate the 3D scene.

Bonus Question:

Impact of real-time ray-tracing.

The impact of real time ray tracing instead of using Gouraud or Phong shading is that it enables us to create ~~high~~ highly realistic environments and lighting effects that are accurat to real life such as shadows reflections and global illumination. Traditional techniques like Phong or Gouraud shading rely on approximations but real time ray tracing produces realistic and lifelike visuals with less developer effort by simulating light behaviour by tracing rays and calculating iteractions.

But ray tracing has also a performance impact. ray tracing is computationaly expensive ~~$~~ as it requires tracing and calculating interactions for millions of rays. In contrast, Gouraud and Phong shading are more efficient but struggle with complex lighting effects like soft shadows and realistic reflections.

So Modern Game engines like Unreal Engine 5 use hybrid rendering, combining ray tracing ~~and~~ with rasterization to optimize performance. Hardware advancements, such as Nvidia's RTX GPUs with ~~enhan~~ dedicated ray tracing cores, further enhance feasibility.

```cpp
#include <GL/glut.h>
#include <cmath>
#include <corecrt_math_defines.h>
#include <iostream>

// Light position
GLfloat lightPos[] = { 10.0f, 10.0f, 10.0f, 1.0f };

// Sphere rotation angles
GLfloat angleX = 0.0f;
GLfloat angleY = 0.0f;

// Phong shading parameters
GLfloat ambient[] = { 0.2f, 0.2f, 0.2f, 1.0f };
GLfloat diffuse[] = { 0.8f, 0.8f, 0.0f, 1.0f };
GLfloat specular[] = { 0.8f, 0.8f, 0.8f, 1.0f };
GLfloat shininess = 50.0f;

// Function to normalize a vector
void normalize(float v[3]) {
    float length = sqrt(v[0] * v[0] + v[1] * v[1] + v[2] * v[2]);
    if (length != 0.0f) {
        v[0] /= length;
        v[1] /= length;
        v[2] /= length;
    }
}

// Function to calculate the reflection vector
void reflect(float light[3], float normal[3], float reflection[3]) {
    float dot = light[0] * normal[0] + light[1] * normal[1] + light[2] * normal[2];
    reflection[0] = 2 * dot * normal[0] - light[0];
    reflection[1] = 2 * dot * normal[1] - light[1];
    reflection[2] = 2 * dot * normal[2] - light[2];
}

// Function to calculate Phong shading
void phongShading(float normal[3], float light[3], float view[3], float ambientColor[4],
float diffuseColor[4], float specularColor[4], float shininess, float outputColor[4]) {
    // Ambient component
    outputColor[0] = ambientColor[0];
    outputColor[1] = ambientColor[1];
    outputColor[2] = ambientColor[2];
    outputColor[3] = ambientColor[3];

    // Diffuse component
    float diffuseIntensity = std::max(0.0f, normal[0] * light[0] + normal[1] * light[1] +
normal[2] * light[2]);
    outputColor[0] += diffuseColor[0] * diffuseIntensity;
    outputColor[1] += diffuseColor[1] * diffuseIntensity;
    outputColor[2] += diffuseColor[2] * diffuseIntensity;

    // Specular component
    float reflection[3];
    reflect(light, normal, reflection);
    float specularIntensity = std::pow(std::max(0.0f, reflection[0] * view[0] +
reflection[1] * view[1] + reflection[2] * view[2]), shininess);
    outputColor[0] += specularColor[0] * specularIntensity;
    outputColor[1] += specularColor[1] * specularIntensity;
    outputColor[2] += specularColor[2] * specularIntensity;
}
```

```
// Function to draw a sphere with Phong shading
void drawSphere() {
    int slices = 100;
    int stacks = 100;
    float radius = 1.0f;

    for (int i = 0; i < slices; ++i) {
        float theta1 = i * 2 * M_PI / slices;
        float theta2 = (i + 1) * 2 * M_PI / slices;

        for (int j = 0; j < stacks; ++j) {
            float phi1 = j * M_PI / stacks;
            float phi2 = (j + 1) * M_PI / stacks;

            // Vertices
            float v1[3] = { radius * sin(phi1) * cos(theta1), radius * sin(phi1) * sin(theta1), radius *
cos(phi1) };
            float v2[3] = { radius * sin(phi1) * cos(theta2), radius * sin(phi1) * sin(theta2), radius *
cos(phi1) };
            float v3[3] = { radius * sin(phi2) * cos(theta2), radius * sin(phi2) * sin(theta2), radius *
cos(phi2) };
            float v4[3] = { radius * sin(phi2) * cos(theta1), radius * sin(phi2) * sin(theta1), radius *
cos(phi2) };

            // Normals
            float n1[3] = { v1[0], v1[1], v1[2] };
            float n2[3] = { v2[0], v2[1], v2[2] };
            float n3[3] = { v3[0], v3[1], v3[2] };
            float n4[3] = { v4[0], v4[1], v4[2] };
            normalize(n1);
            normalize(n2);
            normalize(n3);
            normalize(n4);

            // View vector (assuming camera at (0, 0, 5))
            float view[3] = { 0.0f, 0.0f, 5.0f };
            normalize(view);

            // Light vector
            float light[3] = { lightPos[0] - v1[0], lightPos[1] - v1[1], lightPos[2] - v1[2] };
            normalize(light);

            // Calculate Phong shading for each vertex
            float color1[4], color2[4], color3[4], color4[4];
            phongShading(n1, light, view, ambient, diffuse, specular, shininess, color1);
            phongShading(n2, light, view, ambient, diffuse, specular, shininess, color2);
            phongShading(n3, light, view, ambient, diffuse, specular, shininess, color3);
            phongShading(n4, light, view, ambient, diffuse, specular, shininess, color4);

            // Draw the quad
            glBegin(GL_QUADS);
            glColor4fv(color1);
            glVertex3fv(v1);
            glColor4fv(color2);
            glVertex3fv(v2);
            glColor4fv(color3);
            glVertex3fv(v3);
            glColor4fv(color4);
            glVertex3fv(v4);
            glEnd();
        }
    }
}
```

```
// Display function
void display() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    // Set light position
    glLightfv(GL_LIGHT0, GL_POSITION, lightPos);

    // Rotate the sphere
    glRotatef(angleX, 1.0f, 0.0f, 0.0f);
    glRotatef(angleY, 0.0f, 1.0f, 0.0f);

    // Draw the sphere
    drawSphere();

    glutSwapBuffers();
}

// Keyboard function for rotation
void keyboard(unsigned char key, int x, int y) {
    switch (key) {
    case 'd':
        angleX += 5.0f;
        break;
    case 'a':
        angleX -= 5.0f;
        break;
    case 'w':
        angleY += 5.0f;
        break;
    case 's':
        angleY -= 5.0f;
        break;
    }
    glutPostRedisplay();
}

// Main function
int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(720, 720);
    glutCreateWindow("Phong Shading Sphere");

    glEnable(GL_DEPTH_TEST);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_COLOR_MATERIAL);

    glutDisplayFunc(display);
    glutKeyboardFunc(keyboard);

    glutMainLoop();
    return 0;
}
```