2025 ComputerGraphics Hw2

학과	스마트ICT융합공학과
학번	202012336
이름	김동규

1. Objective

-Environment

IDE : VisualStudio2022

Language : C++

Library : GL , GLFW, GLM (etc)

-Objective

Q1. PhongShading

Q2. GammaCorrection

Q3. Antialiasing

2.Code Explanation

-Extended SClass Definition

Q1.PhongShading

*Material

```
class Material {
public:
    vec3 ka, kd, ks;
    float specularPower;
    Material(const vec3& a, const vec3& d, const vec3& s, float sp)
        : ka(a), kd(d), ks(s), specularPower(sp) {
    }
};
```

-A class that defines the material properties of an object.

*Light

```
class Light {
public:
    vec3 position;
    vec3 color;
    Light(const vec3& p, const vec3& c) : position(p), color(c) {}
};
```

-A class that defines light properties.

-Contains variables for light source position and color.

*Surface

```
class Surface {
public:
    Material material;
    vec3 normal;
    Surface(const Material& mat, const vec3& n) : material(mat), normal(n) {}
    virtual bool intersect(const Ray& ray, float& t, float tMin, float tMax) const = 0;
    virtual vec3 getNormal(const vec3& point) const = 0;
    vec3 getColor(const vec3& n, const vec3& l, const vec3& v, const Light& light) {
        return material.ka * light.color + material.kd * light.color * std::max(0.0f, dot(n, l))
            + material.ks * light.color * std::pow(std::max(0.0f, dot(2.0f * dot(n, l) * n - l, v)), material.specularPower);
    }
    vec3 shade(const Ray& ray, vec3 point, vec3 n, Light light, bool intersect) {
        vec3 v = -normalize(ray.direction);
        vec3 l = normalize(light.position - point);
        if (intersect) {
            return material.ka * light.color;
        }
        vec3 color = getColor(n, l, v, light);
        return color;
    }
};
```

-Added material to the surface class.

-Introduced a normal variable in the Surface class, which is inherited by both Plane and Sphere.

*Sphere

```
class Sphere : public Surface {
public:
    vec3 center;
    float radius;
    Sphere(const vec3& c, float r, const Material& mat)
        : Surface(mat, vec3(0.0f)), center(c), radius(r) {
    }

    bool intersect(const Ray& ray, float& t, float tMin, float tMax) const override {
        vec3 oc = ray.origin - center;
        float a = dot(ray.direction, ray.direction);
        float b = dot(oc, ray.direction);
        float c = dot(oc, oc) - radius * radius;
        float discriminant = b * b - a * c;

        if (discriminant > 0) {
            float temp = (-b - std::sqrt(discriminant)) / a;
            if (temp < tMax && temp > tMin) {
                t = temp;
                return true;
            }
        }
        return false;
    }

    vec3 getNormal(const vec3& point) const override {
        return normalize(point - center);
    }
};
```

-Added the getNormal function to return the normal vector at each point.

*Plane

```
class Plane : public Surface {
public:
    float d;//plane equation: n.x + n.y + n.z + d = 0

    Plane(const vec3& n, float d, const Material& mat)
        : Surface(mat, normalize(n)), d(d) {
    }

    bool intersect(const Ray& ray, float& t, float tMin, float tMax) const override {
        float denom = dot(normal, ray.direction);
        if (abs(denom) > 1e-6) { // Ensure the ray is not parallel to the plane
            t = (d - dot(normal, ray.origin)) / denom;
            return (t >= tMin && t <= tMax);
        }
        return false;
    }

    vec3 getNormal(const vec3& point) const override {
        return normal;
    }
};
```

-Added the getNormal function to return the normal vector.

*Scene

```
class Scene {
public:
    std::vector<Surface*> surfaces;
    Camera camera;
    std::vector<Light*> lights;

    Scene(const std::vector<Surface*>& sur, const Camera& c, const std::vector<Light*>& l)
        : surfaces(sur), camera(c), lights(l) {
    }

    bool traceShadow(const Ray& shadowRay, float tMin, float tMax) {
        for (const auto& surface : surfaces) {
            float tempT;
            if (surface->intersect(shadowRay, tempT, tMin, tMax)) {
                return true;
            }
        }
        return false;
    }

    vec3 trace(const Ray& ray, float tMin, float tMax) {
        float t = tMax;
        Surface* hitSurface = nullptr;
        for (const auto& surface : surfaces) {
            float tempT;
            if (surface->intersect(ray, tempT, tMin, tMax) && tempT < t) {
                t = tempT;
                hitSurface = surface;
            }
        }
        vec3 hitPoint = ray.origin + ray.direction * t;
        Ray shadowRay = Ray(hitPoint, normalize(lights[0]->position - hitPoint));
        float lightDistance = length(lights[0]->position - hitPoint);
        bool isShadow = traceShadow(shadowRay, 0.001f, lightDistance);
        return (hitSurface != nullptr) ? hitSurface->shade(ray, hitPoint, hitSurface->getNormal(hitPoint), *lights[0], isShadow) : vec3(0.0f, 0.0f, 0.0f); //if null -> bl
    }
};
```

-Modified the trace function to handle shadows.

-Added the traceShadow function to determine whether a surface point is in shadow by

-checking if the ray from the point toward the light source is blocked by another object.

*render function

```
void render() {
    OutputImage.resize(Width * Height * 3, 1.0f);
    Material planeMat(vec3(0.2f, 0.2f, 0.2f), vec3(1.0f, 1.0f, 1.0f), vec3(0.0f, 0.0f, 0.0f), 0);
    Material sphere1Mat(vec3(0.2f, 0, 0), vec3(1.0f, 0, 0), vec3(0, 0, 0), 0);
    Material sphere2Mat(vec3(0, 0.2f, 0), vec3(0, 0.5f, 0), vec3(0.5f), 32);
    Material sphere3Mat(vec3(0, 0, 0.2f), vec3(0, 0, 1.0f), vec3(0, 0, 0), 0);

    Plane plane(vec3(0.0f, 1.0f, 0.0f), -2.0f, planeMat);
    Sphere sphere1(vec3(-4.0f, 0.0f, -7.0f), 1.0f, sphere1Mat);
    Sphere sphere2(vec3(0.0f, 0.0f, -7.0f), 2.0f, sphere2Mat);
    Sphere sphere3(vec3(4.0f, 0.0f, -7.0f), 1.0f, sphere3Mat);

    std::vector<Surface*> surfaces = { &sphere1, &sphere2, &sphere3, &plane };

    Camera camera(vec3(0.0f, 0.0f, 0.0f), vec3(0.0f, 0.0f, -1.0f), 0.1f);

    Light light(vec3(-4.0f, 4.0f, -3.0f), vec3(1.0f, 1.0f, 1.0f));
    std::vector<Light*> lights = { &light };

    Scene scene(surfaces, camera, lights);
    ImagePlane imagePlane;

    for (int iy = 0; iy < Height; ++iy) {
        for (int ix = 0; ix < Width; ++ix) {
            float x = (0.2f * (ix + 0.5f)) / Width - 0.1f;
            float y = (0.2f * (iy + 0.5f)) / Height - 0.1f;
            Ray ray = scene.camera.getRay(x, y);
            vec3 color = scene.trace(ray, 0.0f, FLT_MAX);
            imagePlane.set(ix, iy, color);
        }
    }
}
```

-Initialized objects using the Material class.

-Created a light vector to manage multiple light sources.

-Declared Light objects.

Q2.GammaCorrection

*render function

```
void render() {
    OutputImage.resize(Width * Height * 3, 1.0f);
    Material planeMat(vec3(0.2f, 0.2f, 0.2f), vec3(1.0f, 1.0f, 1.0f), vec3(0.0f, 0.0f, 0.0f), 0);
    Material sphere1Mat(vec3(0.2f, 0, 0), vec3(1.0f, 0, 0), vec3(0, 0, 0), 0);
    Material sphere2Mat(vec3(0, 0.2f, 0), vec3(0, 0.5f, 0), vec3(0.5f,0.5f,0.5f), 32);
    Material sphere3Mat(vec3(0, 0, 0.2f), vec3(0, 0, 1.0f), vec3(0, 0, 0), 0);

    Plane plane(vec3(0.0f, 1.0f, 0.0f), -2.0f, planeMat);
    Sphere sphere1(vec3(-4.0f, 0.0f, -7.0f), 1.0f, sphere1Mat);
    Sphere sphere2(vec3(0.0f, 0.0f, -7.0f), 2.0f, sphere2Mat);
    Sphere sphere3(vec3(4.0f, 0.0f, -7.0f), 1.0f, sphere3Mat);

    std::vector<Surface*> surfaces = { &sphere1, &sphere2, &sphere3, &plane };

    Camera camera(vec3(0.0f, 0.0f, 0.0f), vec3(0.0f, 0.0f, -1.0f), 0.1f);

    Light light(vec3(-4.0f, 4.0f, -3.0f), vec3(1.0f, 1.0f, 1.0f));
    std::vector<Light*> lights = { &light };

    Scene scene(surfaces, camera, lights);
    ImagePlane imagePlane;
    float gamma = 2.2f; // gamma
    float invGamma = 1.0f / gamma; // 1/γ

    for (int iy = 0; iy < Height; ++iy) {
        for (int ix = 0; ix < Width; ++ix) {
            float x = (0.2f * (ix + 0.5f)) / Width - 0.1f;
            float y = (0.2f * (iy + 0.5f)) / Height - 0.1f;
            Ray ray = scene.camera.getRay(x, y);
            vec3 color = scene.trace(ray, 0.0f, FLT_MAX);

            color = pow(color, vec3(invGamma));
            imagePlane.set(ix, iy, color);
        }
    }
}
```

-The rest of the classes remain unchanged.

-Added a gamma variable in the render function to implement gamma correction.

Q3.Antialiasing

*render function

```
void render() {
    OutputImage.resize(Width * Height * 3, 1.0f);
    Material planeMat(vec3(0.2f, 0.2f, 0.2f), vec3(1.0f, 1.0f, 1.0f), vec3(0.0f, 0.0f, 0.0f), 0);
    Material sphere1Mat(vec3(0.2f, 0, 0), vec3(1.0f, 0, 0), vec3(0, 0, 0), 0);
    Material sphere2Mat(vec3(0, 0.2f, 0), vec3(0, 0.5f, 0), vec3(0.5f, 0.5f, 0.5f), 32);
    Material sphere3Mat(vec3(0, 0, 0.2f), vec3(0, 0, 1.0f), vec3(0, 0, 0), 0);

    Plane plane(vec3(0.0f, 1.0f, 0.0f), -2.0f, planeMat);
    Sphere sphere1(vec3(-4.0f, 0.0f, -7.0f), 1.0f, sphere1Mat);
    Sphere sphere2(vec3(0.0f, 0.0f, -7.0f), 2.0f, sphere2Mat);
    Sphere sphere3(vec3(4.0f, 0.0f, -7.0f), 1.0f, sphere3Mat);

    std::vector<Surface*> surfaces = { &sphere1, &sphere2, &sphere3, &plane };

    Camera camera(vec3(0.0f, 0.0f, 0.0f), vec3(0.0f, 0.0f, -1.0f), 0.1f);

    Light light(vec3(-4.0f, 4.0f, -3.0f), vec3(1.0f, 1.0f, 1.0f));
    std::vector<Light*> lights = { &light };

    Scene scene(surfaces, camera, lights);
    ImagePlane imagePlane;
    float gamma = 2.2f; // gamma
    float invGamma = 1.0f / gamma;// 1/γ

    int samples = 64;  //number of supersampling 64
    float invSamples = 1.0f / float(samples);

    for (int iy = 0; iy < Height; ++iy) {
        for (int ix = 0; ix < Width; ++ix) {
            vec3 color(0.0f);

            // create multiple rays for supersampling 64
            for (int s = 0; s < samples; ++s) {
                float x = (0.2f * (ix + static_cast<float>(rand()) / RAND_MAX)) / Width - 0.1f;
                float y = (0.2f * (iy + static_cast<float>(rand()) / RAND_MAX)) / Height - 0.1f;

                Ray ray = scene.camera.getRay(x, y);
                color += scene.trace(ray, 0.0f, FLT_MAX);
            }

            color *= invSamples;  //calculate average color
            color = pow(color, vec3(invGamma));  //gamma correction
            imagePlane.set(ix, iy, color);
        }
    }
}
```

-The rest of the classes remain unchanged.

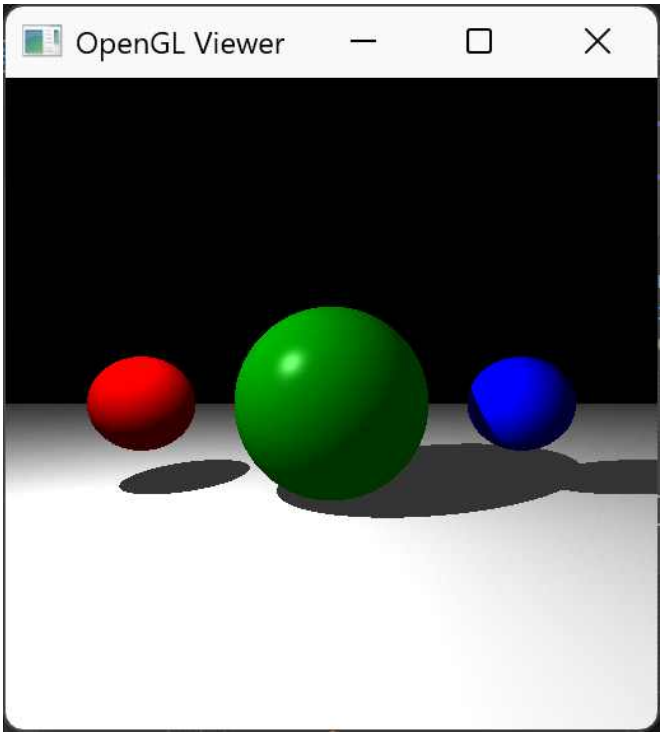
-Set the number of samples per pixel.

-Generated random rays within each pixel.

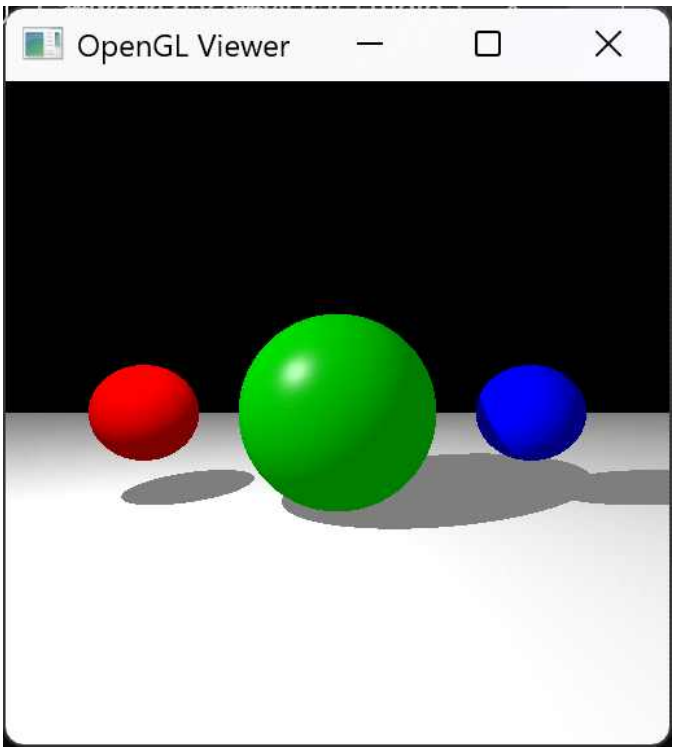
-Averaged the resulting colors to reduce aliasing artifacts.

3. Result screenshots

Q1. PhongShading



Q2.GammaCorrection



Q3.Antialiasing