Akash_2021009
CSE 333/533 - Monsoon 2023
Assignment 4: Raytracing

1. Extend the Object class to implement a Triangle that can be rendered in the raytracer.

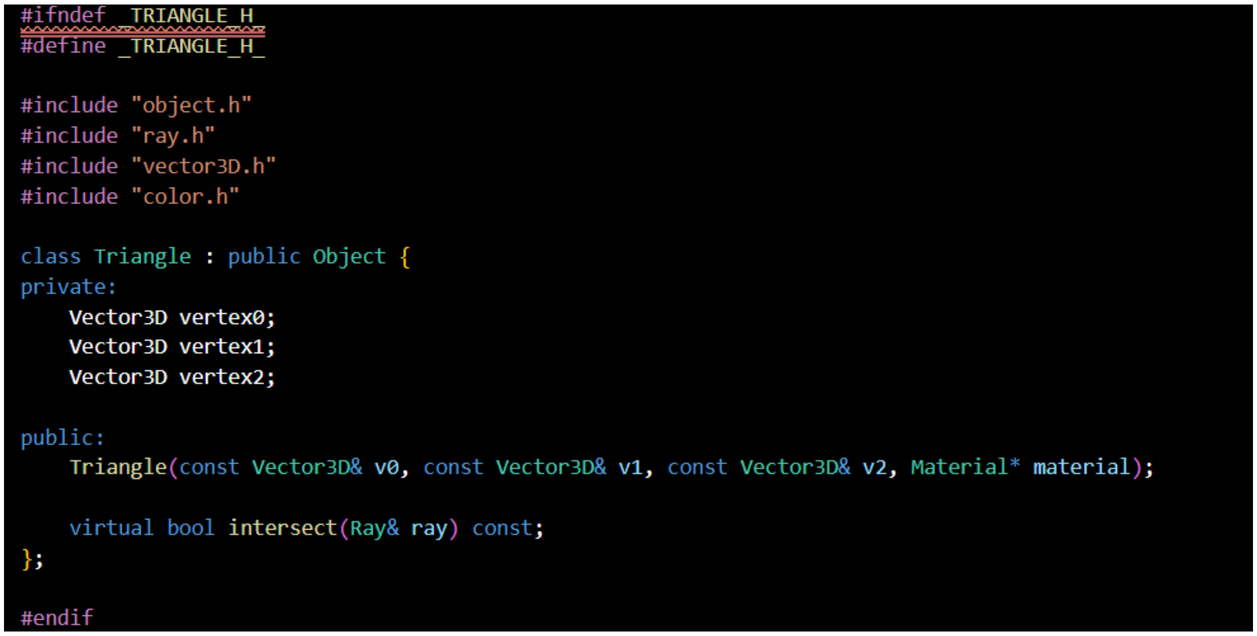
A ray in a 3D space is typically represented by an origin point and a direction vector. The origin represents the starting point of the ray, while the direction vector points along the ray's path.
A triangle in 3D space is defined by three vertices (vertex0, vertex1, and vertex2).

Cramer's rule is applied to the system of linear equations generated by the ray and the triangle. The system is typically represented as follows:

*A * [beta, gamma, t] = [P - V0]*

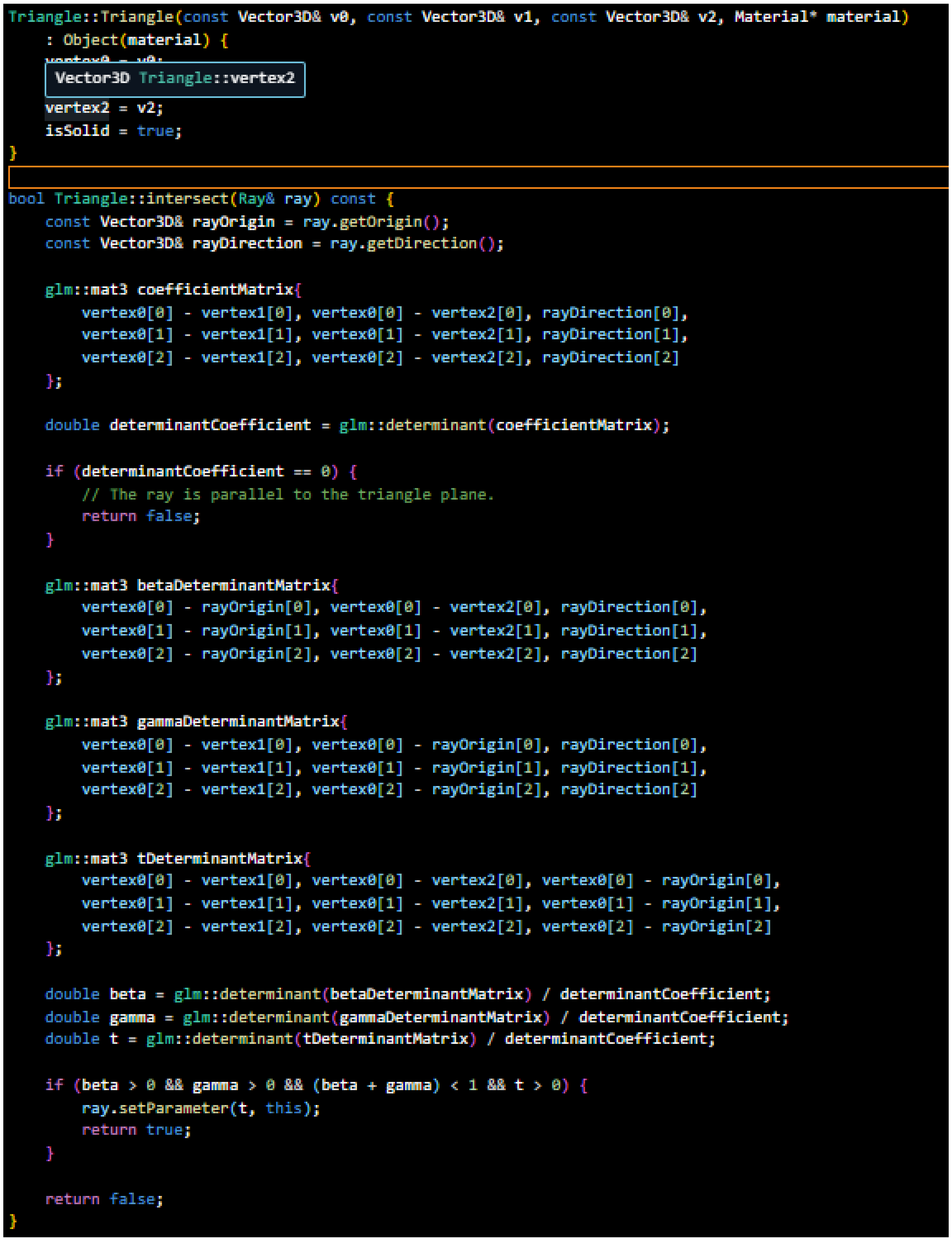
A is a matrix that holds coefficients.
[beta, gamma, t] represents the unknowns to be solved.
[P - V0] is a vector formed by subtracting the ray's origin P from one of the triangle's vertices V0.

//triangles.h



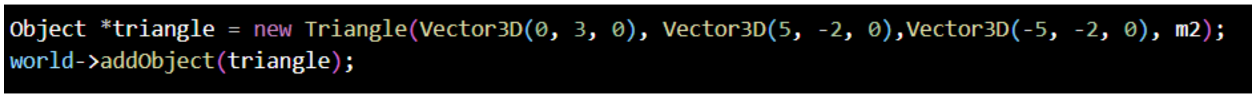
```
#ifndef _TRIANGLE_H_
#define _TRIANGLE_H_

#include "object.h"
#include "ray.h"
#include "vector3D.h"
#include "color.h"

class Triangle : public Object {
private:
    Vector3D vertex0;
    Vector3D vertex1;
    Vector3D vertex2;

public:
    Triangle(const Vector3D& v0, const Vector3D& v1, const Vector3D& v2, Material* material);

    virtual bool intersect(Ray& ray) const;
};

#endif
```

//triangle.cpp



```cpp
Triangle::Triangle(const Vector3D& v0, const Vector3D& v1, const Vector3D& v2, Material* material)
    : Object(material) {
    vertex0 = v0;
    Vector3D Triangle::vertex2
    vertex2 = v2;
    isSolid = true;
}

bool Triangle::intersect(Ray& ray) const {
    const Vector3D& rayOrigin = ray.getOrigin();
    const Vector3D& rayDirection = ray.getDirection();

    glm::mat3 coefficientMatrix{
        vertex0[0] - vertex1[0], vertex0[0] - vertex2[0], rayDirection[0],
        vertex0[1] - vertex1[1], vertex0[1] - vertex2[1], rayDirection[1],
        vertex0[2] - vertex1[2], vertex0[2] - vertex2[2], rayDirection[2]
    };

    double determinantCoefficient = glm::determinant(coefficientMatrix);

    if (determinantCoefficient == 0) {
        // The ray is parallel to the triangle plane.
        return false;
    }

    glm::mat3 betaDeterminantMatrix{
        vertex0[0] - rayOrigin[0], vertex0[0] - vertex2[0], rayDirection[0],
        vertex0[1] - rayOrigin[1], vertex0[1] - vertex2[1], rayDirection[1],
        vertex0[2] - rayOrigin[2], vertex0[2] - vertex2[2], rayDirection[2]
    };

    glm::mat3 gammaDeterminantMatrix{
        vertex0[0] - vertex1[0], vertex0[0] - rayOrigin[0], rayDirection[0],
        vertex0[1] - vertex1[1], vertex0[1] - rayOrigin[1], rayDirection[1],
        vertex0[2] - vertex1[2], vertex0[2] - rayOrigin[2], rayDirection[2]
    };

    glm::mat3 tDeterminantMatrix{
        vertex0[0] - vertex1[0], vertex0[0] - vertex2[0], vertex0[0] - rayOrigin[0],
        vertex0[1] - vertex1[1], vertex0[1] - vertex2[1], vertex0[1] - rayOrigin[1],
        vertex0[2] - vertex1[2], vertex0[2] - vertex2[2], vertex0[2] - rayOrigin[2]
    };

    double beta = glm::determinant(betaDeterminantMatrix) / determinantCoefficient;
    double gamma = glm::determinant(gammaDeterminantMatrix) / determinantCoefficient;
    double t = glm::determinant(tDeterminantMatrix) / determinantCoefficient;

    if (beta > 0 && gamma > 0 && (beta + gamma) < 1 && t > 0) {
        ray.setParameter(t, this);
        return true;
    }

    return false;
}
```

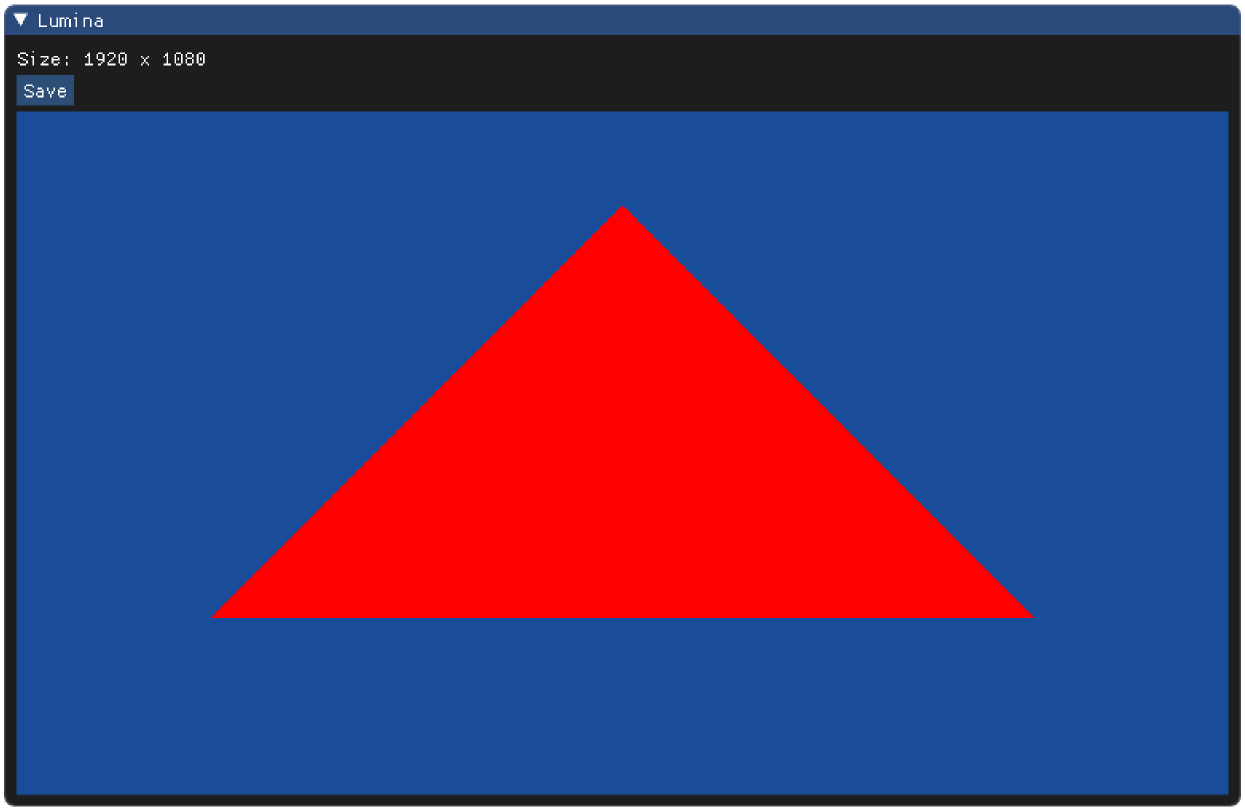
//updating main file to render triangle



```cpp
Object *triangle = new Triangle(Vector3D(0, 3, 0), Vector3D(5, -2, 0),Vector3D(-5, -2, 0), m2);
world->addObject(triangle);
```

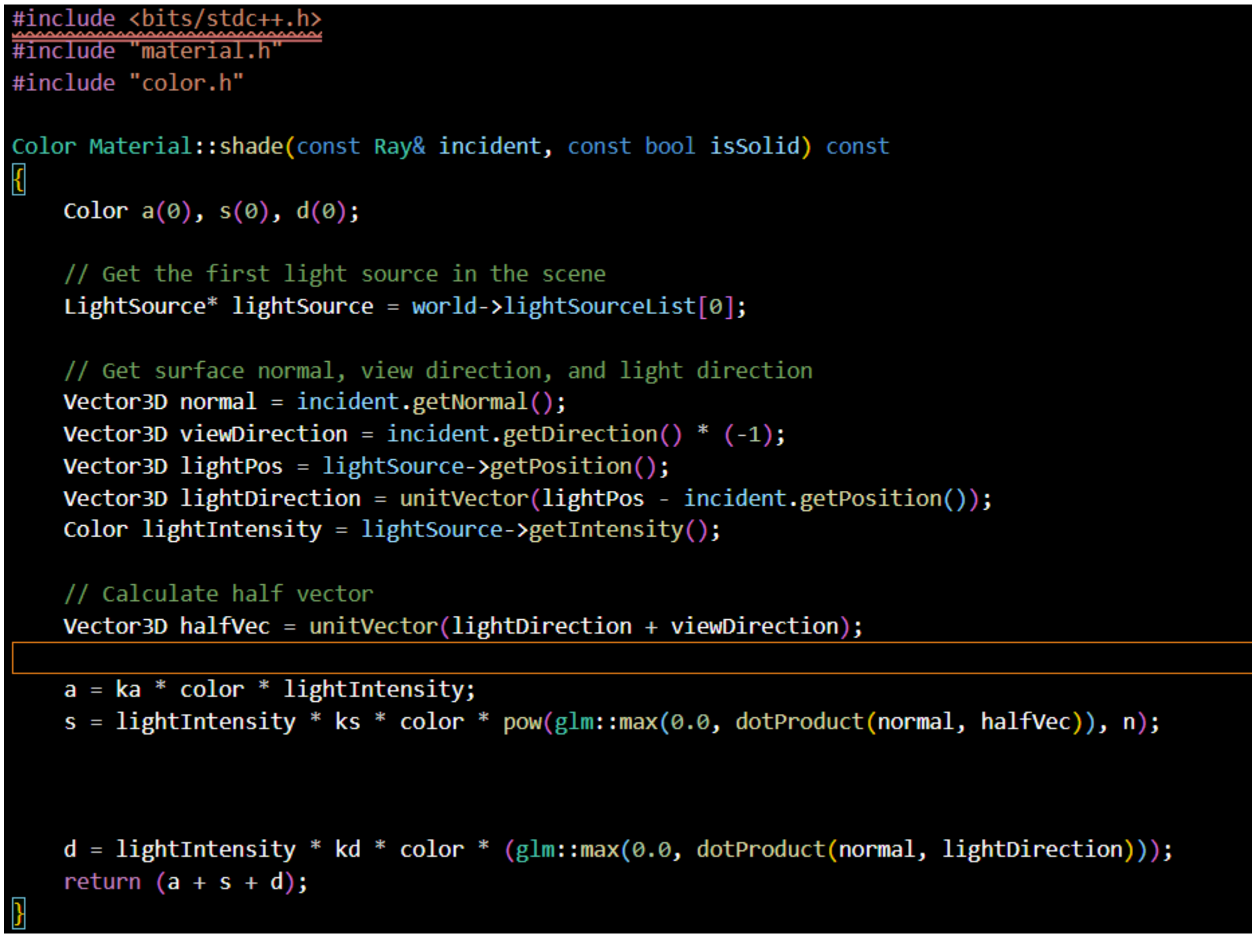
Note: some modification also made in CMakeLists.txt to introduce triangle.h and triangle.cpp

Output:



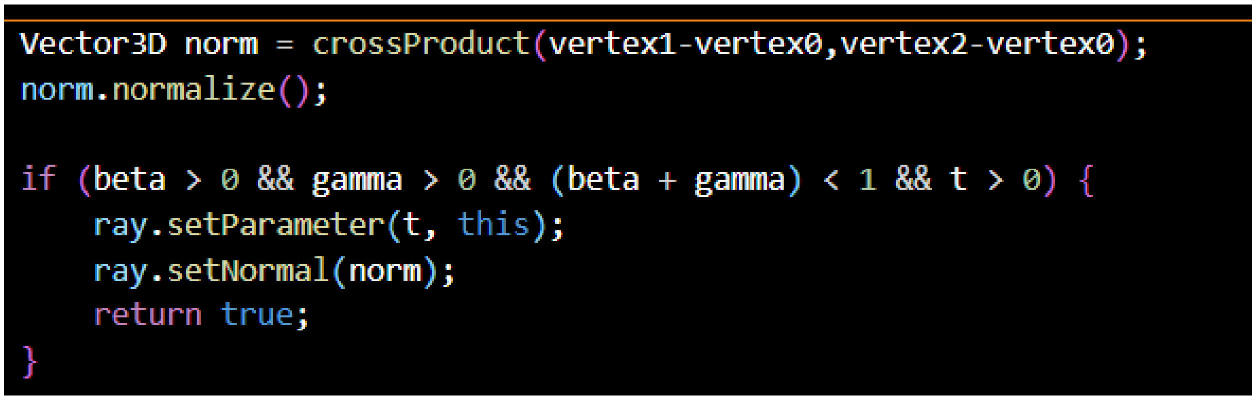
2. Implement Blinn-Phong shading for the shapes.

Update materials.cpp by implementing Blinn-Phong shading as studied in lab05:



```
#include <bits/stdc++.h>
#include "material.h"
#include "color.h"

Color Material::shade(const Ray& incident, const bool isSolid) const
{
    Color a(0), s(0), d(0);

    // Get the first light source in the scene
    LightSource* lightSource = world->lightSourceList[0];

    // Get surface normal, view direction, and light direction
    Vector3D normal = incident.getNormal();
    Vector3D viewDirection = incident.getDirection() * (-1);
    Vector3D lightPos = lightSource->getPosition();
    Vector3D lightDirection = unitVector(lightPos - incident.getPosition());
    Color lightIntensity = lightSource->getIntensity();

    // Calculate half vector
    Vector3D halfVec = unitVector(lightDirection + viewDirection);

    a = ka * color * lightIntensity;
    s = lightIntensity * ks * color * pow(glm::max(0.0, dotProduct(normal, halfVec)), n);



    d = lightIntensity * kd * color * (glm::max(0.0, dotProduct(normal, lightDirection)));
    return (a + s + d);
}
```

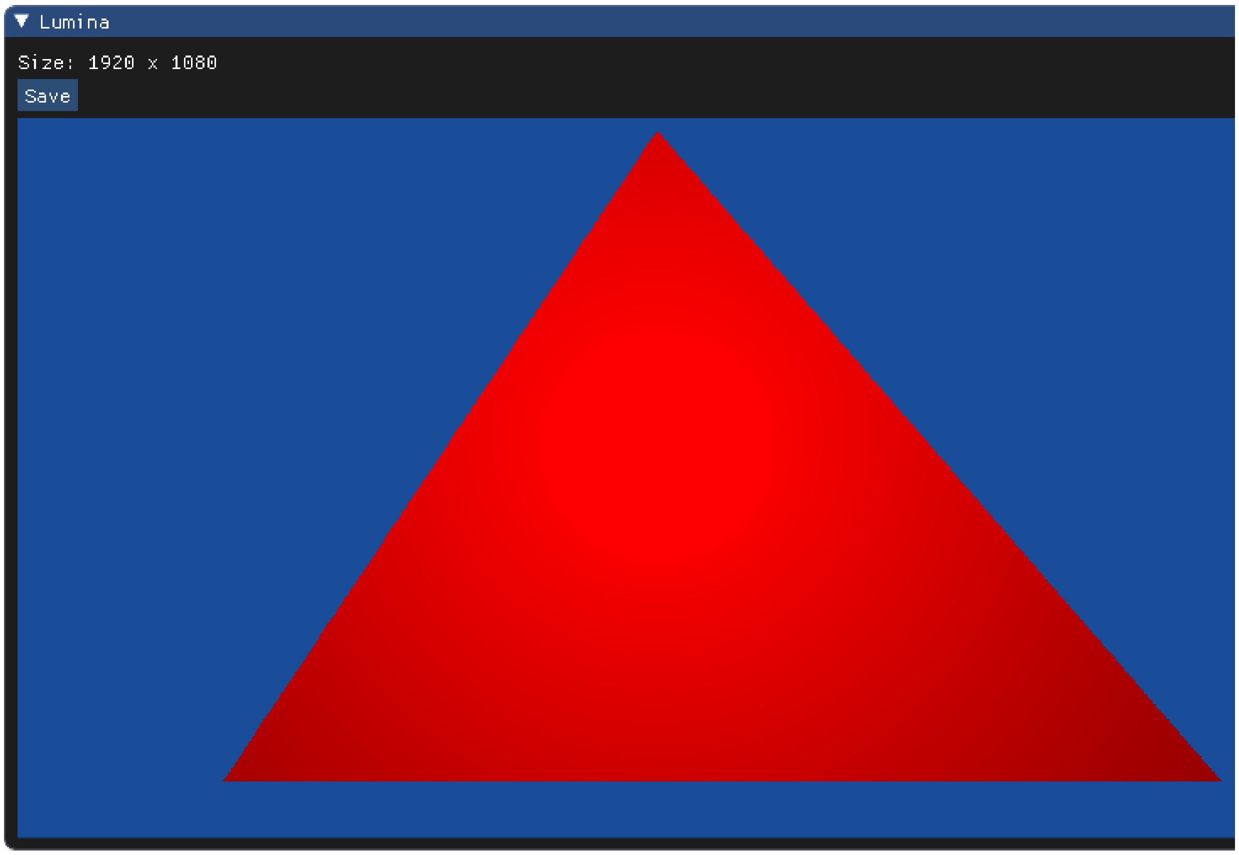
Implement normal calculations in triangle.cpp:



```
Vector3D norm = crossProduct(vertex1-vertex0,vertex2-vertex0);
norm.normalize();

if (beta > 0 && gamma > 0 && (beta + gamma) < 1 && t > 0) {
    ray.setParameter(t, this);
    ray.setNormal(norm);
    return true;
}
```

Note: define setNormal function in ray class

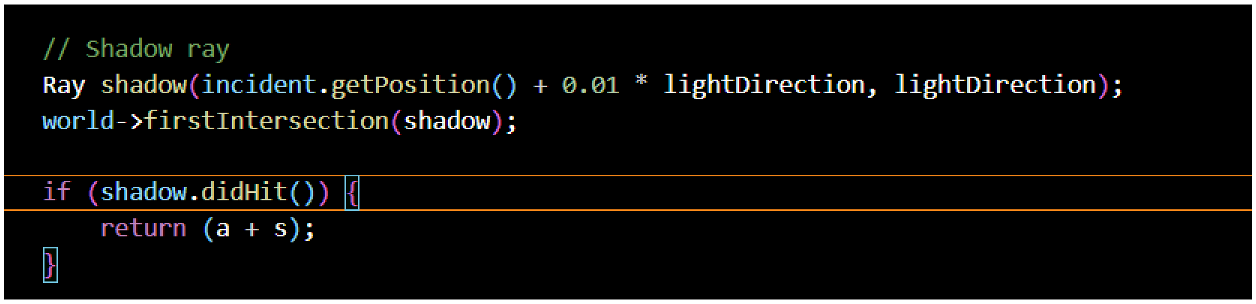
Output:



```
Size: 1920 x 1080
```

(observe the specular glow in middle and dark corners)

3. Implement shadows in the raytracer.
Introduce this peace of code in materials.cpp:

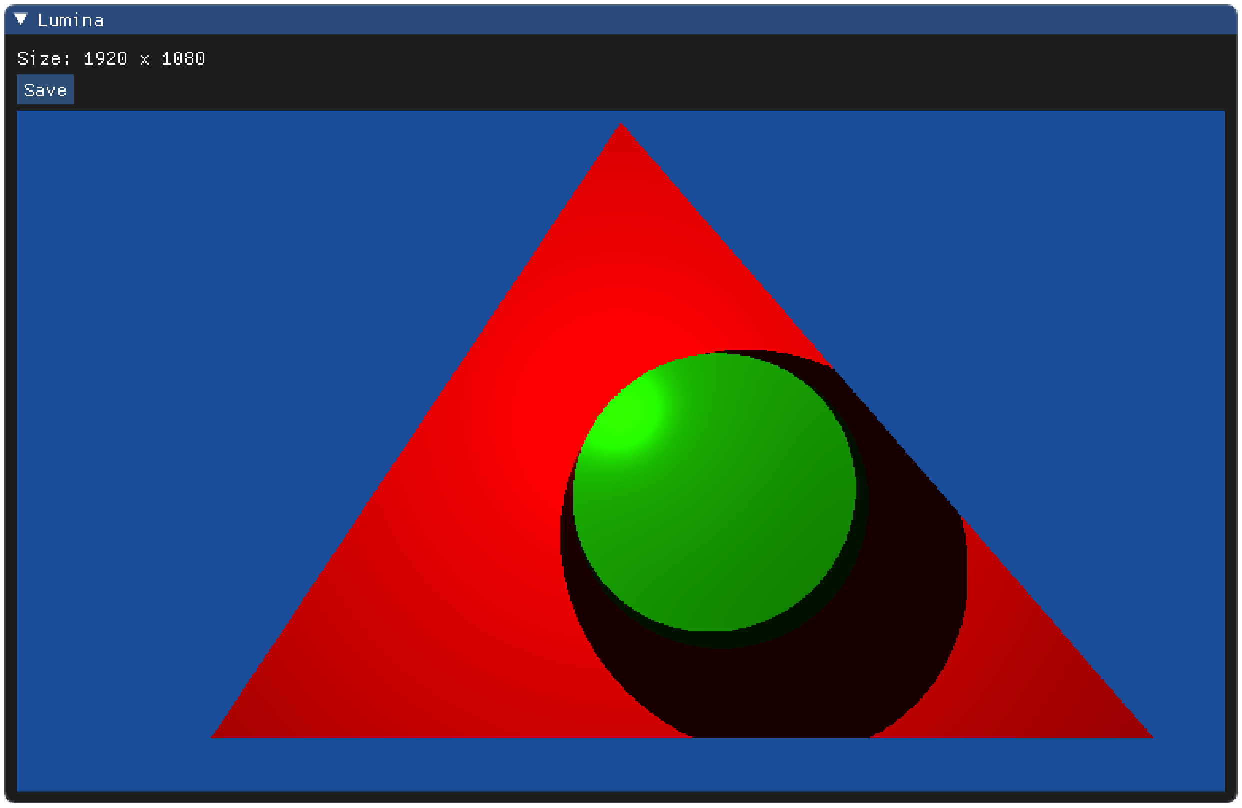


```cpp
// Shadow ray
Ray shadow(incident.getPosition() + 0.01 * lightDirection, lightDirection);
world->firstIntersection(shadow);

if (shadow.didHit()) {
    return (a + s);
}
```

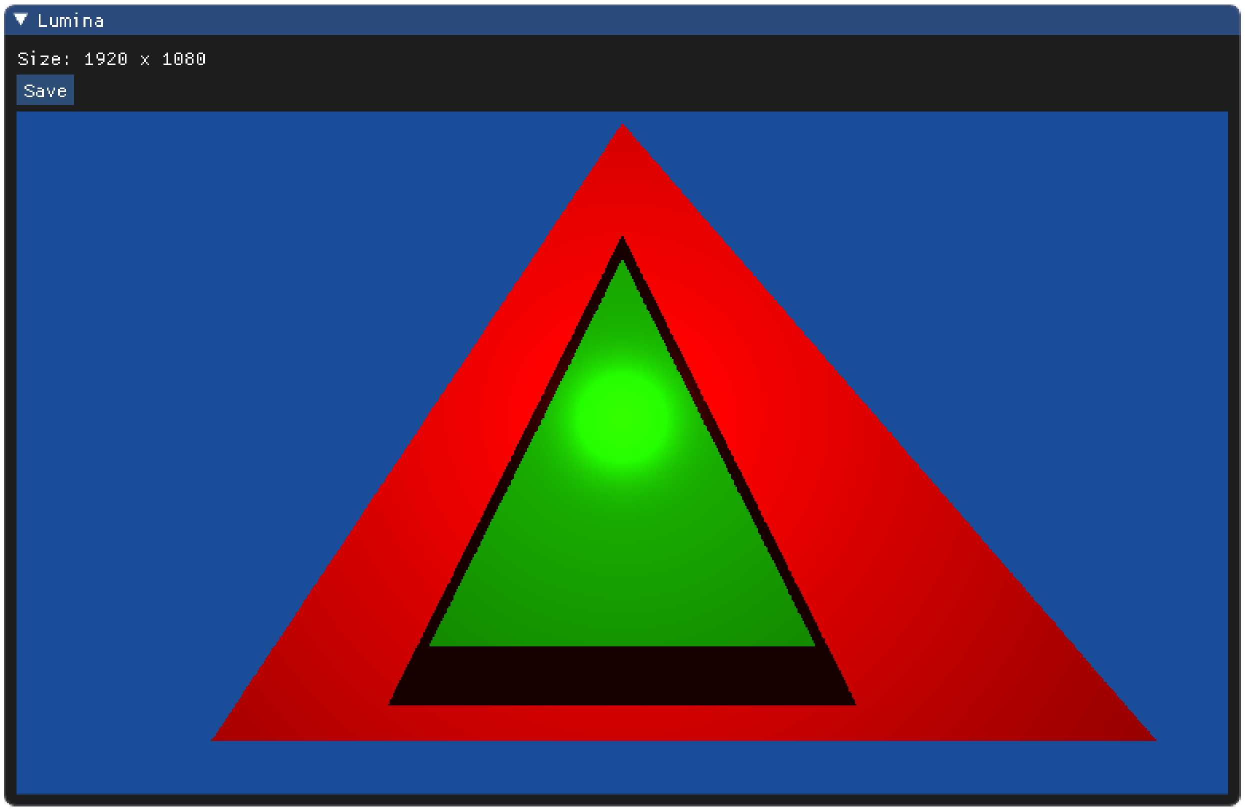
*Ray shadow(incident.getPosition() + 0.01 * lightDirection, lightDirection);* This line creates a shadow ray.
incident.getPosition() is the point on the surface from which the shadow ray is cast.
lightDirection is the direction from this point to the light source, indicating the direction in which the light is coming.

The 0.01 * lightDirection term adds a small offset to the starting point of the shadow ray. This offset is often used to ensure that the shadow ray starts slightly above the surface it originates from, which helps avoid self-intersections with the surface.
world->firstIntersection(shadow);: This line tests the shadow ray for intersections with objects in the scene by calling the firstIntersection method.
The purpose is to check if the shadow ray intersects with any object between the point on the surface and the light source.
If the shadow ray intersects with an object (i.e., *shadow.didHit()* is true), it means that there is an obstruction between the point on the surface and the light source. Therefore, the point is in shadow.
In this case, only the ambient and specular reflection components are returned. The diffuse component is omitted because there is no direct light to create diffuse reflection.

Output:

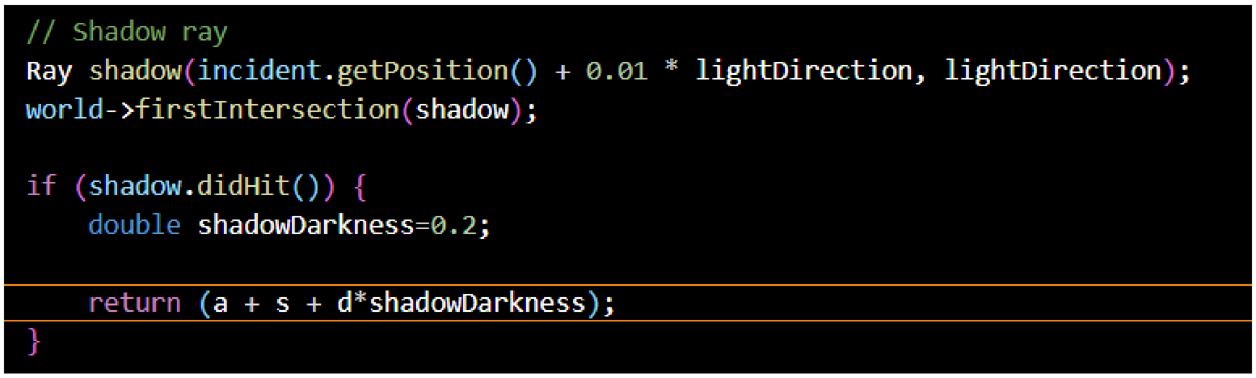


(sphere casting shadow over the triangle)

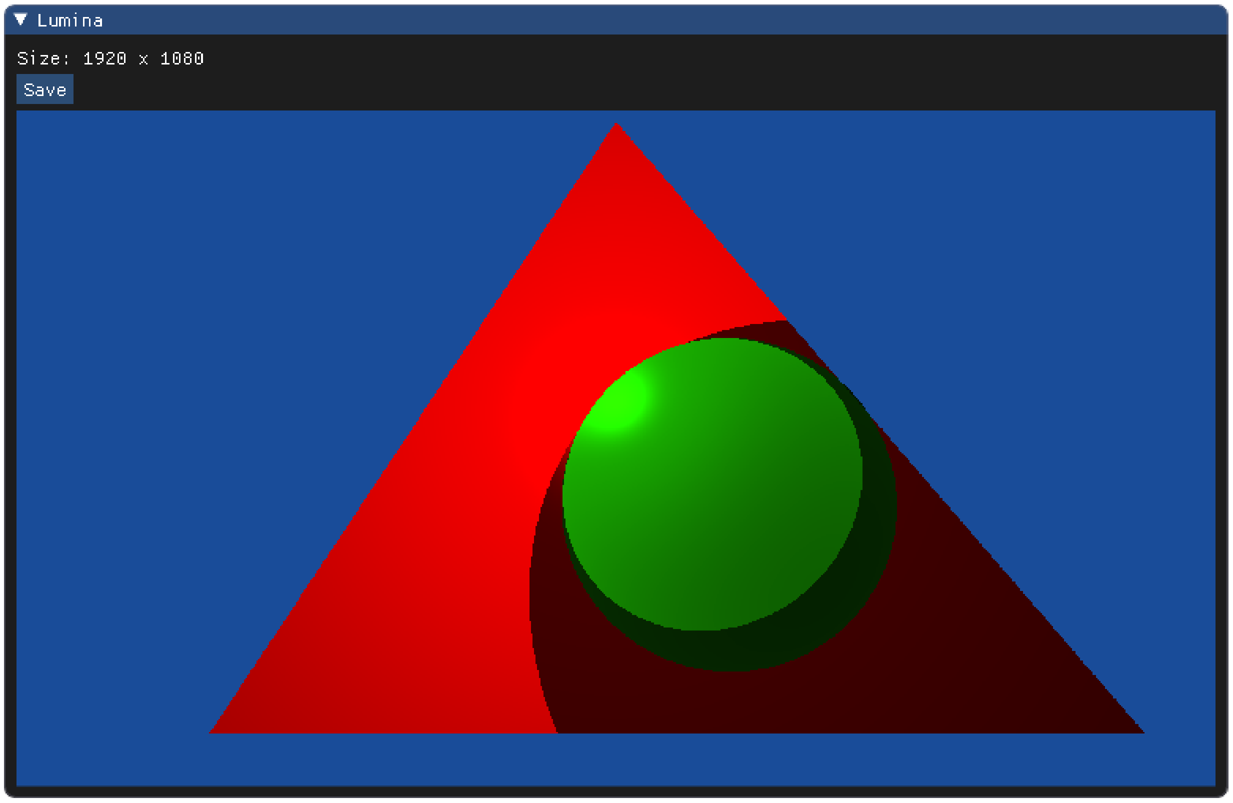


(green triangle casting shadow over the red triangle)

Note: To avoid pitch black shadow's we can update code as



```
// Shadow ray
Ray shadow(incident.getPosition() + 0.01 * lightDirection, lightDirection);
world->firstIntersection(shadow);

if (shadow.didHit()) {
    double shadowDarkness=0.2;

    return (a + s + d*shadowDarkness);
}
```



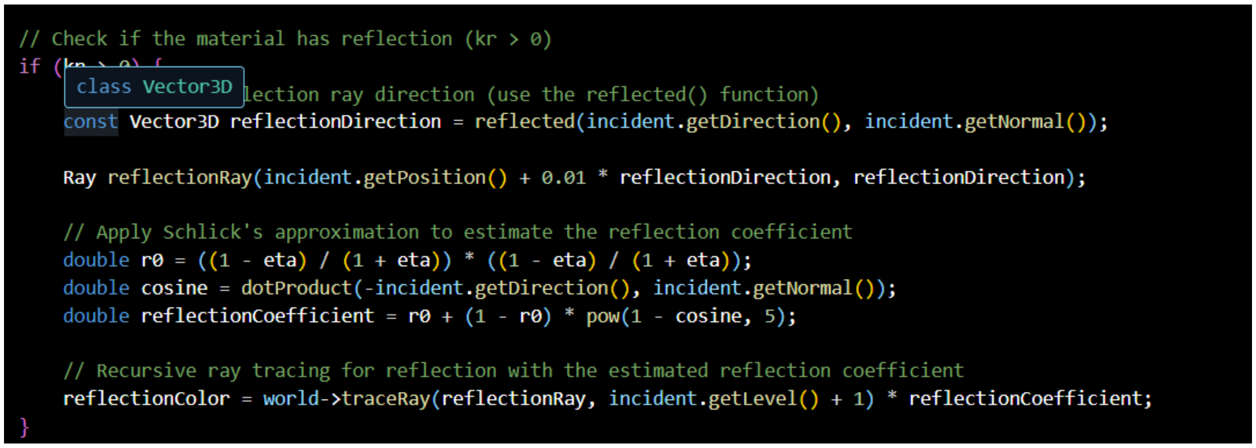
4. Implement reflective and dielectric materials (i.e., make your ray-tracer recursive).

H

Modification in Materials.cpp:

Reflection Handling:

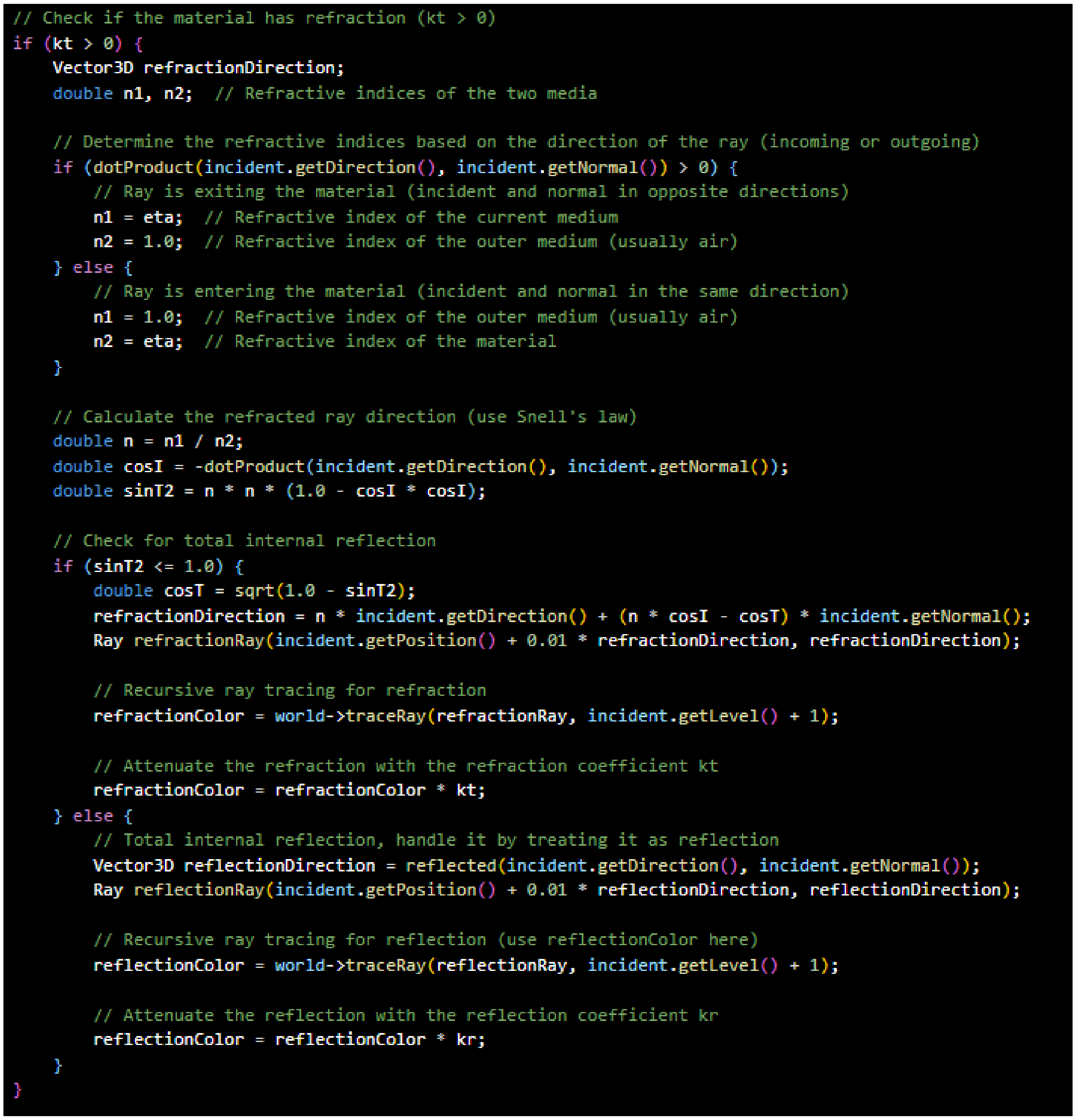
The code checks if the material has a reflection component (kr > 0), where kr is the reflection coefficient.
It calculates the reflection direction using the reflected function, which calculates the reflection of a ray.
Schlick's approximation is used to estimate the reflection coefficient. Schlick's approximation is a simplified model for approximating Fresnel reflection at a dielectric interface. It estimates the reflection coefficient based on the angle between the incident ray and the surface normal. The code calculates reflectionCoefficient using Schlick's formula.
Recursive ray tracing is used to calculate the reflection color. A new ray is traced in the reflection direction (reflectionRay), and the resulting color is attenuated by the reflection coefficient.



```
// Check if the material has reflection (kr > 0)
if (kr > 0) {
                        lection ray direction (use the reflected() function)
    const Vector3D reflectionDirection = reflected(incident.getDirection(), incident.getNormal());

    Ray reflectionRay(incident.getPosition() + 0.01 * reflectionDirection, reflectionDirection);

    // Apply Schlick's approximation to estimate the reflection coefficient
    double r0 = ((1 - eta) / (1 + eta)) * ((1 - eta) / (1 + eta));
    double cosine = dotProduct(-incident.getDirection(), incident.getNormal());
    double reflectionCoefficient = r0 + (1 - r0) * pow(1 - cosine, 5);

    // Recursive ray tracing for reflection with the estimated reflection coefficient
    reflectionColor = world->traceRay(reflectionRay, incident.getLevel() + 1) * reflectionCoefficient;
}
```

Refraction Handling:

The code checks if the material has a refraction component (kt > 0), where kt is the refraction coefficient.
It determines the refractive indices (n1 and n2) based on whether the ray is entering or exiting the material. These indices represent the refractive indices of the two media.
Using Snell's law, the code calculates the direction of the refracted ray. Snell's law relates the angles of incidence and refraction to the refractive indices of the two media.
The code checks for total internal reflection by comparing the sine of the angle of refraction (sinT2) with 1. If sinT2 is greater than 1, it means total internal reflection occurs, and the code handles it by treating it as reflection (similar to the reflection handling described above).



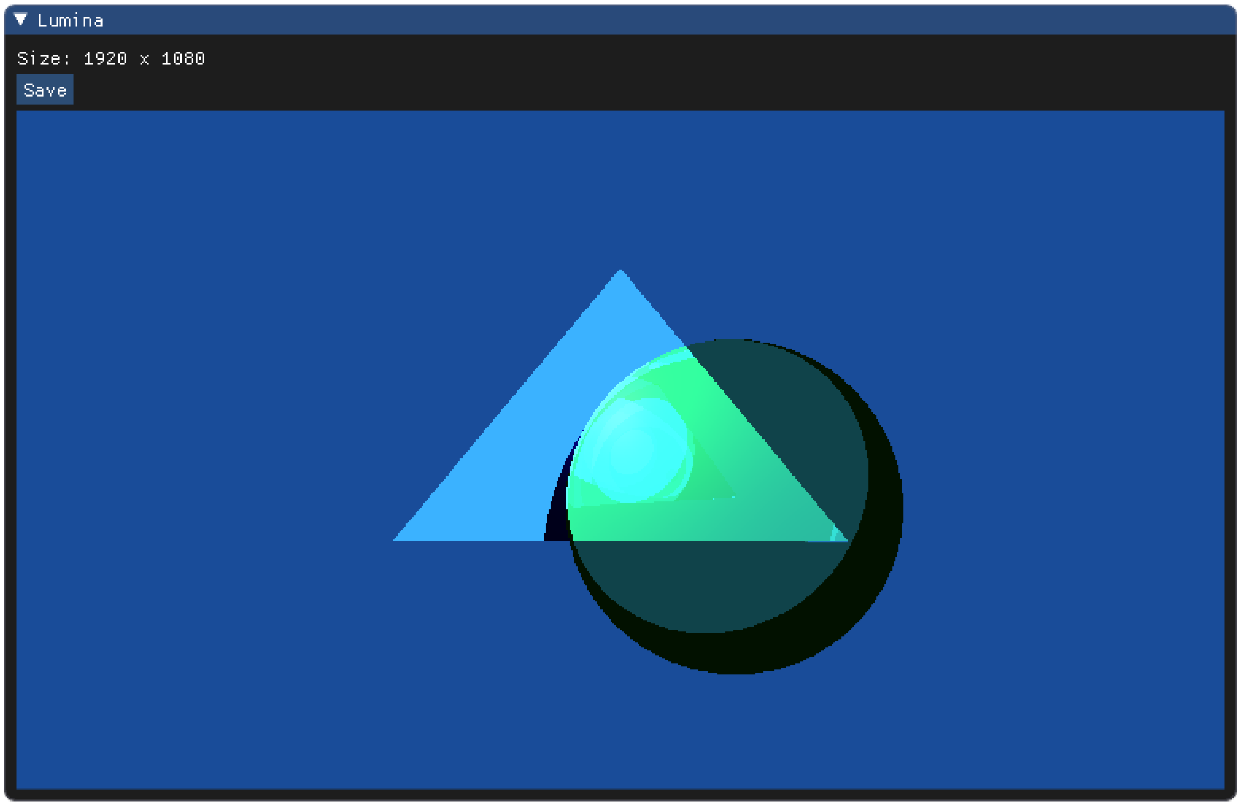
```
// Check if the material has refraction (kt > 0)
if (kt > 0) {
    Vector3D refractionDirection;
    double n1, n2;  // Refractive indices of the two media

    // Determine the refractive indices based on the direction of the ray (incoming or outgoing)
    if (dotProduct(incident.getDirection(), incident.getNormal()) > 0) {
        // Ray is exiting the material (incident and normal in opposite directions)
        n1 = eta;  // Refractive index of the current medium
        n2 = 1.0;  // Refractive index of the outer medium (usually air)
    } else {
        // Ray is entering the material (incident and normal in the same direction)
        n1 = 1.0;  // Refractive index of the outer medium (usually air)
        n2 = eta;  // Refractive index of the material
    }

    // Calculate the refracted ray direction (use Snell's law)
    double n = n1 / n2;
    double cosI = -dotProduct(incident.getDirection(), incident.getNormal());
    double sinT2 = n * n * (1.0 - cosI * cosI);

    // Check for total internal reflection
    if (sinT2 <= 1.0) {
        double cosT = sqrt(1.0 - sinT2);
        refractionDirection = n * incident.getDirection() + (n * cosI - cosT) * incident.getNormal();
        Ray refractionRay(incident.getPosition() + 0.01 * refractionDirection, refractionDirection);

        // Recursive ray tracing for refraction
        refractionColor = world->traceRay(refractionRay, incident.getLevel() + 1);

        // Attenuate the refraction with the refraction coefficient kt
        refractionColor = refractionColor * kt;
    } else {
        // Total internal reflection, handle it by treating it as reflection
        Vector3D reflectionDirection = reflected(incident.getDirection(), incident.getNormal());
        Ray reflectionRay(incident.getPosition() + 0.01 * reflectionDirection, reflectionDirection);

        // Recursive ray tracing for reflection (use reflectionColor here)
        reflectionColor = world->traceRay(reflectionRay, incident.getLevel() + 1);

        // Attenuate the reflection with the reflection coefficient kr
        reflectionColor = reflectionColor * kr;
    }
}
```

Beer's Law:
The code then calculates the distance the shadow ray travels within the material, and it applies Beer's Law to simulate absorption. Beer's Law describes how light is attenuated as it travels through a medium. The absorption coefficient (absorption_coefficient) determines how much light is absorbed over the distance traveled.

distance: The distance the shadow ray travels within the material.
absorption: The color of the material attenuated based on Beer's Law.

```
//BEER'S APPROXIMATION
// Calculate the distance traveled within the material
double distance = shadow.getParameter();

// Implement Beer's Law to simulate absorption
Color absorption = Color(exp(-absorption_coefficient * distance));

// Combine ambient, diffuse, and specular with absorption
Color combinedColor = a + s + d;

combinedColor = combinedColor * absorption;
```

Output:


Lumina
Size: 1920 x 1080
Save