Marina Hernandez Salas – 241273
Iker Landarech Mayo – 241617

Universitat Pompeu Fabra Barcelona

ACG - Lab 2 - Extend your Ray Tracer

In this lab we aim to simulate Phong, Mirror and Transmissive Illuminations (with shadows) using ray tracing. And learn how to simulate indirect illumination using the rendering equation.

Task 1: Enhancing the Direct Shader with Mirror Materials

In this first task, the goal was to implement the *Mirror* material in the *DirectShader*. This involved creating a new class called *Mirror* that represents perfect specular materials like a mirror. The *DirectShader* was modified to compute the color of each pixel based on specular light instead of diffuse or glossy. The color was determined by computing the reflection direction of the incident light using the mirror formula. The result was an image that simulated a mirroring material on one of the objects in the scene. Notice that a perfect specular material has index of refraction **0** and $hasSpecular() \rightarrow True$.

```cpp
#include "mirror.h"
#include "../core/utils.h"

Mirror::Mirror()
{
}

Vector3D Mirror::getReflectance(const Vector3D& n, const Vector3D& wo, const Vector3D& wi) const {
    return Vector3D();
}

bool Mirror::hasSpecular() const
{
    return true;
}

bool Mirror::hasTransmission() const
{
    return false;
}

bool Mirror::hasDiffuseOrGlossy() const
{
    return false;
}

double Mirror::getIndexOfRefraction() const
{
    return 0.0;
}

Vector3D Mirror::getDiffuseCoefficient() const
{
    return Vector3D();
}
```

```cpp
else if (intersection.shape->getMaterial().hasSpecular())
{

    Vector3D wr = Utils::computeReflectionDirection(r.d, n);
    Ray reflectionRay = Ray(intersection.itsPoint, wr, r.depth + 1);
    color = computeColor(reflectionRay, objList, lsList);

}
```

```cpp
Vector3D Utils::computeReflectionDirection(const Vector3D &rayDirection, const Vector3D &normal)
{
    // Compute the perfect reflection direction
    return normal * 2 * dot(normal, -rayDirection) + rayDirection;
}
```

Task 2: Enhancing the Direct Shader with Transmissive Materials

The goal of this second task, was to implement the ***Transmissive*** material in the ***DirectShader***. A new class called ***Transmissive*** was created, which represents materials like glass that refract or reflect light according to Snell's law. The ***DirectShader*** was updated to check if the material passed has transmissive properties and compute the color of the pixel accordingly. The implementation involved determining whether the intersection point was inside or outside the sphere and using the appropriate equation for computing the refracted ray. By applying the ***Transmissive*** material to objects, the ray tracer was able to generate images that simulate the refraction and reflection of light. Notice that we will be using an index of refraction of $\mu_t = 1.1$ and $hasTransmission() \rightarrow True$. We have also taken into account the total internal reflection error control when computing w_t making sure the term inside the square root follows $sqint \geq 0$.

```cpp
#include "transmissive.h"
#include "../core/utils.h"

Transmissive::Transmissive()
{
}

Vector3D Transmissive::getReflectance(const Vector3D& n, const Vector3D& wo, const Vector3D& wi) const
{
    return Vector3D();
}

bool Transmissive::hasSpecular() const
{
    return false;
}

bool Transmissive::hasTransmission() const
{
    return true;
}

bool Transmissive::hasDiffuseOrGlossy() const
{
    return false;
}

double Transmissive::getIndexOfRefraction() const
{
    return 1.1;
}

Vector3D Transmissive::getDiffuseCoefficient() const
{
    return Vector3D();
}
```

Now, $cos\beta = \sqrt{1 - sin^2\beta}$; from Snell's law, we have $sin\beta = sin\alpha\eta_t$; and grouping the terms with **n**

$$\boldsymbol{t} = \left(\sqrt{1 - \eta_t{}^2 sin^2\alpha} - \eta_t(\boldsymbol{l}\cdot\boldsymbol{n})\right)\boldsymbol{n} + \eta_t\boldsymbol{l}$$

where $sin^2\alpha = 1 - cos^2\alpha = 1 - (\boldsymbol{l}\cdot\boldsymbol{n})^2$
Thus we have a formula for the refracted ray from **n**, **l** and η_t

Total Internal Reflection: If $1 - \eta_t{}^2 sin^2\alpha < 0$, the square root has no real solution. What happens then?
Direction is important: If rays go from 2 to 1, then the refraction index will be inverted!!

$$\frac{sin\alpha}{sin\beta} = \frac{\eta_2}{\eta_1} = {}^1/_{\eta_t}$$

```cpp
Vector3D n = intersection.normal.normalized();
Vector3D wo = -r.d;
Vector3D l = wo;
//TRANSMISSIVE//
if (intersection.shape->getMaterial().hasTransmission()) {
    double nt = intersection.shape->getMaterial().getIndexOfRefraction();
    if (dot(n, l) < 0)
    {
        n = -n;
        nt = 1 / nt;
    }
    double sin2alf = 1 - pow(dot(l, n), 2);
    double sqint = 1 - pow(nt, 2) * sin2alf;
    //Checking Internal Reflection:
    if ((sqint) >= 0)
    {
        double cosa = sqrt(1 - sin2alf);
        double sina = sqrt(sin2alf);
        double sin2bet = pow((sina * nt), 2);
        double cosb = sqrt(1 - sin2bet);

        Vector3D wt = Utils::computeTransmissionDirection(r, n, nt, cosa, cosb);
        Ray refractionRay = Ray(intersection.itsPoint, wt, r.depth);
        color = computeColor(refractionRay, objList, lsList);

    }
    else if ((sqint) < 0)
    {
        Vector3D wr = Utils::computeReflectionDirection(r.d, n);
        Ray reflectionRay = Ray(intersection.itsPoint, wr, r.depth);
        color = computeColor(reflectionRay, objList, lsList);

    }
}
```

Task 3: Introduction to Global Illumination:

In the third task, the objective was to introduce global illumination in the ray tracer, thanks to that we improved the simulation of light propagation by accounting for indirect illumination, which refers to the light that arrives at a surface point after bouncing off other objects in the scene. We started by implementing global illumination, so the first step was to create a new shader called *GlobalShader* based on the current *DirectShader*.

This *GlobalShader* uses an *ambient term* to approximate the indirect illumination instead of explicitly computing it. The ambient term represents the small amount of light that is scattered throughout the scene. It assumes that the same small amount of indirect light arrives at all points in the scene. However, it is important to note that this approximation has limitations, such as assuming constant indirect illumination across the entire scene and not simulating effects like color bleeding.

To implement the *GlobalShader*, we copied the structure of the *DirectShader* and introduced this new constant called *ambient term*. The *ambient term* is a color (*Vector3D*) that contributes to the indirect illumination. We also used the diffuse coefficient (*kd*) obtained from the *Phong* material to compute the color. Finally, we computed the color of *GlobalShader* by summing the product of the *ambient term* and *kd* with the direct illumination computed at the shading point.

Please notice that after doing some trial and error in order to match the reference picture we have used an *ambient term* of *Vector3D*(0.7) and that we have computer the final output color as:

```
color = dirIllumination + indirIllumination.operator*(3);
```

```
indirIllumination = intersection.shape->getMaterial().getDiffuseCoefficient() * at
```

Task 4: 2-bounces Indirect Illumination:

In this task the objective was to enhance the ray tracer by implementing two bounces of indirect illumination. As previously mentioned, the previous implementation of indirect illumination using an ambient term was not realistic and lacked effects like color bleeding. So now the illumination and ***ambient term*** were determined by the bounce of rays, adding color bleeding effects.

To achieve this, we defined the indirect incident illumination component, ***Lind***, for the hemisphere centered on the surface normal. We computed this component by sending ***nSamples*** rays through the hemisphere and obtaining a random direction, w_j, using the ***getSample(normal)*** function. And finally, we computed the color using the intersection's normal and the obtained random sample.

$$L_o^{ind}(\mathbf{p}, \boldsymbol{\omega}_o) = \frac{1}{2\pi n} \sum_{j=1}^{n} L_i(\mathbf{p}, \boldsymbol{\omega}_j)\, r(\boldsymbol{\omega}_j, \boldsymbol{\omega}_o)$$

The implementation involved checking if the ray depth was 0, indicating the first intersection with a ***Phong*** material. In this case, ***nSamples*** rays were sent through the hemisphere, and L_{ind} was calculated by adding each ray's contribution. The color was computed using the normal of the intersection.

If the ray depth was greater than 0, the light component was computed as the product of the ambient term and the diffuse component ($at \cdot kd$). This accounted for the indirect illumination in subsequent bounces.

The result of implementing two bounces of indirect illumination can be seen in the figure provided, where the scene appears more realistic and illuminated.

```
//PHONG//
else if (intersection.shape->getMaterial().hasDiffuseOrGlossy())
{
    for (size_t s = 0; s < lsList.size(); s++)
    {
        Vector3D wo = (r.o - intersection.itsPoint).normalized();
        Vector3D wi = (lsList.at(s).getPosition() - intersection.itsPoint).normalized();
        Vector3D lightIntensity = lsList.at(s).getIntensity(intersection.itsPoint);
        Vector3D reflectance = intersection.shape->getMaterial().getReflectance(n, wo, wi);
        Ray wiRay = Ray(intersection.itsPoint, wi, 0, Epsilon, (lsList.at(s).getPosition() - intersection.itsPoint).length());
        if (!Utils::hasIntersection(wiRay, objList))
        {
            visibility = 1;
            dirIllumination = dirIllumination + (lightIntensity * reflectance * visibility);
        }
    }
    if (r.depth == 0)
    {
        Vector3D coeff = (1.0 / (N_DIRECTIONS * 2.0 * M_PI));
        for (size_t i = 0; i < N_DIRECTIONS; i++)
        {
            HemisphericalSampler hs = HemisphericalSampler();
            Vector3D wj = hs.getSample(n).normalized();
            Ray secondaryRay = Ray(intersection.itsPoint, wj, r.depth + 1);
            Vector3D li = computeColor(secondaryRay, objList, lsList);
            Vector3D rp = intersection.shape->getMaterial().getReflectance(n, wo, wj);
            indirIllumination += li * rp;
        }
        indirIllumination = indirIllumination.operator*(coeff);
    }
    else if (r.depth > 0)
    {
        indirIllumination = intersection.shape->getMaterial().getDiffuseCoefficient() * at;
    }
```

Task 5: n-bounces Indirect Illumination:

In this final Task 5, the objective was to extend the ray tracer to support n-bounces of light across the scene. This implementation involved three different options based on the ray depth.

First, we checked whether the ray depth was 0. If this condition was true, we followed the same procedure as in the previous task, sending $nSamples$ rays through the hemisphere centered on the surface normal to compute the indirect light component, L_{ind}.

Next, we checked whether the ray depth was equal to the maximum depth defined. If this condition was true, we used the approximation of the ambient term by the diffuse coefficient. This involved using the ambient term, at, and the diffuse coefficient, kd, to compute the indirect illumination.

Finally, if none of the other options were true, we computed the light component explicitly using an equation. The indirect light component, L_{ind}, was computed by:

After all of that we obtained the following result:

```
    else if (r.depth > 0)
    {
        indirIllumination = intersection.shape->getMaterial().getDiffuseCoefficient() * at;
    }
    else if (r.depth == N_BOUNCES)
    {
        indirIllumination = intersection.shape->getMaterial().getDiffuseCoefficient() * at;
    }
    else
    {
        Vector3D coeff2 = (1.0 / (4.0 * M_PI));
        Vector3D wr = (n * 2 * dot(wo, n) - wo);
        Vector3D wn = n;

        Ray rayWr = Ray(intersection.itsPoint, wr, r.depth + 1);
        Ray rayWn = Ray(intersection.itsPoint, wn, r.depth + 1);
        Vector3D lii = computeColor(rayWr, objList, lsList);
        Vector3D rpi = intersection.shape->getMaterial().getReflectance(n, wo, wr);
        Vector3D lin = computeColor(rayWn, objList, lsList);
        Vector3D rpn = intersection.shape->getMaterial().getReflectance(n, wo, wn);
        indirIllumination = ((lii * rpi) + (lin + rpn)).operator*(coeff2);
    }
    color = dirIllumination + indirIllumination.operator*(3);
}
return color;
```

FINAL TEST:

N_DIRECTIONS = 800

N_BOUNCES = 2