

ACG - Lab 1 - Basic Ray Tracing

In this lab we aim to simulate Phong Illumination (with shadows) using ray tracing. Learning how to use different ray programs and deriving multiple integrators from the Shader parent class.

Task 1: Painting the Image (1 points)

The aim of this initial task was to generate an image constructed by computing the color of each individual pixel in a rendering loop, using the provided formula. This process ultimately produces a smooth transition of colors, creating a gradient effect. We achieved this by filling the formula into a function called *PaintImage()*. This function needs a *film* as an argument, and the rendering loop iterates through all the pixels in the image. To initialize the loop, we extracted the width and height from the provided film, which served as the basis for two separate loops. Within these loops, we performed color computations using the *Vector3D* class, assigning the computed color to the respective pixel.

```
//-----TASK 1-----//
void PaintImage(Film* film)
{
    unsigned int sizeBar = 40;

    size_t resX = film->getWidth();
    size_t resY = film->getHeight();

    // Main Image Loop
    for (size_t lin = 0; lin < resY; lin++)
    {
        // Show progression
        if (lin % (resY / sizeBar) == 0)
            std::cout << ".";

        for (size_t col = 0; col < resX; col++)
        {
            Vector3D pixel_color = Vector3D((double)col / resX, (double)lin / resY, 0);
            film->setPixelValue(col, lin, pixel_color);
        }
    }
}
```



Task 2: Implement the Intersection Integrator (2 points)

The goal of this second task was to produce an image where pixels are either colored in red, or in black.

As in this task now we are working with objects, we started by developing the *hasIntersection()* function, which iterates through a list of objects (in this case, three spheres). Then calls *rayIntersectP()* for each sphere and returns true if there's an intersection with any of them. This latter function is part of the *Sphere* class and simply checks if the ray intersects the current object.

As a result, when an intersection is detected, the red circles corresponding to the spheres are shown in *red* and *black* background where no intersection is detected. Then, we used the provided shader, *Intersection Shader*, and directly call the ray tracing function from

the main program inside the `computeColor()` function to return `hitcolor(1,0,0)` when `hasIntersection()` changes state to true.

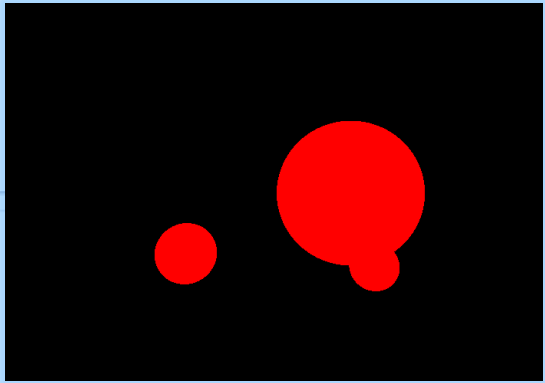
```
bool Utils::hasIntersection(const Ray& cameraRay, const std::vector<Shape*>& objectsList) //or Shadow Ray
{
    bool intersection = false;
    for (size_t objIndex = 0; objIndex < objectsList.size(); objIndex++)
    {
        const Shape* obj = objectsList.at(objIndex);
        if (obj->rayIntersectP(cameraRay))
        {
            intersection = true;
        }
    }
    return intersection;
}

#include "intersectionshader.h"
#include "../core/utills.h"

IntersectionShader::IntersectionShader() :
    hitColor(Vector3D(1, 0, 0))
{ }

IntersectionShader::IntersectionShader(Vector3D hitColor_, Vector3D bgColor_) :
    Shader(bgColor_), hitColor(hitColor_)
{ }

Vector3D IntersectionShader::computeColor(const Ray &r, const std::vector<Shape*> &objList, const std::vector<PointLightSource> &lsList) const
{
    if (Utils::hasIntersection(r, objList))
        return hitColor;
    else
        return bgColor;
}
```



Task 3: Implement the Depth Integrator (2 points)

In this third task the aim was to depth the objects by its position on the scene. To achieve this, we developed the `getClosestIntersection()` function by calling the function `rayIntersect()` iterating for each the objects of the scene. The main change between this function and the previously used `rayIntersectP()` is that this particular function does not only identifies intersections but also updates them. Meaning it establishes an auxiliary intersection and compares it with the remaining objects in the list to determine the closest intersection to the ray origin.

In this task we also derived a new shader `depthshader` class from the parent `Shader` class. And we have used this new integrator to compute the `color` by considering the ray, the shapes in the scene, and the position of the light source.

This has enabled us to factor in shading when determining the `color`, following the formula presented in class. In the main program, we employ this function by creating a new instance of a `depthshader`. Consequently, this new shader is passed into the `raytrace()` function.

```

bool Utils::getClosestIntersection(const Ray& cameraRay, const std::vector<Shape*>& objectsList, Intersection& its)
{
    bool intersection = false;
    for (size_t objIndex = 0; objIndex < objectsList.size(); objIndex++)
    {
        const Shape* obj = objectsList.at(objIndex);
        if (obj->rayIntersect(cameraRay, its))
        {
            intersection = true;
        }
    }
    return intersection;
}

#include "depthshader.h"
#include "../core/Utils.h"

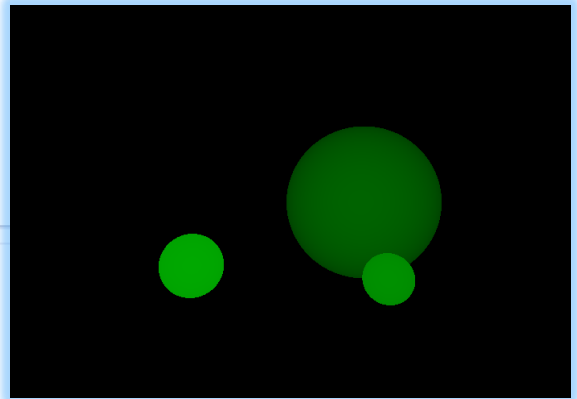
DepthShader::DepthShader() :
    color(Vector3D(1, 0, 0))
{ }

DepthShader::DepthShader(Vector3D hitColor_, double maxDist_, Vector3D bgColor_) :
    Shader(bgColor_), color(hitColor_), maxDist(maxDist_)
{ }

Vector3D DepthShader::computeColor(const Ray& r, const std::vector<Shape*>& objList, const std::vector<PointLightSource>& lsList) const
{
    double ci = 0.0;
    double hitDist = 0.0;
    Intersection* intersection = new Intersection();

    if (Utils::getClosestIntersection(r, objList, *intersection)) {
        hitDist = (r.o - intersection->itsPoint).length();
        ci = 1.0 - (hitDist / maxDist);
        if (ci < 0.0) {
            ci = 0.0;
        }
        return Vector3D(0, ci, 0);
    }
    return bgColor;
}

```



Task 4: Implement the Normal Integrator (1 points)

In this task, we were asked to display the *normals* of the objects placed in the scene. To achieve this, we followed a procedure similar to the previous task by changing the compute color function inside a new introduced a class named *normalshader* where the color computation was computed using the formula provided in class. This formula computed the *normal* vector and added 1 to each color channel in the result. Different from the previous shader, this new formula takes into account not only the depth but also the shape of the sphere. We also updated the main file, creating a new instance of *normalshader* instead of *depthshader* and launching some rays with *raytrace()*.

```

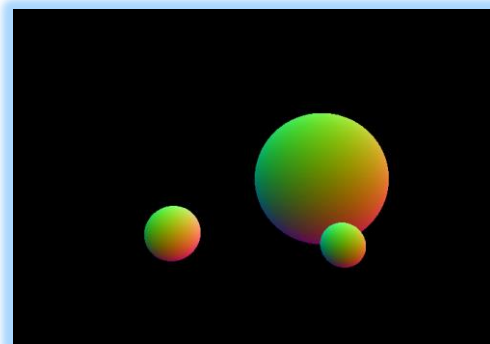
#include "normalshader.h"
#include "../core/Utils.h"

NormalShader::NormalShader() :
    color(Vector3D(1, 0, 0))
{ }

NormalShader::NormalShader(Vector3D hitColor_, double maxDist_, Vector3D bgColor_) :
    Shader(bgColor_), color(hitColor_), maxDist(maxDist_)
{ }

Vector3D NormalShader::computeColor(const Ray& r, const std::vector<Shape*>& objList, const std::vector<PointLightSource>& lsList) const
{
    Intersection* intersection = new Intersection();
    Vector3D color;
    Vector3D normal;
    Vector3D normone = Vector3D(1.0, 1.0, 1.0);
    if (Utils::getClosestIntersection(r, objList, *intersection)) {
        normal.x = intersection->normal.x;
        normal.y = intersection->normal.y;
        normal.z = intersection->normal.z;
        color = (normal + normone) / 2;
        return color;
    }
    return bgColor;
}

```



Task 5: Implement Phong Integrator (4 points)

In this last assignment, we were asked to perform several key objectives in order to enhance the realism of our rendered scenes.

First, we need to derive a new class called *Phong* from the existing *Material* parent class. The purpose of this new material was to compute the reflectance and refraction for a given object and ray. Since we didn't involve refraction in our scenario, we simply returned -1 for this aspect in the material main class function `getIndexOfRefraction()`. On the other hand, for computing the object's reflectance, we used the formulas provided in the guide of the lab assignment and the modified expressions of the theory lectures slides. To achieve the reflectance, we needed to determine the direction of the reflection and later apply it to compute the *Phong* material.

One of the biggest challenges we have had to face in this task has to do with computing the reflectance of the *Phong* material, given by its coefficients $r(\omega_i, \omega_o)$, since when computing the *ideal reflection direction* given by the expression $\omega_r = 2(n \cdot \omega_i)n - \omega_i$ we have involved the `dot()` function between the normal n and the incoming direction toward the light source ω_i , we have also involved the `operator *` function from the *Vector3D* class to perform the $(n \cdot \omega_i)n$ operation. Finally, following the lecture notes we have involved a *max* variable between the $dot(n \cdot \omega_i)$ and 0. See expression below from slide 45 of *RT*.

At a point P, for each light

$$I = k_a \otimes I_a + k_d \otimes \left(\frac{I}{r^2} \right) \max(0, n \cdot l) + k_s \otimes \left(\frac{I}{r^2} \right) \max(0, n \cdot h)^p$$

We had to also normalized the ideal reflection direction ω_r .

Once we had the reflectance of the *Phong* material computed we developed the *directShader* integrator from the parent *Shader* class.

Similarly, to previous shader integrators we have updated the `computeColor()` function to compute the outgoing light leaving a given shading point p in the ω_o direction, opposite to the camera ray. For this last task, we have introduced 3 spheres with Phong material illuminated by 3 point-light sources. Meaning in this function inside direct shader we have to take into account that the output light $L_o(\omega_o)$ will need to be computed for each of the 3 light sources and reflectance will change. Thus, we have implemented all the elements of the expression

$$L_o(\omega_o) = \sum_{s=1}^{nL} L_i^s(p) r(\omega_i^s, \omega_o) V_i^s(p)$$

inside a for loop which updates the elements values depending on the light sources that illuminate each of the object of the scene. Following the exercises of the theory lectures

it has been pretty easy to derive all the expression for most of the variables of the equation, we have also taken into account a normalization for the w_i and w_o directions vectors. The main challenge of this part was to compute the shadows of the objects of the scene, meaning change the visibility variable state accordingly depending on the intersections. Again, thank to some theory help we derived 2 solutions or approaches, the first one consisted on computing the angle of the reflection between the incoming light direction and the normal of the object and evaluating if this angle was bigger than 0 to determine if that specific intersection involved a reflection or not to change the visibility state accordingly but after some trial and error we derived a second approach which consisted on creating (launching) a new *Ray* called *wiRay* which has its origin at the intersection point and w_i direction. This way if evaluating the intersection between this ray and each of the objects with the *hasIntersection(wiRay,objList)* function returned *false* state, it meant that there was no intersection and thus, there was proper visibility $V(p) = 1$.

```
#include "directshader.h"
#include "../core/utils.h"

DirectShader::DirectShader() :
    color(Vector3D(1, 0, 0))
{ }

DirectShader::DirectShader(Vector3D hitColor_, double maxDist_, Vector3D bgColor_) :
    Shader(bgColor_), color(hitColor_), maxDist(maxDist_)
{ }

Vector3D DirectShader::computeColor(const Ray& r, const std::vector<Shape*>& objList, const std::vector<PointLightSource*>& lsList) const
{
    Intersection intersection;
    Vector3D outputLight = Vector3D(0, 0, 0);
    if (Utils::getClosestIntersection(r, objList, intersection)) {
        bool visibility = false;

        for (size_t s = 0; s < lsList.size(); s++)
        {
            Vector3D normal = intersection.normal;
            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(normal, wo, wi);
            Ray wiRay = Ray(intersection.itsPoint, wi, 0, Epsilon, (lsList.at(s).getPosition() - intersection.itsPoint).length());
            if (!Utils::hasIntersection(wiRay, objList))
            {
                visibility = 1;
                outputLight = outputLight + (lightIntensity * reflectance * visibility);
            }
        }
        return outputLight;
    }
    return bgColor_;
}

Phong::Phong(Vector3D kd_, Vector3D ks_, double alpha_) :
    kd(kd_), ks(ks_), alpha(alpha_)
{ }

Vector3D Phong::getReflectance(const Vector3D& n, const Vector3D& wo, const Vector3D& wi) const {
    double max = std::max(0.0, dot(n, wi));
    Vector3D reflectance = { 0, 0, 0 };
    Vector3D wr = ((n.operator*(dot(n, wi)) * 2.0) - wi).normalized();
    reflectance.x = kd.x * max * dot(wi, n) + ks.x * max * pow(dot(wo, wr), alpha);
    reflectance.y = kd.y * max * dot(wi, n) + ks.y * max * pow(dot(wo, wr), alpha);
    reflectance.z = kd.z * max * dot(wi, n) + ks.z * max * pow(dot(wo, wr), alpha);
    return reflectance;
}
```

