# LAB 1: Basic Ray Tracer

Advanced Computer Graphics

2022-2023

## 1 Introduction

In this lab you will be asked to complete and implement new methods to the Framework that has been provided to you. At the end of the lab, your code should be able to simulate Phong illumination using ray tracing (Fig 1).

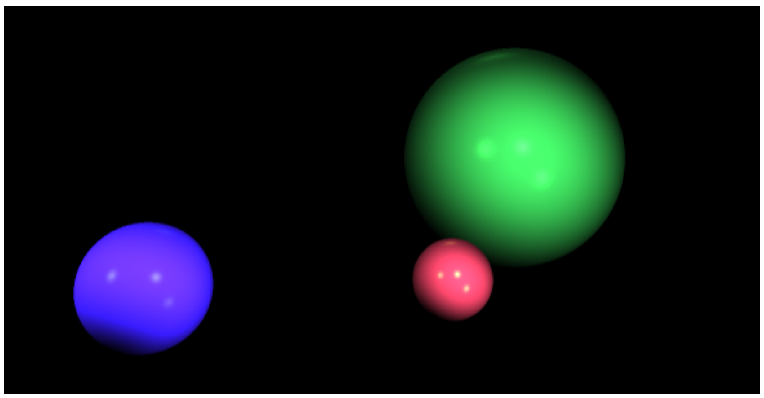

Figure 1: Expected image for Task 5, generated with the direct shader

To start the lab go to the aula global of this course, download the program RTACG - Ray Trace for "Advanced Computer Graphics", open the file RTACG.sln, compile it and execute it. RTACG is composed of a set of classes which will be the base of your first ray tracer. In this assignment, we will only use three of those classes: Shader, Material, and Utils.

## 2 Objectives

This assignment consists of getting familiar with the framework that we provide to you and grasp the basic concepts of ray tracing. At the end of this assignment you should have gained the following competences:

- How to set up a scene.
- How to use the different ray programs: Shadow Rays (or Any Hit Rays) and Closest Hit Ray.
- How to extract relevant geometric information and display it on the screen.
- How to apply ray tracing to compute Phong illumination.

## 3 Introduction to the Framework

In this section, the essential components of the Framework are described, read it carefully and especially, check the code to see how it is programmed and how you need to interact with it.

- **Shader class**: *abstract class* which is the base class for all shaders. Shaders are objects responsible for computing the light which travels along a given ray. The most important feature of the Shader class is that it forces the classes derived from it to implement a *computeColor()* method which, given a ray and the scene information, returns an RGB value corresponding to the light intensity along that ray.
- **IntersectionShader class**: it is an example of a shader derived from the parent Shader class which you will use to complete the first task. The implementation of the *computeColor()* method made by this shader returns a certain color for the rays which intersect a scene object, and another color if no intersection is detected.
- **Lightsource class**: class which implements the concept of point light sources, which are characterised by a position and an intensity.
- **Material class**: abstract class which will be used as base class to derive the materials used in the ray tracer. Note that it forces the derived classes to implement the getReflectance() method which, given the normal $\mathbf{n}$ at a shading point, an incident light direction $\boldsymbol{\omega}_i$ and an outgoing direction $\boldsymbol{\omega}_o$, returns the light arriving from $\boldsymbol{\omega}_i$ reflected in direction $\boldsymbol{\omega}_o$
- **Shape class**: this abstract class, from which common geometric shapes (e.g., sphere, triangle, plan, etc.) must be derived, has two points that must be highlighted for the assignment:
  1. This class has a variable of type *Material** which points to the material of the current shape. This allows to associate a shape with a given material.
  2. Here we have two important methods. The first one called *rayIntersectP()* (where ' P ' stands for predicate function). *rayIntersectP()* returns true if the ray intersects the current object, and false otherwise. This function is useful if we want, for example, to test the visibility of a light-source, where all we want to know is whether or not an intersection exists. However, in other situations, one would like to have information regarding the intersection itself (such as which object was intersected, the position of the intersection point, etc). This second intersection function,*rayIntersect()*, tests the intersection of the ray with the current object and, if an intersection is detected, fills an Intersection object with information about the closest intersection point.

# 4 Assignment

You have to code as many tasks from the list as you can. Each task has a different value depending on its complexity.

## 4.1 Task 1: Painting the Image (1 points)

The goal of this task is to make you familiar with the main rendering loop and the manipulation of the images pixels. The rendering loop traverses all the pixels of the image, and assigns a color value to them. Currently, it just assigns a random color to each of them. However, you should change this behaviour such that the color of each pixel is given by the pixel **x** and **y** coordinates, such that:

$$color = RGBColor(\frac{x}{width}, \frac{y}{height}, 0) \tag{1}$$

where the *width* and the *height* correspond to the image resolution. **Note**: You need to fill the function *PaintImage()*.

Figure 2: Expected image for Task 1.

### 4.2 Task 2: Implement the Intersection Integrator (2 points)

The goal of the second task is to produce an image where pixels are either colored in red, or in black. Given a ray with origin in the camera position and passing through the centre of a given pixel, the pixel will be colored in red if the ray intersects any object of the scene, and black otherwise. Inside the function *raytrace()*, you will find the main ray-tracing loop, which creates the ray for each pixel. You can see an example of such a result in Figure 2.

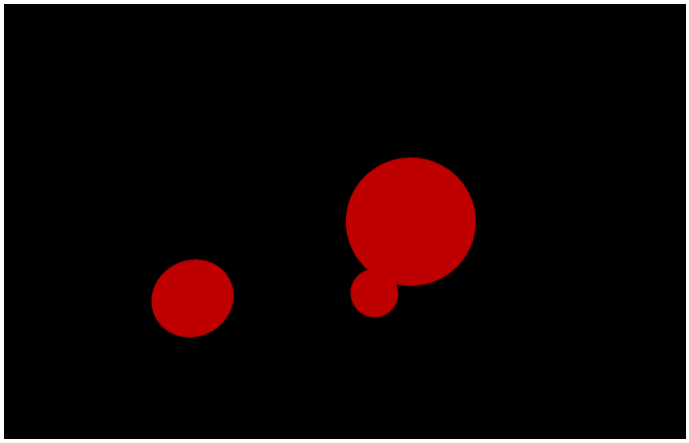

Figure 3: Expected image for Task 2.

***Hint**: to complete the Utils::hasIntersection() function, the method *rayIntersectP()*, which must be implemented by all classes derived from the base class Shape, is particularly useful. **This method has been described in the Introduction!**

### 4.3 Task 3: Implement the Depth Integrator (2 points)

Following the previous exercise you are asked to complete the Utils::getClosestIntersection() function which need to return **closest** intersection from the ray origin.

Once you've done this, you can implement your Depth Intersection Shader, that should return a color computed as:

$$c_i = max(1 - \frac{hitDistance}{maxDistance}, 0) \tag{2}$$

Where $c_i$ is one of the three color components (R, G and B), hitDistance is the distance from the ray origin to the intersection points and $maxDistance$ is a parameter specified to the constructor of the Intersection Shader.

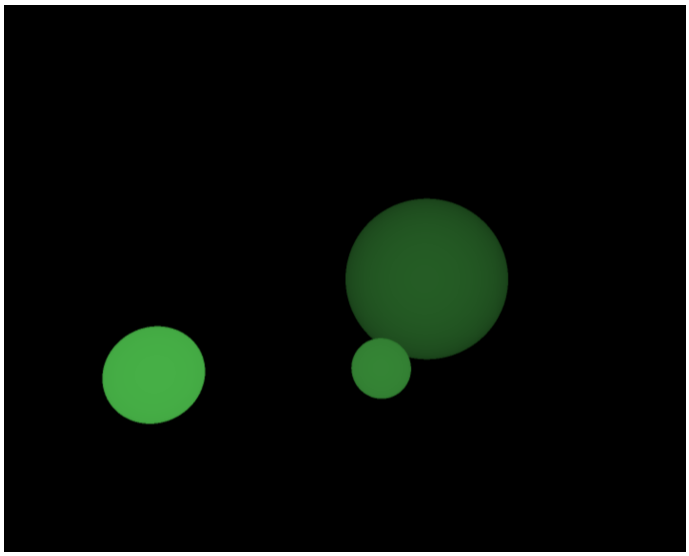

Figure 4: Expected image for Task 3.

***Hint**: to complete the Utils::getClosestIntersection() function, the method *rayIntersect()*, which must be implemented by all classes derived from the base class Shape, is particularly useful. **This methods has been described in the Introduction!**

### 4.4 Task 4: Implement the Normal Integrator (1 points)

Here you are asked to display the normals of the objects placed in the scene. The idea is to learn how to use the geometric information of the object to be able to compute the Phong Integrator.

Here the color of your Normal Shader should be computed as:

$$color = \frac{normal + (1.0, 1.0, 1, 0)}{2} \tag{3}$$

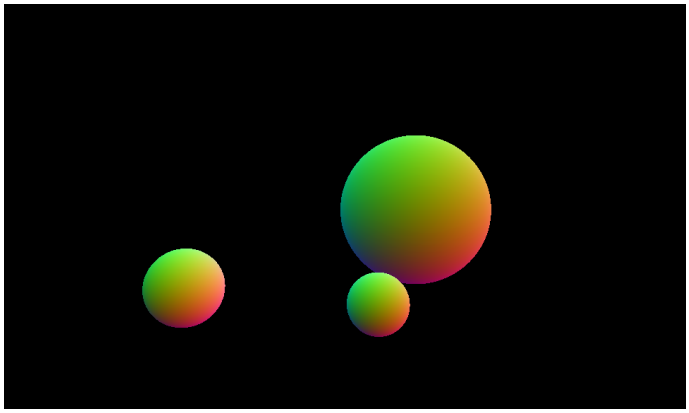

Figure 5: Expected image for Task 4.

## 4.5 Task 5: Implement Phong Integrator (4 points)

In this task you are asked to do the following:

1. Derive a new class called Phong which inherits from the class Material. The Phong class should implement the Phong material described above.
2. Create different Phong materials and associate them with the spheres of your scene.
3. Implement a new class, called DirectShader, which inherits from the parent class Shader and computes the direct illumination at a shading point as described below.
4. Distribute some light sources in the scene and used the direct shader to synthesize an image with direct illumination.

### 4.5.1 Phong Material

In the context of ray tracing, materials are used to model the interaction of the light with the objects of the scene. They specify how light is reflected at a given object, depending on the viewing direction $\boldsymbol{\omega}_o$, on the normal $\mathbf{n}$ at the shading point $\mathbf{p}$, the incident light direction $\boldsymbol{\omega}_i$, and the material properties.

A material of type Phong, as you have seen in the theory class, is characterized by two coefficients $k_d$ and $k_s$ (which give the diffuse and specular object color, respectivelly), and by a third coefficient $n$ (called shininess coefficient) which characterizes the roughness of the material. Given these coefficients, the reflectance $r(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o)$ of a Phong material is given by:

$$r(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o) = k_d (\boldsymbol{\omega}_i \cdot \mathbf{n}) + k_s (\boldsymbol{\omega}_o \cdot \boldsymbol{\omega}_r)^n , \tag{4}$$

where $\boldsymbol{\omega}_r$ is the ideal reflection direction given by

$$\boldsymbol{\omega}_r = 2 (\mathbf{n} \cdot \boldsymbol{\omega}_i) \mathbf{n} - \boldsymbol{\omega}_i \tag{5}$$

### 4.5.2 Direct Illumination Shader

The PointLightSource class has a method which, given a point $\mathbf{p}$ in world coordinates, returns the amount of incident light arriving at $\mathbf{p}$ due to that light source (denoted by $L_i(\mathbf{p})$ in the slides of the theory class). Such a method is called getIntensity(). Take a look at the implementation of this method and verify that, as you have learned in the theory class, the intensity decreases with the square of the distance between $\mathbf{p}$ and the light source.

Now that our ray tracer supports Phong materials and point light sources, we can pass to the last stage of this assignment: the implementation of a direct illumination shader. The direct shader will be used to compute $L_o(\boldsymbol{\omega}_o)$, i.e., the outgoing light leaving a given shading point $\mathbf{p}$ in the direction $\boldsymbol{\omega}_o$ (which is the direction opposite to that of the camera ray). This is achieved by taking into account the contribution of all visible light sources to the illumination at point $\mathbf{p}$, weighted by the reflectance of the material of point $\mathbf{p}$. Note that the reflectance depends on the direction of the light source, which means that Eq. (5) must be evaluated for each visible light source.

Mathematically Phong can be expressed as:

$$L_o(\boldsymbol{\omega}_o) = \sum_{s=1}^{nL} L_i^s(\mathbf{p}) r(\boldsymbol{\omega}_i^s, \boldsymbol{\omega}_o) V_i^s(\mathbf{p}) \tag{6}$$

where:

- $nL$ is the number of point light sources in the scene and $s$ is the index of the current light source;
- $L_i^s(\mathbf{p})$ is the incident light arriving at point $\mathbf{p}$ from the light source with index $s$;
- $\boldsymbol{\omega}_i^s$ is the direction of the light source with index $s$ as seen from $\mathbf{p}$;
- $r(\boldsymbol{\omega}_i^s, \boldsymbol{\omega}_o)$ is the reflectance term given by Eq. (1);
- and $V_i^s(\mathbf{p})$ is the visibility term, which yields:
  ⋆1 if the light source with index $s$ is visible from point $\mathbf{p}$
  ⋆0 otherwise

# 5 Delivery

Create a ZIP file with RTACG Framework, named P1_NIA1_NIA2.zip. The ZIP file must include the PDF of the Lab Report. **Note**: Submission only 1 per group.