Imatge Sintètica - Assignment 3

Ricardo Marques

2017/2018

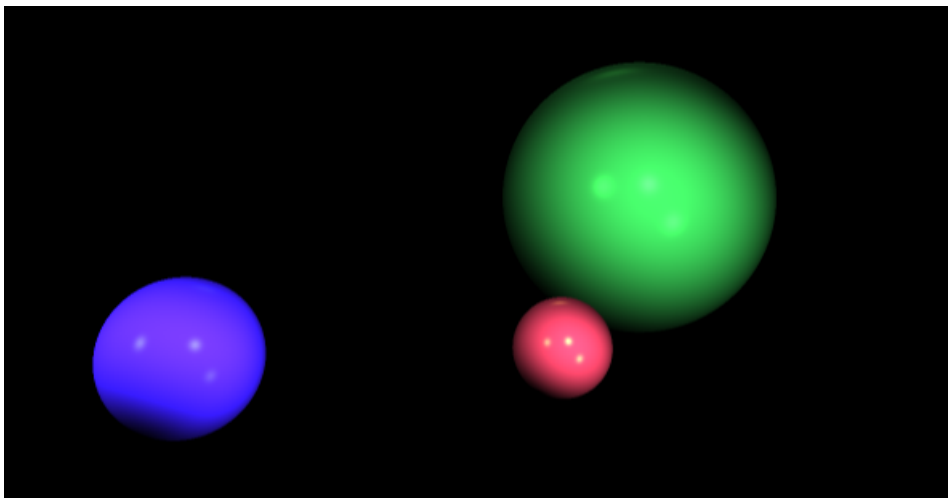

Figure 1: Example of an image generated with the direct shader.

1 Introduction

The previous assignment (i.e., assignment 2) marked a milestone in the development of your ray tracer: using your knowledge about camera rays and ray-sphere intersection you were able to generate your first ray-tracing image, showing that you were able to assimilate and apply basic ray tracing concepts. However, despite the important advancements, the ray tracer you developed so far is still quite basic, and continuing with it until the end of the term would mean that you could not go very far implementing interesting light effects. Therefore, for this assignment, you will be given a new ray tracer skeleton, more complex, better structured, and with some extra features, on which you will work until the end of the term.

In its current state, the new ray tracer can easily be programmed to create a similar image to that which you have created in the end of assignment 2. However, there are some new features which must be mentioned. **Please read carefully the following list** containing the **most important changes**:

- **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 reproduce the image you generated in the last assignment. 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 *diffuse point light source* seen in the theory class. It is characterized 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, is already know of you from the second assignment. Two points must be highlighted in the new version:

  1. As opposed to the old version, this class has now 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. The old version had a single intersection method, 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 lightsource, 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).

     The **new version contains another intersection function** which 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.

Take your time to navigate in the different classes composing the new ray tracer skeleton and to understand their role in the ray tracing framework.

2 Objective

At the end of this assignment your ray tracer should be able to produce an images similar to the one shown in Fig. 1. To reach this goal you will have to **complete the skeleton of the functions corresponding to Assignment 3** in RTIS V2. The rest of this assignment will guide you through this task by

giving you a set of exercises which will allow you to progressively complete the missing parts of the RTIS V2. The main steps can be summarized as follows:

1. Use RTIS V2 to reproduce an image of the same type as the one generated in the last assignment (where the pixel is painted in red if an intersection is detected, and black otherwise).

2. Develop a simple Depth shader to visualize the distance traveled by the ray until an intersection is detected. This will help you to get familiar with the Intersection class, used to store information about the ray-object intersection.

3. Implement a Phong material class by deriving it from the parent class Material.

4. Implement a Direct shader class which computes the color of the shading points using direct illumination.

3 Intersection Shader

3.1 Context

One of the main changes introduced in RTIS V2 is the ability to deal with multiple objects when computing ray-object intersections. This is done by two functions which you can find in the name space Utils (both are incomplete):

- Utils::hasIntersection(): This function receives as input a reference to a ray, as well as the scene geometry (i.e., a vector of pointers to objects of type Shape, passed by reference). It returns true if any intersection is detected, and false otherwise. Note that, that this function does not need to find the closest intersection along the ray.

- Utils::getClosestIntersection(): This function receives three arguments (all of them passed by reference): a ray, a vector of pointers to objects of type Shape, and an object of type Intersection which is used as output parameter. If the ray intersects the scene geometry, then the function returns true and the Intersection object is filled with the details of the closest intersection found. Otherwise, the function returns false.

3.2 Your Task

Complete the Utils::hasIntersection() function following the description of this function made in section 3.1. Once you have accomplished this task, use an IntersectionShader to sintesize an image and to validate the result. You should obtain an image similar to Fig. 2.

> **Hint 1:** 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!**

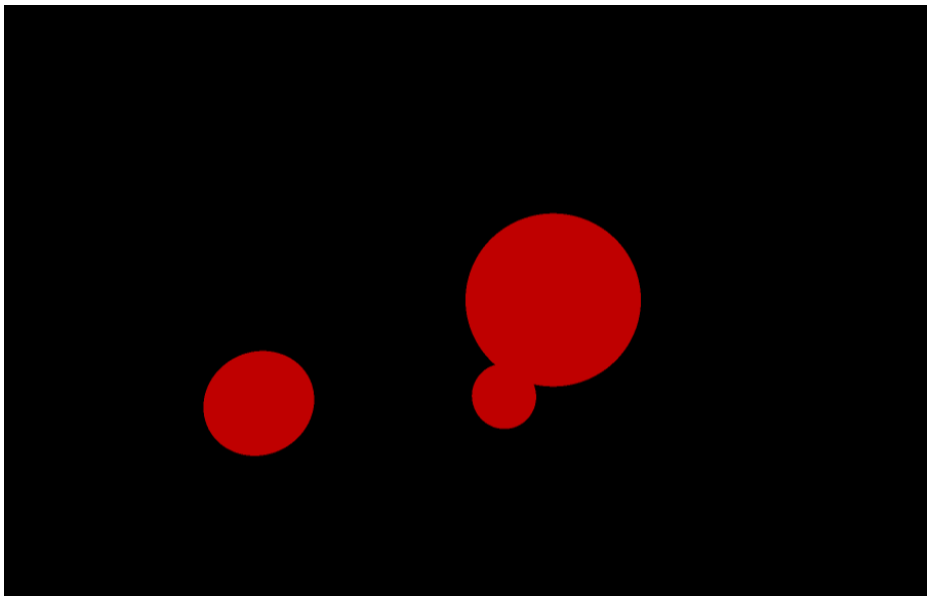

Figure 2: Example of an image generated with the intersection shader.

4 Depth Shader

4.1 Context

In many situations, simply knowing if the ray intersects or not the scene geometry is not enough to accomplish the task of rendering an image. For example, to compute the color of a shading point corresponding to the closest intersection of a camera ray with the scene geometry, one needs to know the type of material of the intersected object and the normal at the shading point. Such information can be obtained by calling the function Utils::getClosestIntersection() (which is currently incomplete).

4.2 Your Tasks

4.2.1 Task 1

Complete the Utils::getClosestIntersection() function following the description of this function made in section 3.1.

> **Hint 2:** 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!**

4.2.2 Task 2

The next step after having accomplished Task 1 is to test the getClosestIntersection() function. For this, you will develop a simple shader, called DepthShader,

which allows to visualize the distance traversed by a ray until the closest intersection with the scene geometry. The interface of the DepthShader is already defined in the file depthshader.h. **Please read it carefully.**

Your task is to implement the single DepthShader constructor available to the programmer, and the function DepthShader::computeColor(). The former is trivial. To implement the latter, you need to know the following:

- If the ray does not intersect the scene geometry, then the function should return the background color as result.
- If an intersection exists, then two cases can occur:
  1. The distance between the ray origin and the closest intersection is greater or equal than maxDist, in which case the function returns the background color.
  2. The distance between the ray origin and the closest intersection is smaller than maxDist, in which case the function returns:

```
return color * (1.0 - depth/maxDist);
```

Finally, synthesize an image using a DepthShader constructed as follows:

```
Shader *shader = new DepthShader (Vector3D(0.4, 1, 0.4), 8, bgColor);
```

The obtained result should be similar to that shown in Fig 3

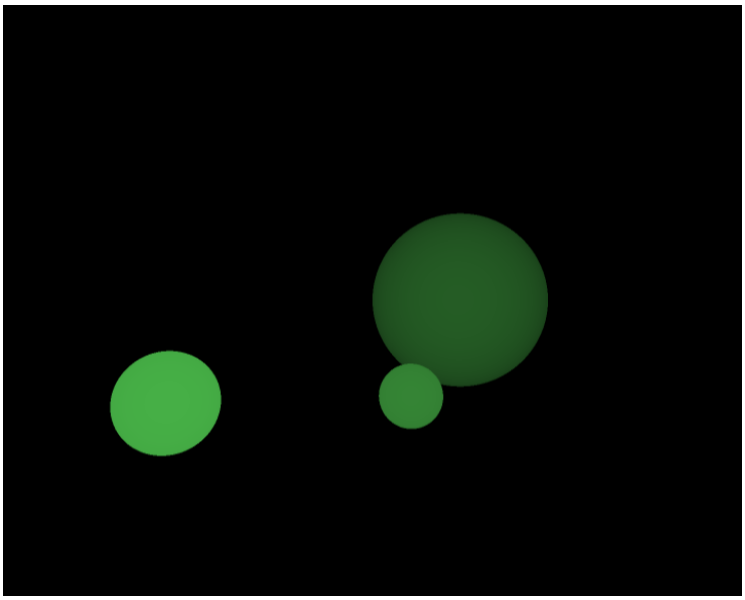

Figure 3: Example of an image generated with the depth shader.

5 Phong Material

5.1 Context

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:

$$r(\boldsymbol{\omega}_i, \boldsymbol{\omega}_o) = k_d \left(\boldsymbol{\omega}_i \cdot \mathbf{n}\right) + k_s \left(\boldsymbol{\omega}_o \cdot \boldsymbol{\omega}_r\right)^n , \tag{1}$$

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

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

5.2 Your Task

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.

6 Direct Illumination Shader

Some motivation: `https://www.youtube.com/watch?v=Qx_AmlZxzVk`

6.1 Context (back to the reality of RTIS V2)

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. (1) must be evaluated for each visible light source. Mathematically, this

can be expressed as follows:

$$L_o(\boldsymbol{\omega}_o) = \sum_{s=1}^{nL} L_i^s(\mathbf{p})\, r(\boldsymbol{\omega}_i^s, \boldsymbol{\omega}_o)\, V_i^s(\mathbf{p}) \tag{3}$$

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

6.2 Your Task

1. Implement a new class, called DirectShader, which inherits from the parent class Shader and computes the direct illumination at a shading point as described above.
2. Distribute some light sources in the scene and used the direct shader to synthesize an image with direct illumination.