KTH Stockholm
CSC :: CST
Introduction to Visualization and Computer Graphics, Fall 2016,
DH2320
Tino Weinkauf, Himangshu Saikia, Gregorio Palmas

Homework assignment No. 05

Due December 1, 2016

Task 5.1: Raytracer: Intersections of a Ray with Geometric primitives **16 P**

In this task you will have to program a Raytracer by extending the methods in the classes within the *raytracer* subdirectory. Prepare the solution file by extracting the zip file and running cmake as in previous assignments. Now set *raytracer_task1* as the startup project and run the program. If all goes well, you should see a blue screen. Implement the following functions:

- For the unit-sphere-ray intersection test (8 P)

  ```
  bool Sphere::closestIntersectionModel(const Ray &ray, double maxLambda,
  RayIntersection& intersection) const
  ```

- For the triangle-ray intersection test (8 P)

  ```
  bool Triangle::closestIntersectionModel(const Ray &ray, double maxLambda,
  RayIntersection& intersection) const
  ```

You will find these functions in the files `raytracer/Sphere.cpp` and `raytracer/Triangle.cpp`. If an intersection is found, the function shall return `true`, else it shall return `false`. In case your function returns `true`, both functions should construct objects of type `RayIntersection` and assign them to the parameter `&intersection`. You can refer to the implementation of the Plane-Ray intersection in the file `raytracer/Plane.cpp`.

The constructor of the class `RayIntersection` takes the following parameters :

- `const Ray &ray`: constant reference of the Ray with which an intersection is found.
- `std::shared ptr<const Renderable> renderable`: Reference in the form of a smart pointer to the Object with which the intersection occurred. Use the term `shared_from_this()` here.
- `const real lambda`: Ray parameter where $\lambda \in [0, maxLambda]$.
- `const Vec3d &normal`: constant reference to the normal direction at the point of intersection on the Object's surface.
- `const Vec3d &uvw`: ignore this and assign the Null Vector (0, 0, 0).

In the file `raytracer_task1.cpp` you will find a test scene for our Raytracer. The scene includes three spheres, and a triangle against a blue background. After you implement the aforementioned functions, the result should look like the image in Figure 1.

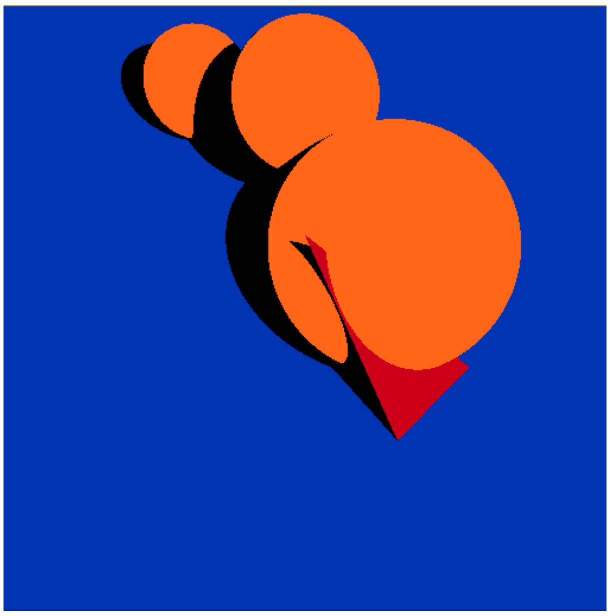

Figure 1: Result of Task 1

Task 5.2: Raytracer: Phong Shading 8 P

Set *raytracer_task2* as the startup project. If you correctly implemented the method for the Ray-Sphere intersection you should see an output similar to Figure 2.

The scene in the file `raytracer_task2.cpp` consists of 3 point light sources, 4 spheres and a plane. All of these objects are assigned a `PhongMaterial` material. Your task is to implement the Phong shading model.

In the file `raytracer/PhongMaterial.cpp` implement the function

```
Vec4d PhongMaterial::shade(const RayIntersection& intersection,
    const Light& light) const
```

for the reflectance at the point of intersection between a ray and an object. At the moment a simple Lambertian model is implemented with ideal diffuse reflection. You are supposed to implement the Phong model. Use the specular coefficient $\rho_s = 0.04$. The return value should be color (red, blue, green, 1). If all goes well, you should see a result similar to Figure 3. Notice the nice specular highlights of the point light sources on the surfaces of the spheres.

Task 5.3: Raytracer : Rendering Triangle Meshes (Bonus) 5 BP

In the file `raytracer_task2.cpp` uncomment the line

```
std::shared_ptr<rt::Scene> scene = makeMeshScene("assets/smooth_dragon.obj",showArrows);
```

and comment out the line

```
std::shared_ptr<rt::Scene> scene = makeTask2Scene();.
```

This allows us to load a Triangle Mesh in the Wavefront-OBJ file format into our framework. Your task is to load any other triangle mesh instead of the dragon, one which you might have created yourself, or downloaded from the internet. Use proper positioning of the object(s), light sources and camera in the scene. Set the material(s) accordingly. Reasonable computation times will be achieved for scenes with less than 500,000 triangles. If you did everything correctly, the result should look like Figure 4 (for the dragon).

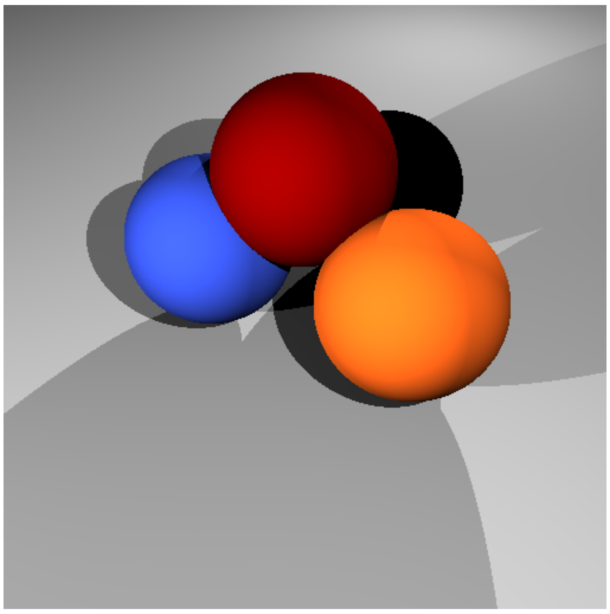

Figure 2: Initial setup for Task 2

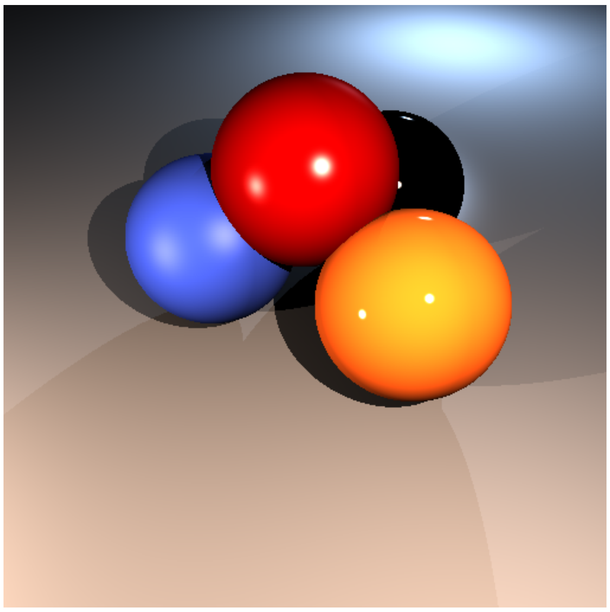

Figure 3: Task 2 result with Phong illumination model

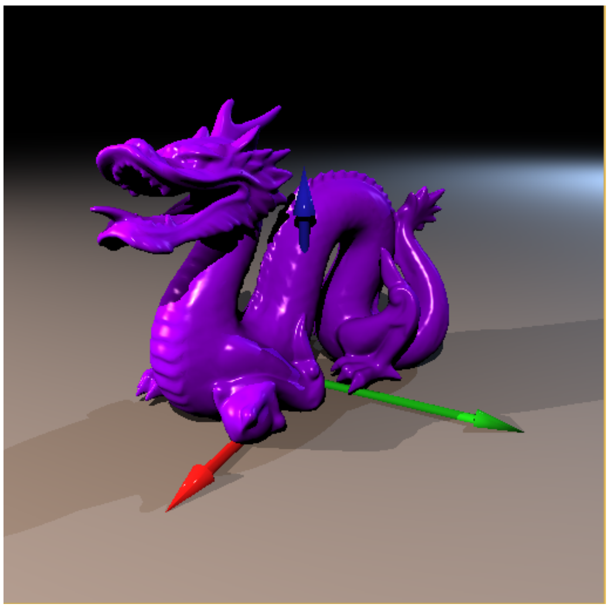

Figure 4: Bonus task result with the Dragon model.