Ray Tracing Assignment 1: Ray-Object Intersection and the Phong Illumination Model

February 21, 2022

Computer Graphics

1 Theoretical questions (1.0 point)

The answers to the theoretical questions must be submitted as a PDF file. We recommend using LaTeX, but we allow other submissions (including handwritten ones) as well. This question is parametrised by the digits of one of your group member's student number. If your student number is $s7654321$, then $g = 7$, $f = 6$, $e = 5$, $d = 4$, $c = 3$, $b = 2$, $a = 1$. For all theoretical questions please explain your answers. If applicable, show your mathematical derivations.

Consider the following 2D scene description; see Figure 1. The eye/camera is at the origin $\mathbf{O} = [0, 0]$, the white point light source is at $\mathbf{S} = [4 + \frac{a}{10}, 12 + \frac{b}{10}]$, and the only geometric object in the scene is a white circle with radius $r = 4$ centered at $\mathbf{C} = [16 + \frac{c}{10}, 8 + \frac{d}{10}]$.

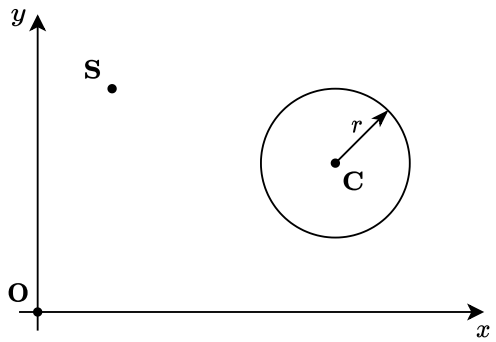

Figure 1: *A 2D scene with point light source **S** and a circle with radius r centered at **C**.*

1. (0.4 points) Consider the ray $\mathbf{O} + t\mathbf{W}$, where $\mathbf{O}$ is the origin and $\mathbf{W} = [2, 1]$. Determine the point on the circle that will be observed from this ray.

2. (0.2 points) Following the Phong Illumination model, determine the point on the circle with the largest *diffuse* component I_D.

3. (0.4 points) Describe which points on the circle receive no light (i.e. where $I_D = I_S = 0$). Also give the points where the circle transitions from lit to non-lit.

2 General instructions

In the following exercises you will be asked to repeatedly add new functionality to an existing ray tracing application. The ray tracer should always be able to render the scene files from previous (sub-)assignments and produce the correct images. In other words, adding new functionality should not break the existing functionality.

If you work with a lab partner, then both of you will need to **be in a group on Themis before submitting**. In order to do so,

- visit `https://themis.housing.rug.nl`,
- go to *Courses* → *2021–2022* → *Computer Graphics*,
- have one student create a group and the other one join that group.

After completing each sub-assignment, create a **zip** file containing:

- `CMakeLists.txt`,
- the `Code` folder.

Do **not** include any other folders, executables, a README file or rendered images. Submit the created zip file to Themis. In total, 11 points can be obtained.

3 Getting started

In this part of the assignment you will set up the environment for your ray tracer implementation. The framework provided is written in C++.

Tasks:

- Download the source code of the ray tracer framework from Nestor. It includes a CMakeLists.txt file for use with CMake. This build setup should work on both the LWP systems and on Themis.
- Look at the included README file for a description of the source files and how to compile and run the ray tracer.
- The folder `Scenes` contains one or more example scene files for each sub-assignment on Themis. The folder `other` contains fancier scene files not used in any sub-assignment, but which you can render after completing the assignment to show the full capabilities of your ray tracer.
  You are encouraged to create your own scene files to test your implementation.
- Compile the application and test whether it works. Using the supplied example scene `other/scene01.json` the image in Figure 2 should be created.
- The ray tracer currently renders spheres using a fixed radius, and its color is defined in its `Material` data member. We will change this later.

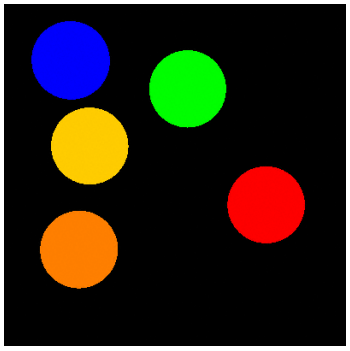

Figure 2: *`scene01.json`, as rendered using an unmodified framework.*

- Look at the source code of the classes to understand the application. Of particular importance is the file `triple.h` which defines mathematical operators on vectors, points and colors. The actual ray tracing algorithm is implemented in `scene.cpp`. The JSON-based scene files are parsed in `raytracer.cpp`.
- While you are free to create your own scene files, do **not** modify the constants in the source code, such as the resolution of the output image or the attribute names used in the JSON parsing. If these constants have changed, Themis may no longer be able to judge your submission.

4 Ray tracing with spheres and Phong illumination

In this assignment your application will produce a first image of a 3D scene using a basic ray tracing algorithm. The intersection calculation together with normal calculation will be the groundwork for the illumination.

- For now, your ray tracer only needs to support spheres. Each sphere is given by its center, its radius, and its surface parameters (see `material.h`).
- A point light source is defined by its position (x, y, z) and color (r, g, b). In the example scene `other/scene01.json` a single white light source is defined.
- The viewpoint is defined by its position (x, y, z). To keep things simple the other view parameters are static: the image plane is at $z = 0$ and the viewing direction is along the negative z-axis (you may improve this later).
- The scene description is read from a file in `raytracer.cpp`.

4.1 Sphere intersection and the ambient component (1.0 point)

The function `Sphere::intersect` in the file `sphere.cpp` contains a placeholder implementation for sphere intersection. Replace it with a real intersection calculation, with the function correctly returning either `Hit::NO_HIT()`, or `Hit(t, N)` with the correct value for `t`. The normal `N` can remain at its default value, i.e. $(0, 0, 0)$, for now.

In the function `Scene::trace`, the returned color is the material color. Replace this by returning the ambient component I_A of the Phong illumination model. This component should not be influenced by any lights in the scene.

Figure 3: *Spheres with different radii. Ambient component only.*

Figure 3 shows sphere intersection and ambient lighting for two different scenes.

4.2 Sphere with the ambient and diffuse components (1.5 points)

In this sub-assignment, we will add the diffuse component to the lighting implementation. As the diffuse component uses the normal at the point of intersection, the normal calculation has to be implemented first.

4.2.1 Normal calculation

Complete the function `Sphere::intersect` in the file `sphere.cpp` by implementing the calculation of the normal `N`. For the illumination calculations, it is necessary that the normal is directed towards the ray's origin, i.e. $\mathbf{N} \cdot \mathbf{V} \geq 0$. Ensure that this is the case in your implementation. As the normal is not yet used by the ray tracing implementation, the ray tracer should produce identical images as before.

Before adding the diffuse component, it is useful to first test the normal calculation by (temporarily) implementing normal shading. In normal shading, the normals of an object are visualized by mapping them to a color.

Extend `scene.cpp` to map the normal to a color using this equation:

$$\mathbf{C} = \frac{\mathbf{N} + 1}{2},$$

where $\mathbf{C}$ is the color and $\mathbf{N}$ is the normal. Note that this maps the range of a normal-component, i.e. $[-1, 1]$, to the range of a color-component, i.e. $[0, 1]$. Return $\mathbf{C}$ as the result of the `Scene::trace` function, thus (temporarily) ignoring the ambient component.

Rendering `1.json` should produce the image in Figure 4.
If the normals have the correct values, then **remove the normal mapping** and continue.

4.2.2 Adding the diffuse component

For this sub-assignment, you may assume that the scene contains exactly one light source.

The normal, as implemented in Section 4.2.1, can now be used in the function `Scene::trace` in the calculation of the diffuse component I_D of the Phong illumination model.

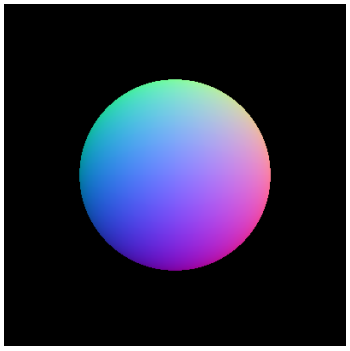

Figure 4: *Sphere with normal mapping.*

Modify the function `Scene::trace` to return the sum of the ambient and diffuse components.

Figure 5 shows an example of using both the ambient and the diffuse components.

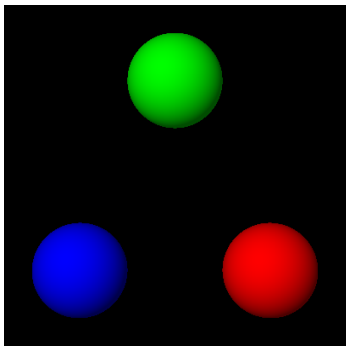

Figure 5: *Spheres with ambient and diffuse shading.*

4.3 Sphere with Phong illumination (1.5 points)

For this sub-assignment, you may assume that the scene contains exactly one light source.

Extend the shading calculations in `Scene::trace` with the specular component I_S, thereby completing the implementation of the Phong illumination model for one light.

An example result of rendering spheres with Phong shading is shown in Figure 6.

4.4 Sphere with multiple lights (1.0 point)

Extend the `Scene::trace` function such that the ray tracer now takes all light sources into account. As in Section 4.1, the ambient component should not be influenced by any lights. Note that the ray tracer should also produce the correct output for 0 light sources.

Figure 7 shows an example scene with multiple lights.

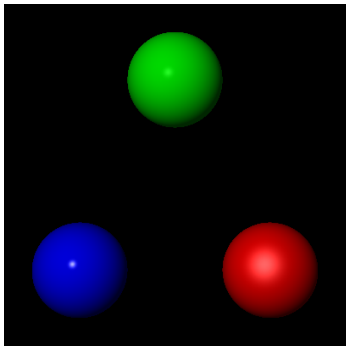

Figure 6: *Spheres with Phong shading.*

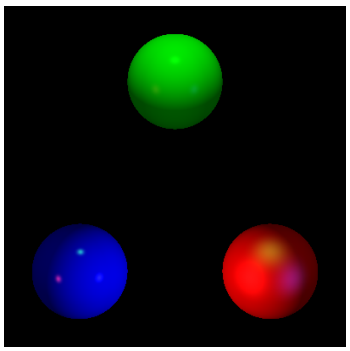

Figure 7: *Spheres with Phong shading and multiple lights.*

5 Additional geometry

This part of the assignment consists of adding support for additional shapes to the ray tracer, including support for a mesh. All scenes in this section should be rendered using Phong illumination with support for multiple lights.

5.1 Triangle (2.0 points)

The ray tracer framework contains two files pertaining to the triangle shape: `triangle.h` and `triangle.cpp`. Complete the constructor `Triangle::Triangle` by calculating the normal and storing it in `N`. Implement the intersection function `Triangle::intersect` to either return `Hit::NO_HIT()` or `Hit::Hit(t, N)`. Note that, just as for the sphere in Section 4.2.1, the normal must be oriented towards the ray's origin.

Figure 8 shows an example scene comprised of three colored triangles.

5.2 Mesh (1.5 points)

Pre-condition: the ray tracer must be able to render triangles.

Since meshes are typically sets of triangles, it is now possible to implement meshes using the mesh "shape". As you can see in the provided scene files and in `raytracer.cpp`, a mesh shape consists of a position, scale and path to the `.obj` file. The mesh shape, as implemented in `mesh.h` and `mesh.cpp`, works as follows. The vertices (see `vertex.h`) are read in groups of three from the `.obj` file and converted to `Point`'s for use in the ray tracer. For each group of three `Point`'s, a triangle shape is constructed and added to the

Figure 8: *Three colored triangles.*

mesh shape's `std::vector`.

Implement the intersection function `Mesh::intersect`.

The mesh as a whole can be transformed by transforming the individual `Point`'s before these are formed into triangles and added to the `std::vector`. Implement:

- Non-uniform scaling. The factors with which to scale are in `Vector scale`. Scale the object in the x, y and z dimension using `scale.x`, `scale.y` and `scale.z`, respectively.
- Rotation around three axes. The angles (in radians) for rotation around the x-, y- and z-axis are given in `Vector rotation`. First apply rotation around the x-axis, then around the y-axis, and finally around the z-axis.
- Translation. To this end, translate each `Point` by the values given in `Point position`.

Please note:

- The OBJLoader class is the very same class used in the OpenGL assignments. Observe that the normals are not loaded from the `.obj` file, but that the normals are calculated in the constructor of the triangle shape.
- The provided scene files and objects should render within a couple of seconds. However, larger meshes may take a long time to render.
- **Optional!** If you want, you may easily speed up the ray tracing using multithreading using OpenMP. Add:
  `-fopenmp`
  to the `CXX_FLAGS` in your `CMakeLists.txt` file and add
  `#pragma omp parallel for`
  just before the outer loop in `Scene::render()`.

Figure 9 shows an example of a rotated mesh.

5.3 Planar quad (0.5 points)

Implement the planar quad shape by completing the `Quad::Quad` constructor and the `Quad::instersect` function. Make sure the normal `N` is oriented towards the ray's origin, as in Section 4.2.1.

Figure 10 shows an example scene consisting of multiple planar quads.

Figure 9: *A rotated goat mesh.*

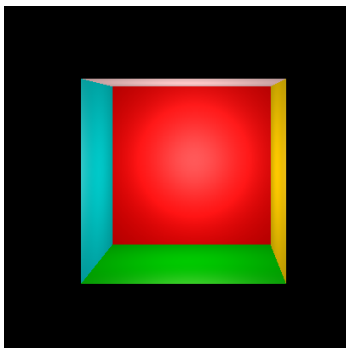

Figure 10: *An open cube consisting of multiple quads.*

5.4 (Bonus) Closed cylinder (1.0 point)

Implement the closed cylinder shape by completing the `Cylinder::intersect` function. The cylinder is defined by `Point position` (the base of the cylinder), `Vector direction` (its direction and length) and its radius. Make sure the normal `N` is oriented towards the ray's origin, as in Section 4.2.1. You may assume that the cylinder is axis-aligned.

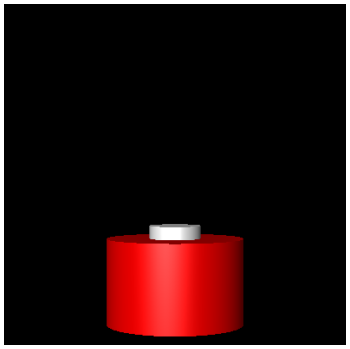

Figure 11: *Two intersecting closed cylinders.*

Figure 11 shows an example scene consisting of two intersecting closed cylinders.

6 Ideas for the competition

- Create your own scenes. Build your own composite objects using triangles and quads. Even with just spheres you can build interesting scenes.
- Implement additional shapes, such as a cone or torus.
- When rendering a mesh, use the normals provided by the `.obj` file instead of calculating the normal of the (planar) triangle.
- Speed up the ray tracing by implementing binary space partitioning.