Ray Tracing

ab-gh

1 December 2021

Task: To create an advanced optical simulation program in C++. Using the principles of *Ray Tracing*, *Object-Oriented Design*, and the material covered in PH3170, this project aims to produce a realistic image of a simulated scene, including accurate objects and lighting.

1 Introduction and Theory

To start the project, I set out a series of objectives for the final code, and a set of milestones to progress through. In order to fulfill the requirements of the task, my code needs to be able to:

- Allow a user to specify the geometry, location, and material properties of primative shapes in the scene.
- Allow a user to specify the location and intensity of sources of light.
- Simulate the object's interactions with the light sources and other objects in the scene using *reverse ray tracing*.
- Produce an image of the simulated scene, as if a camera was taking a photograph.

To achieve these aims, I set out to complete the following milestones, roughly in order. By breaking the project down into smaller components, I not only ensure that I don't waste time working on one feature when an underlying feature is broken, but also lend my code-writing process to Object Oriented Design (OOD) principles.

1. Construct a fully-functional but abstract Vector data type, capable of defining vectors and points in 3D cartesian space, and computing all relevant vector operations
2. Construct a "virtual camera", which sends rays out into the "world", and calculates the ray's interactions with objects and lights
3. Output the intensity and colour of each ray as a pixel in an image
4. Define primative objects, classes which contain the object's geometry and properties, as well as functions to calculate the intersections between its self and a camera ray, and a function to calculate the surface normal of the shape at any point.
5. Define light sources, with an intensity and position
6. Construct a "rendering engine", capable of simulating various types of illumination, and linking the virtual camera to the "world" (scene) of objects in front of it.

1.1 Ray Tracing Theory

To produce a realistic depiction of the scene, this project uses a technique called *reverse ray tracing.* In a real physical scene, the illumination of objects can be approximated as light sources emitting rays which interact with the object, reflecting light to a new ray direction. If one of these light rays, after however many reflections, hits a sensor (be it an eye or camera), it is “seen” by the observer. In a computational simulation, a tiny fraction of the rays emitted from a source will actually make it to the observer, so this process is very inefficient.

To combat this, *reverse* ray tracing is used, with the same principles, except the observer emits a “sight” ray, which reflects according to the same laws, until it hits a light source. To produce the image, the observer “looks” through a viewport - a rectangular boundary in space - similarly to a camera sensor. Sight rays are constructed with a vector from an “eye” (focus of the observation plane) to each pixel on the viewport, which is then projected into the scene, and the reverse ray tracing begins.

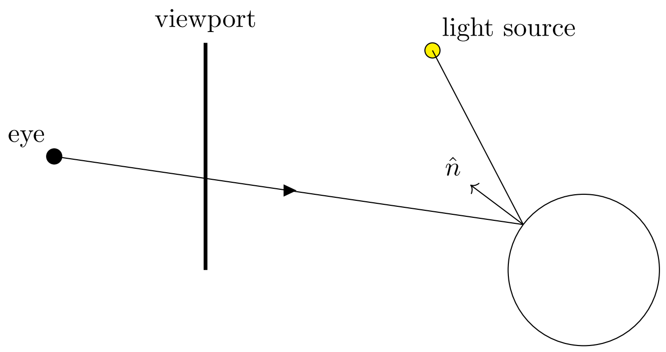

Figure 1: Demonstration of reverse ray tracing, reflecting a sight ray from the eye through the viewport off a sphere.

1.2 Object Oriented Design

This project required a lot of different components to work together accurately, yet still abstract enough to be flexible, allowing the development of more shapes, interactions, and materials in future. Because of this, applying the principles of Object Oriented Design was essential. Each component of the project was “compartmentalized” in its own header file, using inheritance where possible. Each class interacts with the renderer, which produces the final image.

2 Implementation

This section explains the implementation of the ray tracing simulator, following the process outline in the milestones above.

2.1 Vector base class

The first task of the implementation was to construct a fully-featured vector class. Defined in `Vector.h`, the `Vector` class serves as the basis for almost every other element of the simulator. As such, it defines an extensive list of methods and operations for vectors.

```cpp
class Vector {
  private:
      double vector[3];
  public:
      Vector() : vector{0, 0, 0} {};
      Vector(double x, double y, double z) : vector{x, y, z} {};
      ...
  };
```

Operations for vectors are also defined outside of the `Vector` class, as inline functions, so that they can be accessed outside of a Vector:

```cpp
inline Vector operator+(const Vector &u, const Vector &v) {
    return {u.x() + v.x(), u.y() + v.y(), u.z() + v.z()};
}
...
```

In addition to the basic mathematical operations, the header file defines the dot and cross product (`dot(u,v)`, `cross(u,v)`), unit vector (`unit(u)`), as well as reflections (`reflect(i,n)`) and refractions (`refract(i,n,eta)`). The `Vector` class also has three aliases, which help to make their specific use clearer in context: `V3`, `P3`, `RGB`.

2.2 Camera, viewport, and scene

Now that the basic vector class is implemented, the virtual world of the simulation can be built. The file `Camera.h` defines the `Camera` class, which includes the eye and viewport. For simplicity, this has default values of a 16:9 aspect ratio image, with 1000 pixels across, which translates to a height of 2 in the simulated world. The focal length is defined as 1, which means the "eye" is one unit behind the centre of the viewport. The viewport is centered on the origin, looking down the negative z axis. This geometry allows you to consider the image as if it were an x, y plane, with depth as $-z$.

Next, the `Scene.h` file defines the simulated world as the class `Scene`. This is a very simple class, with two `std::vector` attributes: one to hold the list of objects in the scene, and one to hold the list of light sources. Two methods are also defined, which add objects and light sources to the scene once created.