UM EECS 487
Lab 6: Ray Tracing

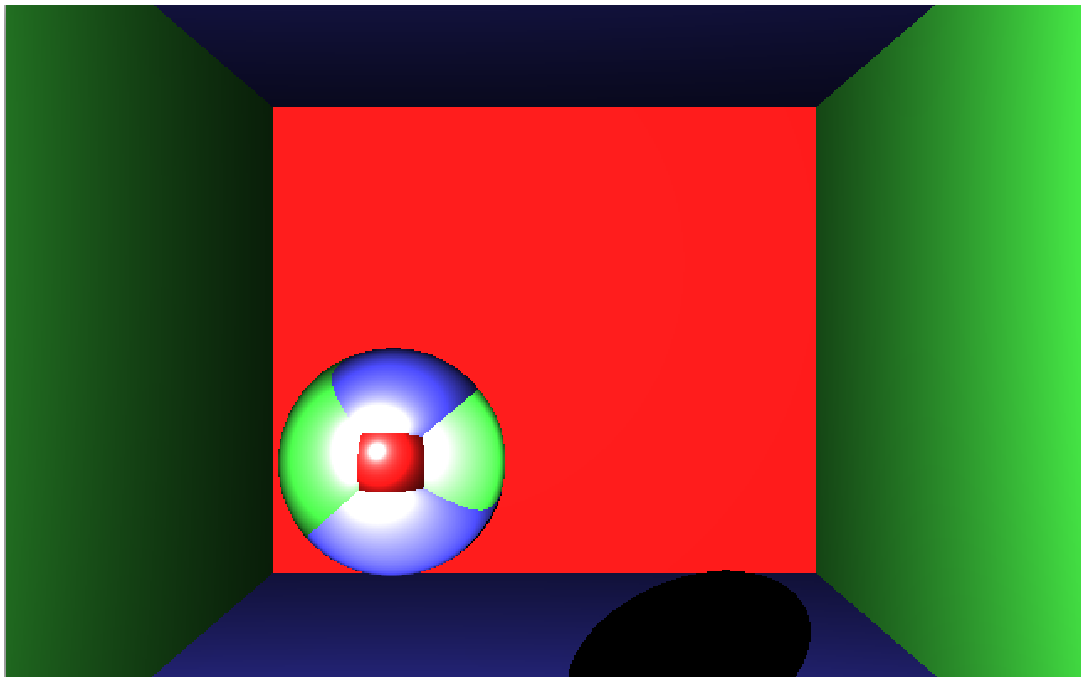

FIGURE 1. Screen shot of a sphere reflecting the box around it. The image shows reflection, lighting, and simple, hard shadows.

1. Introduction

In this lab you will implement basic ray-tracing, as shown in Figure 1. A perfectly reflecting sphere is suspended within a box. The front and back walls are red, the left and right walls are green, and the top and bottom are blue. A purely-diffuse, point light-source illuminates the scene from over the viewer's left shoulder. Observe the following:

(1) The sphere reflects the box around it and the reflection is modulated (deformed) to the curvature of the sphere.
(2) The sphere casts a shadow on the surface opposite the light,
(3) The attenuation of illumination on walls is captured convincingly,
(4) The reflection of the back wall (behind the transparent viewer) appears as the red quadrilateral in the center of the sphere.
(5) The viewer is between the sphere and the front wall, yet is not reflected in the sphere, while the front wall is. Who has no reflection?

2. Learning Objectives

(1) The mathematical creation of (reflected) rays,
(2) computing points of intersection with geometric objects, and
(3) writing a simple, recursive function to trace the path of a ray in a depth-first search (DFS) manner.

3. Support Code Documentation

Points and 3D-vectors are represented using the `XVec3f` class that should be familiar to you by now. A ray is represented as a direction $\vec{d}$ and the eye point $\vec{e}$ through which the ray passes. Each `Ray` class has a parameter t which is used to generate points $\vec{r}$ along the ray using the formula

$$\vec{r} = \vec{e} + t\vec{d}.$$

The `Ray` class and all the other classes in this lab are defined in `scene.h`. Unless otherwise noted, all the methods below are defined in `scene.cpp`.

The `Sphere` class represents the properties of a sphere. Useful methods are:

(1) `XVec3f Sphere::unit_normal(XVec3f const&)` (in `scene.h`) which returns a unit vector normal to the sphere through the point given,
(2) `bool Sphere::is_intersecting(Ray const&)` which tells you if the given ray intersects the sphere,
(3) `double Sphere::intersect(Ray const&)` which returns the t parameter for the given ray at the point of intersection. *You are to implement this function.*

The `Plane` class represents the walls. It consists of a point in the plane, a normal to the plane, and the material that the plane is made of which is used for (diffuse) lighting calculations. Useful methods are:

(1) `bool Plane::is_intersecting(Ray const&)` which tells you if the given ray intersects the plane,
(2) `double Plane::intersect(Ray const&)` which returns the t parameter for the given ray at the point of intersection. *You are to implement this function.*

The `Color` class is a wrapper for a triplet of floats that represent the RGB color just like in OpenGL. The `Material` class is used in lighting calculations. The `Light` class represents the positional, point light source. There is only one light in the scene. The light consists of its position and ambient, diffuse, and specular contributions (though only diffuse is used). The method useful to you is `Light::pt(XVec3f const& p, XVec3f const& n, Material const& mat)` which returns the `Color` due to illumination of the point $\vec{p}$ (on a surface) with normal $\hat{n}$ and material `mat`. This function performs a diffuse lighting calculation to determine the color of the walls.

4. Instructions

- First implement the functions that return the point of intersection of a ray with a plane and with a sphere. These should return the t parameter for the ray at the point of intersection (described later). The actual point can then be determined by calling `Ray::pt(t)`.
  - **Plane**: Modify `Plane::intersect()`.
  - **Sphere**: Modify `Sphere::intersect()`.
  - Both classes have an `is_intersecting(Ray const&)` method that you may find helpful for this task.
- Implement the function `raytrace()` in `raytrace.cpp`. This function is called by `display()` and is responsible for refreshing the window. The `display()` function creates a ray from the eye-position through each screen point and calls `raytrace()` with this ray. Your task is to implement `raytrace()` with a recursive formulation.

- If we've already touched the sphere or the ray doesn't intersect the sphere then find out which wall the ray is intersecting and obtain its color. To do this,
  (1) call `Plane::intersect(ray)` for each of the walls so that you get the t parameter for the ray at which the intersection occurs. Negative t tells you that the plane is behind the origin of the ray (the eye). Also, t is an indicator of how far one must travel along the ray to reach the point of intersection.
  (2) Once you get the t, computing the actual world point can be done by calling `Ray::pt(t).`
  (3) Only the t that is within the visible world is useful for rendering. Keep track of the wall corresponding to the visible t also.
  (4) Calculate the color at the intersection point of the ray and the wall by calling `Light::pt(XVec3f, plane-normal, plane-material)`.
  (5) Hard shadow: if the ray from the light source to point on the wall intersects the sphere, the sphere casts a shadow. Otherwise the wall reflects light in the usual way. You may want to implement the next task before implementing this one.
- If the ray intersects the perfectly-reflecting sphere, then
  (1) determine the point of intersection,
  (2) determine the normal at that point,
  (3) use this to determine the reflected ray, and
  (4) call `raytrace()` with this new ray to see which object is contributing to the illumination of that point on the sphere.

5. SLOW...

When you finish the tasks above, use the keyboard shortcuts 'h', 'j', 'k', 'l', 'w', 's' to interactively move the sphere around in the scene. See how much time the scene takes to render. Ray-tracing renders scenes in a manner that is opposite to OpenGL's (which proceeds from primitives to pixels). Hardware ray-tracing is currently a hot topic in the graphics industry with NVidia being a big proponent. Any hardware ray-tracer will have to perform tasks like this efficiently and the architecture and software will have to be designed carefully and tuned heavily.