UM EECS 487

Lab 7: Basic Ray Tracing

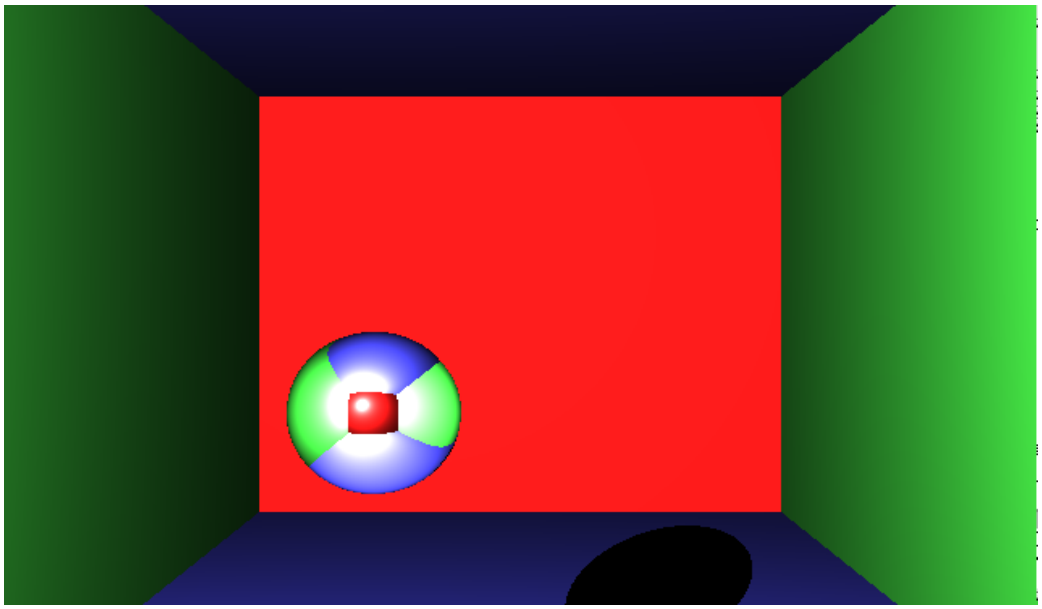

Fig 1: Screenshot showing the scene after implementation of (a) basic ray tracing, (b) lighting calculations, and (c) basic shadow effects, all of which can be achieved with one simple recursive function with simple tests to check for various conditions.

In this lab we will implement basic ray-tracing. As shown in Fig 1, a perfectly reflecting sphere remains suspended within a box with colored walls: 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. Note the following:

1. the sphere reflects the box around it and the reflection is modulated (deformed) by its position within the box,
2. the sphere casts a shadow on the wall 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.

Learning objectives from this lab:

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 the ray in a depth-first search (DFS) manner.

Support Code Documentation

Points are represented using the **`Pt`** class and 3D-vectors as the **`Vec`** class, both of which are declared in `point.h`. There are operators and methods for common operations such as length, normalization, distance calculation, addition, subtraction, and scalar multiplication. A ray is represented as a direction **v** and a point **p** through which it passes. Each **`Ray`** class has a parameter t which is used to generate points **r** along the ray using the formula

$$\mathbf{r} = \mathbf{p} + t\,\mathbf{v} \quad . \tag{1}$$

The **`Ray`** class is declared at the end of `point.h`.

The **`Sphere`** class (`sphere.h`) represents the properties of a sphere. Useful methods are:

1. `Vec Sphere::unit_normal(Pt const&)` 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 will need to implement this*.

The **`Plane`** class (`plane.h`) represents the walls. It consists of a point through which the plane passes, 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 will need to implement this*.

The **`Color`** class (`color.h`) is a wrapper for a triplet of doubles that represent RGB color just like in OpenGL. The **`Material`** class (`material.h`) is used in lighting calculations.

The **`Light`** class (`light.h`) represents (the only) positional, point, light source. It consists of the position, ambient, diffuse, and specular contributions (though only diffuse is used). The method useful to you is:

1. `Light::pt(Pt const& p, Vec const& n, Material const& mat)` which returns the `Color` due to illumination of the point **p** (on a surface) with normal **n** and material *mat*. This function performs a diffuse lighting calculation to determine the color of the walls.

Instructions

- Compile the code by running `make`. Run the executable named `raytrace`.
- First implement the functions that return the point of intersection or a ray with a plane and with the sphere. These should return the *t* parameter for the ray for the point of intersection (described later). The actual point can then be determined by calling `Ray::pt(t)`.
  - **Plane:** Modify `Plane::intersect()` in `plane.cpp`.
  - **Sphere:** Modify `Sphere::intersect()` in `sphere.cpp`.
  - 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 latter 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)`. See `point.h`.
    3. Keep track of the most desirable candidate for *t* from among the walls as well as the corresponding wall.
    4. Calculate the color at the intersection point from the best wall candidate by calling `Light::pt(Pt, plane-normal, plane-material)`. See `plane.h`.
    5. Basic shadow: If the ray from the light source to point on the wall intersects the sphere then the sphere casts a shadow. Otherwise the wall reflects light in the usual way. This should be an easy modification.
  - 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.

When you finish the tasks above, modify the keyboard callback so that you can interactively move the ball around the room. 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 Intel 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.