Assignment 6: Ray Tracing

Written Homework Due: Tuesday, Apr 7 (9:30 AM)
Programming Project Due: Wednesday, Apr 8 (5:00 PM)

1 Introduction

Throughout this semester you have written code that manipulated shapes and cameras to prepare a scene for rendering. In this (final) assignment, you will be writing a raytracer for your scenes.

2 Requirements

2.1 The Raytracing Pipeline

You learned in lecture the basic steps of the raytracing pipeline's inner loop:

Generating rays (simplest case: Just shoot a ray through the center of each pixel)

Finding the closest object along each ray

Illuminating samples

2.2 Implicit Equations

One of the real advantages of raytracing is that you don't have to work with approximations to the objects in your scenes. When your objects are defined by an implicit equation, you can render that object directly with a resolution as high as your image allows. You need to be able to render the following objects using their implicit equations: Cube, Cylinder, Sphere, and Cone.

2.3 Ray intersections and normals

Your first task is to intersect a ray with objects in the scene. You can do so by writing an intersection code for a shape that allows for any position, orientation, and scale, but this is likely to be quite complicated. To re-use your code in the previous lab, you can instead transform the ray into the standard object's space (i.e., where the object has unit size centered at the origin) and then perform

intersection. This is done simply by multiplying the ray equation with the inverse of the object transformation.

After intersection, you will need to transform both the intersection point and its normal back to the world space. To transform the intersection point from object space into world space, simply use the object transformation matrix. **But**, to transform the normal vector you computed in the object space to the world-space, you need to do something different. In particular, let the object transformation matrix be M, the intersection point be p_{object} in the object space, and the normal vector be n_{object} in the object space. The intersection and normal vector in the world space are obtained by

$$\begin{aligned} p_{world} = &\quad M \cdot p_{object} \\ n_{world} = &\quad (M^{-1})^T \cdot n_{object} \end{aligned}$$

where n_{world} needs to be re-normalized. That is, the normal vector is transformed by the inverse-transpose of the object transformation. This is necessary for the normal vector to be again orthogonal to the tangent plane of the shape at the intersection point in world-space. To see why this works, take a vector v_{object} that lies on the tangent plane of the object at p_{object} in the object-space, which is transformed into the world space by $v_{world} = M \cdot v_{object}$. The normal vector n_{world} in the world space is orthogonal to v_{world}:

$$(n_{world})^T \cdot v_{world} = ((n_{object})^T \cdot (M^{-1})) \cdot (M \cdot v_{object}) = (n_{object})^T \cdot v_{object} = 0$$

2.4 The Illumination Model

Your illumination model will be the basic Phong model, with additional terms to handle attenuation, shadowing, and reflection. You'll also have the opportunity to add special features such as texture mapping, transparency, and soft shadows.

With the basic Phong model your scene will look a lot like the output from Sceneview. As you add in shadows and reflection, your scenes will become more interesting.

You must be able to handle point light sources.

For this assignment you are required to write a program that, given a name of a scene file, will generate a high-quality image that correctly takes into account lighting, all surface characteristics, as well as shadows. You have already implemented the ray tracing portion of the assignment. Your primary job in this assignment is to correctly handle various surface characteristics:

- Phong lighting
- Shadows
- Attenuation
- Reflection

- (Optional) Texture mapping — must be able to map a texture onto the following shapes:
  - Cube
  - Cylinder
  - Cone
  - Sphere
- (Optional) Advanced Lighting (e.g. transparency, refraction)

To calculate the intensity of the reflection value, you need to determine the reflection vector based on object's normal and the look vector. You then need to recursively calculate the intensity of the intersection point that the reflection vector hits. With each successive recursive iteration, the amount of the reflection contribution to the overall intensity drops off. For this and for practical reasons, you need to set a limit to the amount of recursion your reflection is calculated with. Make sure that this value (level of recursion) is a variable or a constant instead of a hard coded value (i.e., check your current level of recursion against a variable like RECURSIVE_LIMIT instead of a hard coded number like 5). I may want to change it when grading your work. If you do not implement your program this way, you will get some deduction.

Texture mapping: To support texture mapping you can use the `Fl_BMP_Image` image class in FlTk to read bmp files. You should correctly deal with missing files — preferably by popping up a dialog asking where the file is.

To derive the final light intensity value for each of your pixels, you need to make sure that you can parse, store and use the following parameters for each of your objects:

```
trans [
      .
      object object name [
            ambient <color>
            diffuse <color>
            specular <color>
            reflect <color>
            shine float
            texture filename u v
            ]
      ]
```

Here, shine is the specular exponent, and reflect is the coefficient specifically for weighting the sum of illumination along the reflected ray (coefficient k_r in the lecture).

Some notes about texture mapping parameters: Filename should be a text file name of the texture you want to use for that shape. U and V values are how many times you should repeat the texture for that shape in X and Y texture coordinates. For example, if the 2 and 1 are used for U and V respectively, then, the pattern should repeat itself twice in X coordinates, but only once in Y coordinate. You can chose to ignore these parameters, but you get more bonus points if they work.

3 Support Code

The support code is similar to the previous assignments. You should be familiar with how to set up your scene to be loaded. The code will call `MyScene::render` with the rendering type (which you can safely ignore unless you are going to implement several rendering methods), the width and height of the target image, and an array of bytes that will hold the image. Recall from the first brush assignment how pixels can be stored as a big array of bytes representing the red, green and blue coefficients of each pixel in row major order. Feel free to refer to the support code from the brush lab if you need help with how to access and plot pixels into this bitmap.

A first step is to simply plot some simple pattern into the array of pixels and return, so you can become familiar with how the code works. Next, implement ray-object intersection and simply plot the diffuse color of the objects you find. Finally, write a shade function which evaluates the lighting equation at the intersection point for each pixel.

There is also a `stopRendering` method which gets called when the user clicks that button. You should gracefully bail out of your rendering code at the next scanline after `stopRendering` is called. (FlTk code is threaded.)

4 Extra credit

Extra credit: Support one additional kind of light, either a directional light or a spotlight. If you want to implement the spotlight, the scenefile would look something like the following. Spotlights have a point, direction, and an aperture in degrees, which dictates the steradian angle being subtended by the light cone.

```
light [
      id <number>
      type spotlight
      color <color>
      function <color>
      direction <vector>
      point <point>
      aperture <angle>
      ]
```

Texture mapping: Requires calculating texture map coordinates for the intersected object point.

Transparency/refraction: Refraction earns you more points than simple transparency, since you have to calculate the new refraction ray.

Speed optimization: Speeding up raytracing is difficult. Usually, it is not enough to optimize inner loops, as there is just too much work that needs to be done for ray creation, intersection testing and illumination. You need to cut away large amounts of this work. You can try implementing some of the culling tricks, such as bsp-trees, octrees or grids.

Others: See me if you have/need any ideas. Remember, if you implement something interesting, make sure you create sample scenefiles to demonstrate the capabilities clearly. Document all of this in your readme.

5 FAQ

1. **My ray-tracer seems to be running really slow. What do I do?**

   You are probably going to want small windows for testing your ray tracer. (Time spent is proportional to size of window.)

2. **Where do I start?** Start with a camera looking down the z axis and a single object (say a sphere) at the origin, no transforms. In this case the camera to world matrix is close to the identity (minus a scale) and the object's matrix is the identity. The intersection of the ray with the sphere should be on the positive z side, and the length of the vector from the intersection point to the origin is 0.5.

   Now add some simple lighting.

   Once this is working, apply a matrix transformation and see if it still works (make sure the sphere is visible by checking with your sceneview code).

   Once you've got the scenes working with the basic Phong lighting, then add in shadows.

   Finally, add in reflection.