RAY TRACING

Author: Jin Wu

December 1, 2008

INTRODUCTION

This report complements the ray tracing program written in C++ with the help of OpenGL library. (**Figure 1**).

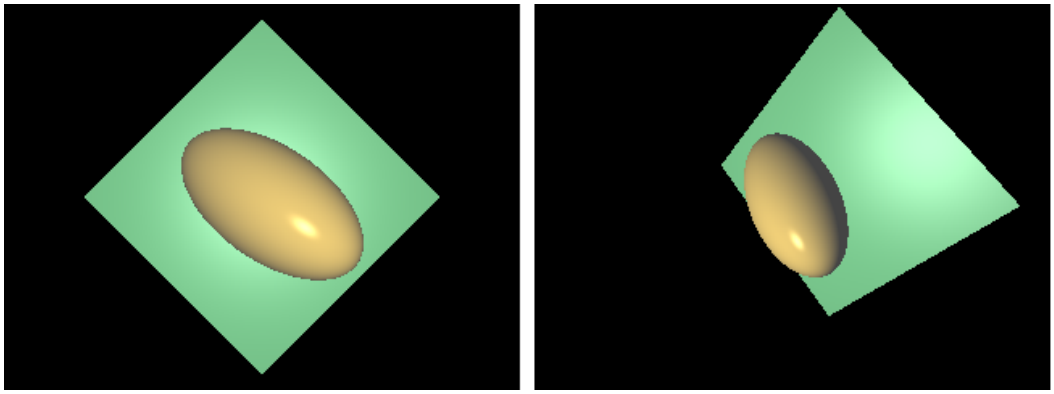


Figure 1: Ray traced scene of an ellipsoid and a plane

This report consists of 7 parts:

A. Secondary Reflection
B. Shadow Casting
C. Quadratic Object 1: Cylinder
D. Quadratic Object 2: Cone
E. Refraction
F. Extra Object: Checker King
G. Instructions on Running the Program

A. SECONDARY REFLECTION

This was mainly implemented in `raytracing.cpp` under the `shadeRay` function as follows:

I. Defining the Reflective Ray

The **origin** of `reflectionRay` is found at the point of intersection on the surface of an object. However, to prevent effects of round off errors, 0.0001 was added to each of the X, Y, Z coordinates of the `reflectionRay` origin to produce better looking results.

The **direction** of `reflectionRay` (r) as seen in **Figure 2** below can be defined as follows:

r = normalized (-s + 2t)

Where s = normalized (light - ray.intersection.point)

t = normalized (n·s)*n

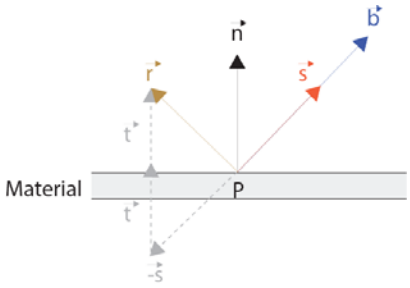


Figure 2: Reflection Diagram

II. Casting the Ray

Before the `reflectionRay` is casted off, the intersection point is set to a large value (as close to "infinity" as possible) in order to better predict the first hit object. Then, shadeRay is called recursively until the `maxdepth` value is reached. The colour of the object hit by `reflectiveRay` is then returned, scaled, and added to the colour of the original surface.

The scaling parameter `reflection_coeff` is added to the Material class of the `util.h` file and called upon while defining the materials in the `main` function of `raytracer.cpp`.

B. SHADOW CASTING

This function was implemented in `raytracer.cpp under the functions traverseShadow and computeShading`.

a. defining and casting the ray
   The ray was casted from the point of intersection back to the light source.

b. Predicting intersection
   The `traverseShadow` function was used for determining whether `shadowRay` hit anything. If it did, a shadow was casted at the original point on the surface.

   The `traverseShadow` function is similar to the `traverseScene` function, except the intersection is performed reversely.

c. Shading the shadow
   The shadow was shaded by setting the colour to 50% of the original instead of turning off everything but the ambient term. This produces better results.

C. RAY-CYLINDER INTERSECTION

This function was implemented in `scene_object.cpp` and `raytracer.cpp`.

I. scene_object.cpp

a. The ray was transformed into object space by multiplying the origin and direction by the worldToModel matrix.

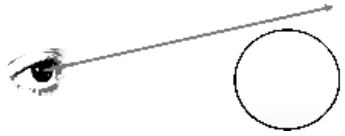
b. Implementing the intersection of cylinder centred at (0,0,0) with z ranging from -0.5 to 0.5:

$$\lambda^* = \frac{-2B \pm \sqrt{4B^2 - 4AC}}{2A} = -\frac{B}{A} \pm \frac{\sqrt{D}}{A}$$

where A = ray.dir[x,y] · ray.dir[x,y]
B = 2(ray.origin[x,y] · ray.dir[x,y]
C = ray.origin [x,y]· ray.origin [x,y]
$D = B^2 - AC$

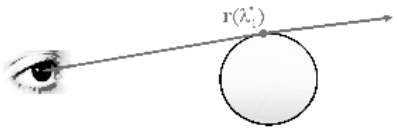
From this equation, the following scenarios can result:

Case I. D < 0: no intersection



flag = 0

Case II. D = 0: 1 hit



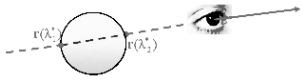
t_value = -B/A
point = modelToWorld * (origin+t_value*dir)
normal = normalized transNorm(worldToModel, ray.intersection.point-Point3D(0,0,Z))
none = false
flag = 1

Case III. D > 0: 2 hits

$$\lambda^*_1 = -\frac{B}{2A} + \frac{\sqrt{D}}{2A}$$

$$\lambda^*_2 = -\frac{B}{2A} - \frac{\sqrt{D}}{2A}$$

III(a): $\lambda^*_1 < 0$ and $\lambda^*_2 < 0$: not visible

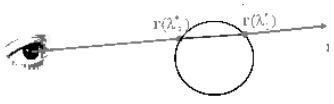


return false

III(b): $\lambda^*_1 > 0$ and $\lambda^*_2 < 0$: inside sphere, first hit valid

t_value = λ^*_1
point = modelToWorld * (origin+t_value*dir)
normal = normalized transNorm(worldToModel, ray.intersection.point-Point3D(0,0,Z))
none = false
flag = 1

III(c): $\lambda^*_1 > \lambda^*_2 > 0$: in front, 2nd closer



t_value = λ^*_2
point = modelToWorld * (origin+t_value*dir)
normal = normalized transNorm(worldToModel, ray.intersection.point-Point3D(0,0,Z))
none = false
flag = 1

* Images above were taken from Leonid Sigal's lecture powerpoint slides at the University of Toronto.

c. Implementing the intersection of top and bottom caps at z=-0.5 and 0.5:
This was implemented similarly to a plane, but with a check of X^2+Y^2 = R^2 for boundary points instead of the four sides of a square.

II. First Hit

A complication arises when there is more than one object being rendered as multiple λ* values will arise when a ray intersects multiple objects.

The implementation provided in this program to solve this problem was done in the render function.

After a ray has been created, The t_value of the ray was set to a very large number (i.e. 10000000). It then tries to call the shadeRay function, which in turn calls the traverseScene function, which tries to detect the intersecting points.

In the scene_object.cpp file, detection is done so that an intersection occurs ⇔ the new t_value is smaller than the present t_value. If it is greater than the present t_value, then although an intersection occurs, it's not a *first hit*.

D. RAY-CONE INTERSECTION

This was implemented similarly to the Ray-Cylinder intersection described in section C. The only differences are:

1. One bottom cap was used instead of two
2. The equations for A, B, and C were different as follows:

```
double A = coneRay.dir[0]*coneRay.dir[0] +
                        coneRay.dir[1]*coneRay.dir[1] -
                        coneRay.dir[2]*coneRay.dir[2] ;
double B = 2*coneRay.origin[0]*coneRay.dir[0] +
                        2*coneRay.origin[1]*coneRay.dir[1] +
                        - 2*coneRay.origin[2] *coneRay.dir[2] ;
double C = coneRay.origin[0]*coneRay.origin[0] +
                        coneRay.origin[1]*coneRay.origin[1] -
                    coneRay.origin[2]*coneRay.origin[2] ;
```

E. REFRACTION

Refraction was implemented similarly to reflection. The only difference is in the ray (D) that is casted. It is defined as follows:

$$\vec{d} = \frac{c_2}{c_1}\vec{b} + \left(\frac{c_2}{c_1}(\vec{n}\cdot\vec{b}) - \left(1 - \left[\frac{c_2}{c_1}\right]^2 \left(1 - (\vec{n}\cdot\vec{b})^2\right)\right)^{1/2}\right)\vec{n}$$

Where C1 and C2 are the index of refraction defined in the `material` class of `util.h` under the variable name `refraction_coeff` and set arbitrarily in the `main` function of `raytracer.cpp`. B is the incoming ray vector, and n is the normal.

F. ADDITIONAL OBJECT

This was originally part of the King chess piece that was to be implemented. Due to time restrictions, only the top part (the cross) was implemented. The piece consists of 17 rectangular planes and checking for the closest intersection of each.

F. INSTRUCTIONS FOR RUNNING PROGRAM

To run the program in:

I . Regular Phong model with Depth set to 2, type in:
```
raytracer
```

II . Diffuse and ambient components of Phong model only, type in:
```
raytracer -diffuse
```

III. Scene Signature, type in:
```
raytracer –sig
```

IV. Diffuse with Shadows, type in:
```
raytracer –diffshade
```

V. Depth Set to 1, type in:
```
raytracer –depth1
```

VI. Final Image (produces one image), type in:
```
raytracer –scene
```

CONCLUSION

In conclusion, the above explanations are sufficient add-ons to the given code for simple ray tracing to work.