Ritesh Sharma

3/8/2015

Project #4

CS 551

OREGON STATE UNIVERSITY

This project gives us the better understanding of Ray Tracing. This was perhaps the most difficult project in this class. It took me more than 4 days to complete this project and 1 day to decide and write report displaying the scene which can capture all my work. Class Lecture and the theoretical concepts given in the book by peter Shirley helped me a lot to understand the concept behind ray tracing. I followed the approach given in the book. Since for this project, sample framework was not given, I had difficulty in thinking what I should include from the previous project and what I should change. Understanding the concept and coming up with an algorithm took me one day. I didn't wanted to waste my time debugging code after I implement a faulty algorithm, so I gave enough time to come up with an algorithm and verify it without implementing it. It took me 3-4 hours to correctly write an algorithm. As I had the code for drawing triangles and shading model. It saved me some time as I reuse them in my code. Data structure implementation was straight forward as I had code from previous projects. Coming up with the right scene to show all the feature of the implementation took me around 2-3 hours.

Ray Tracing

In computer graphics, ray tracing is a technique for generating an image by tracing the path of light through pixels in an image plane and simulating the effects of its encounters with virtual objects. The technique is capable of producing a very high degree of visual realism, usually higher than that of typical scan line rendering methods, but at a greater computational cost [1].

Overview

Ray tracing is so named because it tries to simulate the path that light rays take as they bounce around within the world - they are traced through the scene. The objective is to determine the color of each light ray that strikes the view window before reaching the eye. A light ray can best be thought of as a single photon. The name "ray tracing" is a bit misleading because the natural assumption would be that rays are traced starting at their point of origin, the light source, and towards their destination, the eye (see Fig. 1). This would be an accurate way to do it, but unfortunately it tends to be very difficult due to the numbers involved. Consider tracing one ray in this manner through a scene with one light and one object, such as a table. We begin at the light bulb, but first need to decide how many rays to shoot out from the bulb. Then for each ray we have to decide in what direction it is going [2].

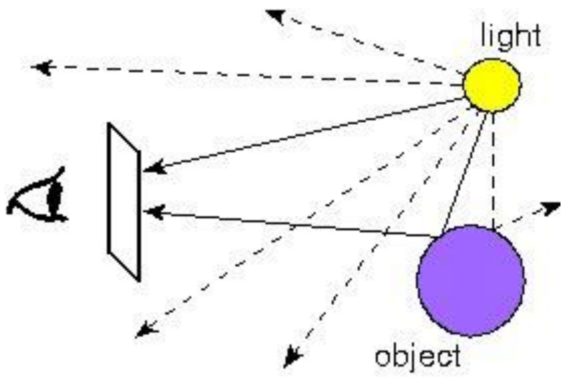


Figure 1. Tracing rays from the light source to the eye. Lots of rays are wasted because they never reach the eye.

In order to save ourselves this wasted effort, we trace only those rays that are guaranteed to hit the view window and reach the eye. It seems at first that it is impossible to know beforehand which rays reach the eye. After all, any given ray can bounce around the room many times before reaching the eye. However, if we look at the problem backwards, we see that it has a very simple solution. Instead of tracing the rays starting at the light source, we trace them backwards, starting at the eye. Consider any point on the view window whose color we're trying to determine. Its color is given by the color of the light ray that passes through that point on the view window and reaches the eye. We can just as well follow the ray backwards by starting at the eye and passing through the point on its way out into the scene. The two rays will be identical, except for their direction: if the original ray came directly from the light source, then the backwards ray will go directlyto the light source; if the original bounced off a table first, the backwards ray will also bounce off the table. You can see this by looking at Figure 1 again and just reversing the directions of the arrows. So the backwards method does the same thing as the original method, except it doesn't waste any effort on rays that never reach the eye.

Brief Algorithm

Step 1: Compute the basis vector where eye is located

Step 2: For each pixel, compute the viewing ray by finding which object lies first and which lies next by calculating the surface normal

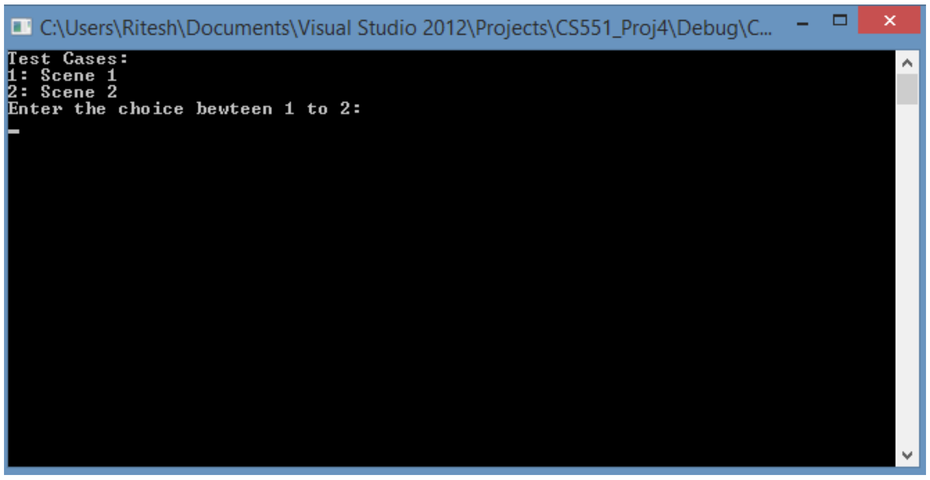
Step 3: Color the pixel based on lighting model and the surface normal calculated in step 2

The algorithm for ray intersection with sphere and triangle was implemented based on the book by Peter Shirley. The properties of the radius of the sphere and it's position gives us a better way of finding the intersection and thus helping in solving the linear equation that results.

The formula that determines the intersection point is:
$(\mathbf{e} + t\mathbf{d} - \mathbf{c}) \cdot (\mathbf{e} + t\mathbf{d} - \mathbf{c}) - R^2 = 0$

Where **e** represents the ray's origin, **d**represents the ray's direction,**c** represents the centerpoint of the sphere, R represents the radius of the sphere, t represents the distance the ray must travel to intersect with the sphere. Solving this equation for t, using the quadratic equation, will result in the intersection of the ray. If the determinant of the quadratic equation is negative, then the ray will never touch the sphere. If the determinant is 0, then the ray touches the edge of the sphere. If the determinant is positive, the ray passes through the sphere and the smallest positive value of t represents the nearest point of intersection. Multiplying t by **d** and adding the result to **e** results in the actual point of intersection **p**, and subtracting **c** from **p** gives the normal of the surface **n**.

In order to have flexibility to check two scenes user can run the program and select the test cases. I have provided to test cases and the GUI appears as given in figure below:



```
Test Cases:
1: Scene 1
2: Scene 2
Enter the choice bewteen 1 to 2:
_
```

C:\Users\Ritesh\Documents\Visual Studio 2012\Projects\CS551_Proj4\Debug\C...

Some of the results obtained from the implementation of this project is given below:

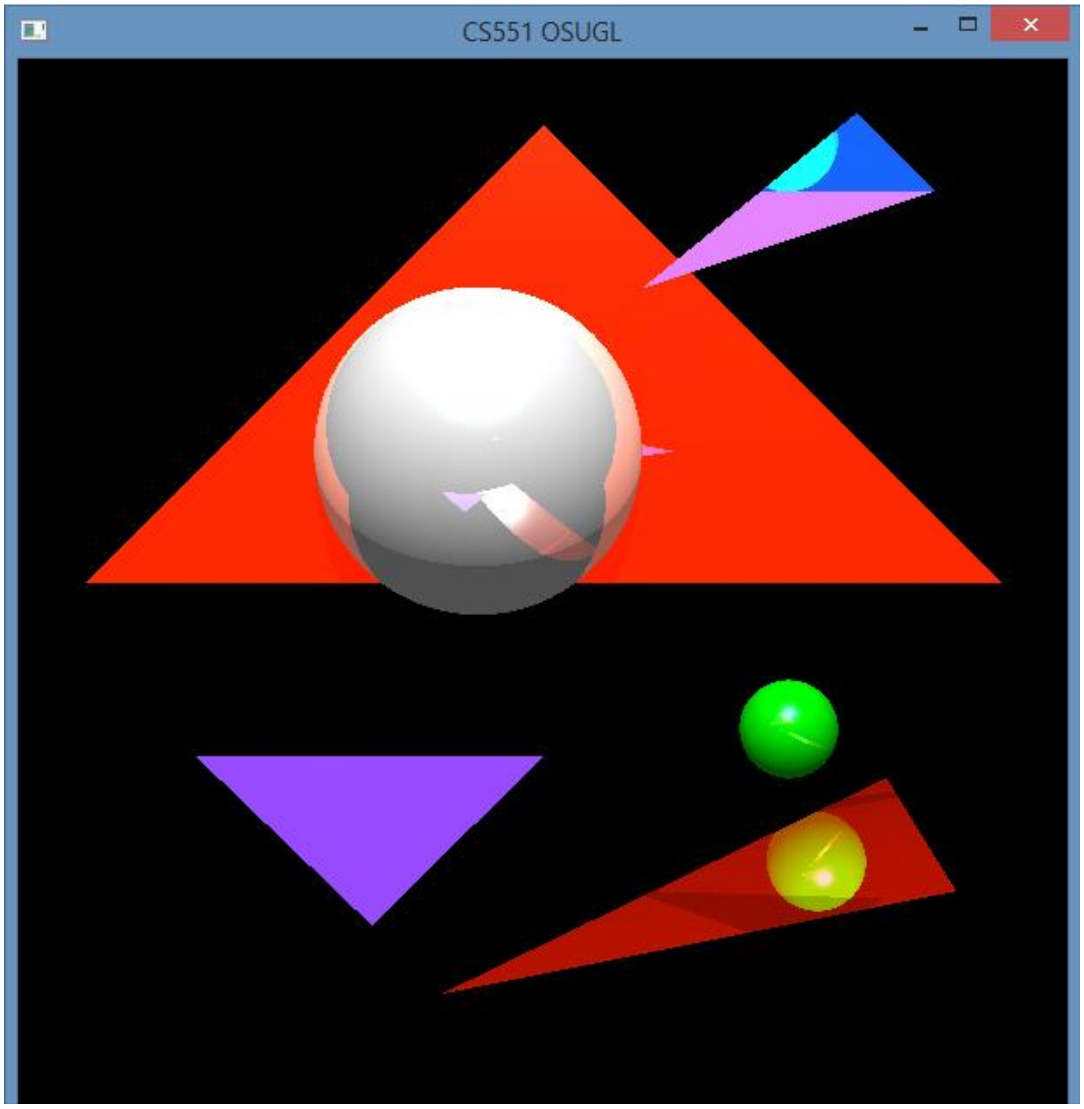


Figure 2. Results based on Scene 1

The above figure is the best example to see the lighting and the reflection. It can be seen that the smaller sphere reflects over a triangle at the bottom where larger sphere reflects the triangles. Shadow can be seen at bottom part of the larger sphere. Another test case is shown below in figure 3.

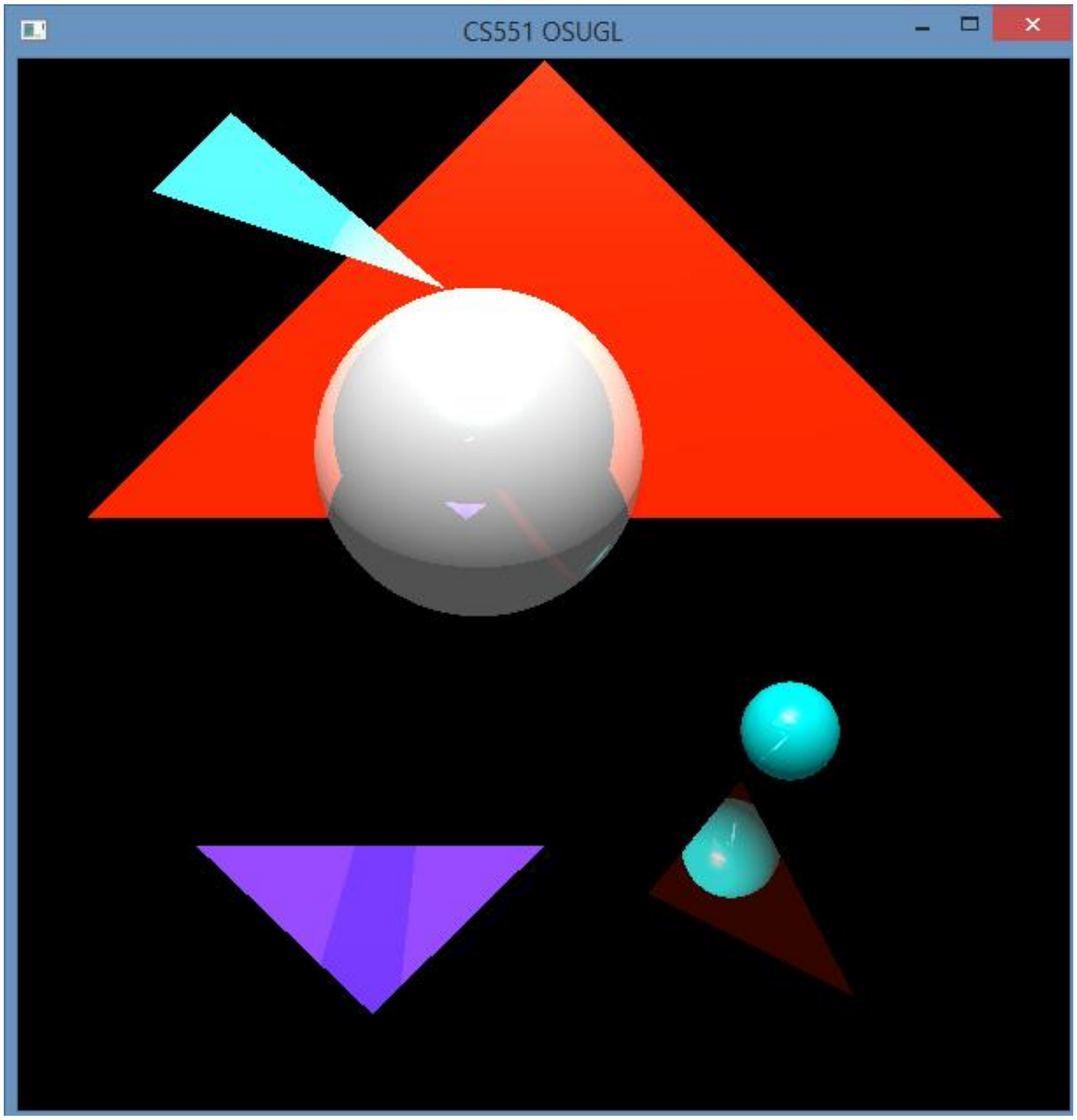


Figure 3. Results based on Scene 2

From a data-structure perspective, everything is essentially as close to optimal as possible. From an algorithm perspective, I don’t think there is any way to improve. There can be artifacts in the image that was produced but due to reflection and lighting, I am unable to verify it. Given ample time, some more test cases can be generated to verify the results.

References:

1. ***http://en.wikipedia.org/wiki/Ray_tracing_graphics***
2. ***https://www.cs.unc.edu/~rademach/xroads-RT/RTarticle.html***
3. ***Fundamental of Computer Graphics by Peter Shirley***