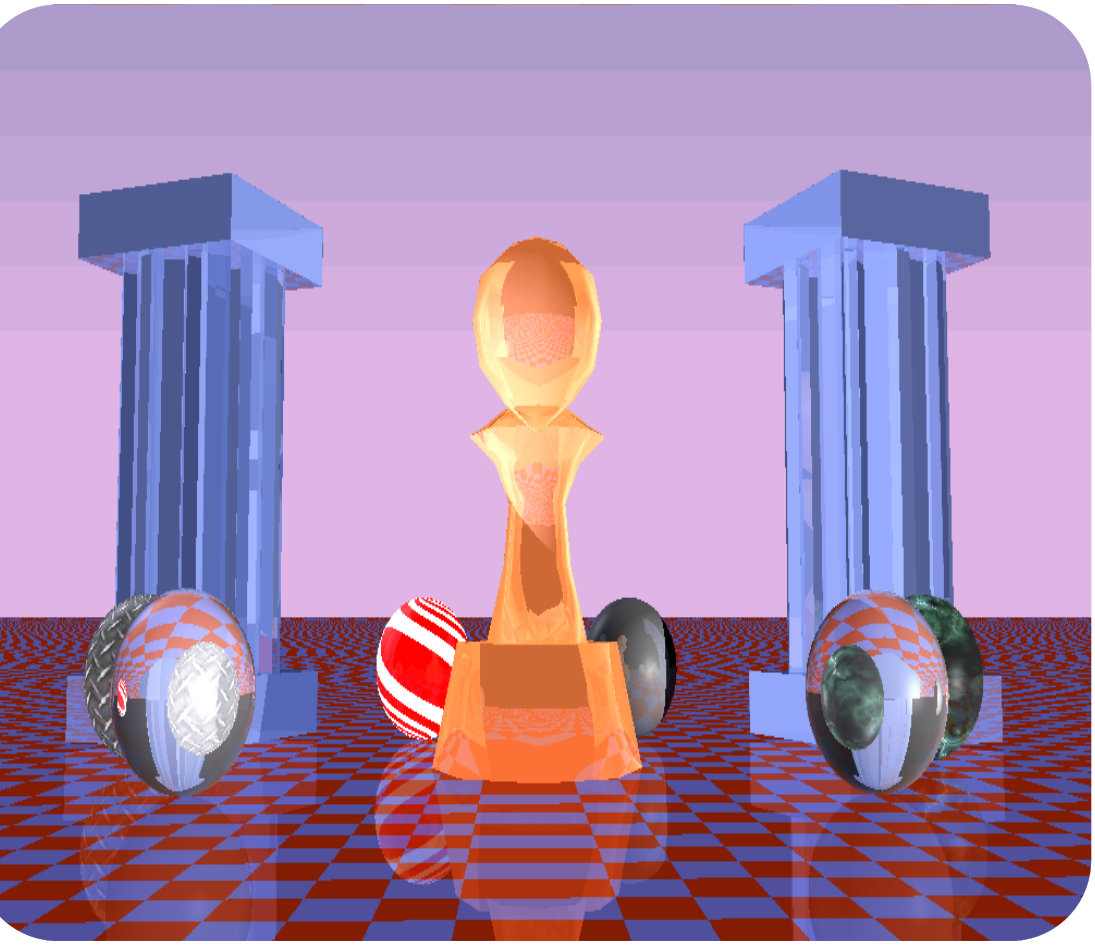


Infinity Ray Tracer

by

Uthara Thelagar
Anil Ramakrishna
Tanmay Patil
Himanshu Joshi
Srikanth Madhava

Table of Contents

1. INTRODUCTION

In this section we review our motivations for choosing Ray Tracing and our project objectives.

1.1 *MOTIVATION*

The CS580 course has been about growth – both as software engineers and graphics programmers, with the distinction between the two constantly blurring with each project. In a few weeks we were able build a renderer that took input coordinates in the model space and rendered them in screen space with different effects such as texturing, procedural texturing, anti-aliasing etc. Inspired by the Professor's enthusiasm and experience we were very motivated to explore several topics for the project.

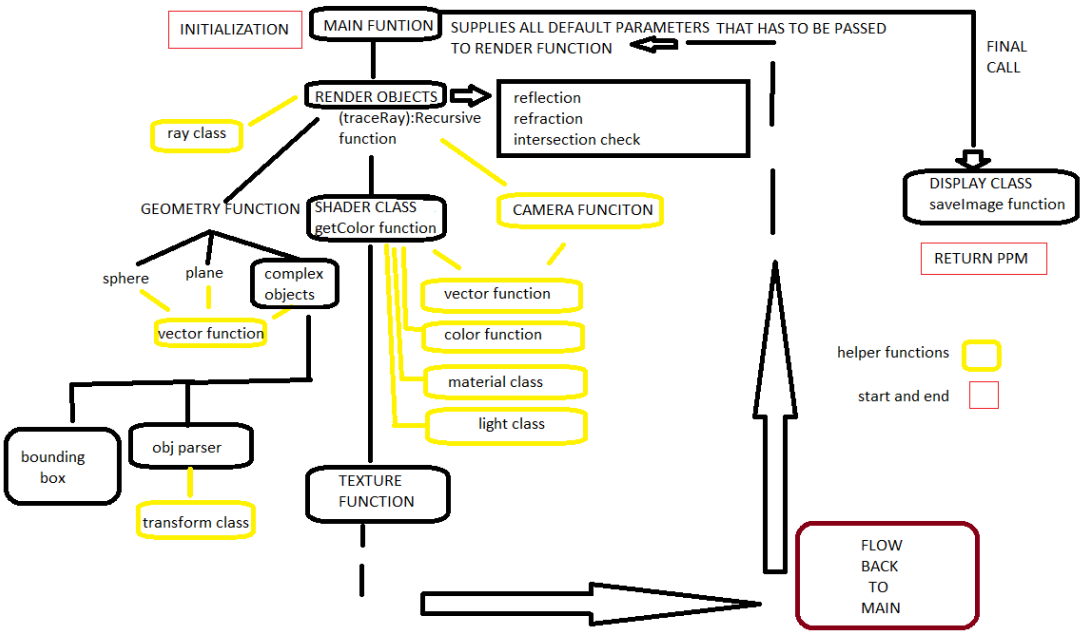
We considered Interactive animation, Cloth simulation, Procedural Animation and several other ideas for the project. Finally, we decided to go with building a ray tracer since we were inspired by its simple yet powerful concept. We wanted to understand how to construct this renderer from scratch with the possibility to explore any apparent bottlenecks. We were also keen on adding another fundamental algorithm to our arsenal of 3d computer rendering techniques.

1.2 *OBJECTIVES*

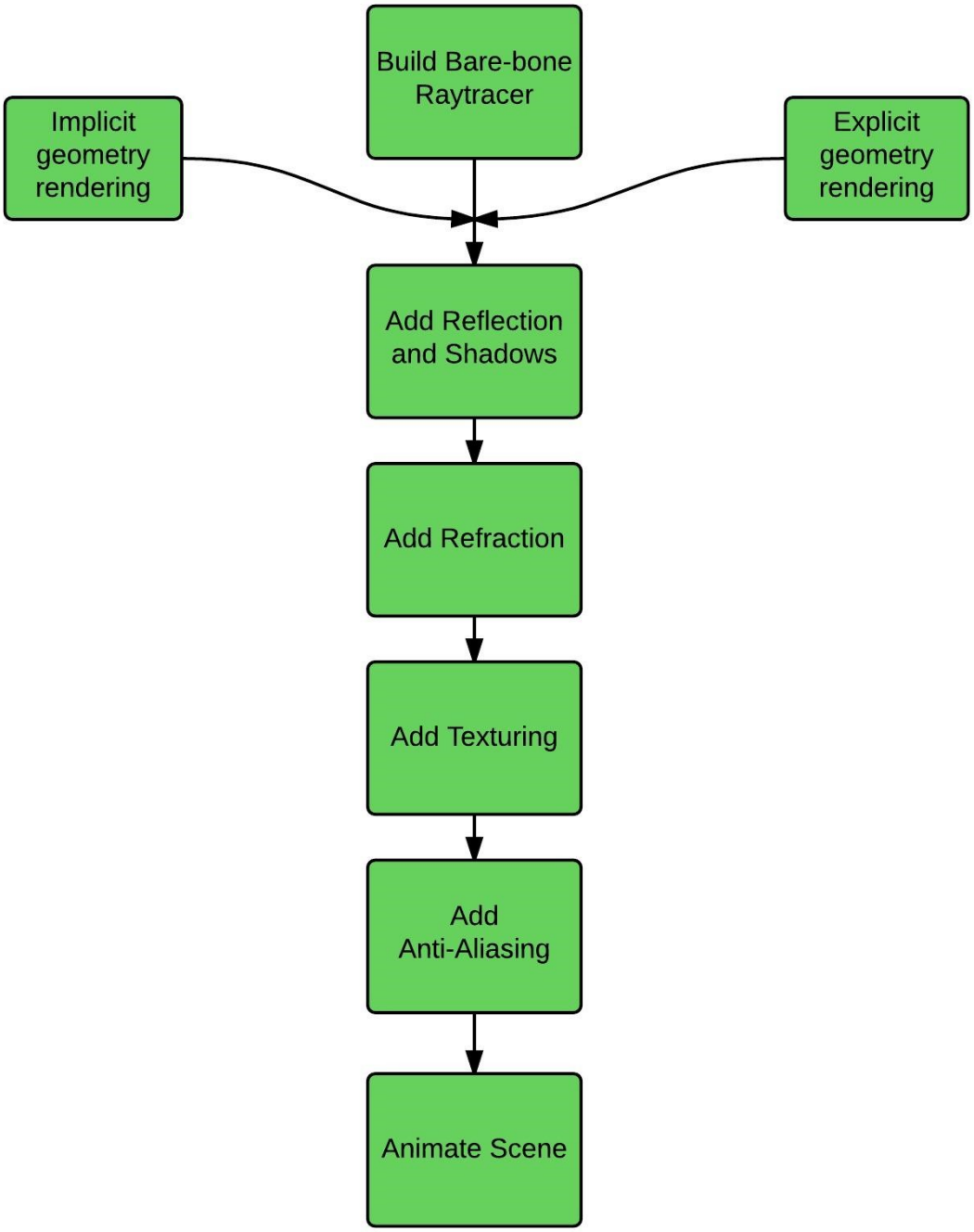
- Develop a ray tracing library from scratch
    - Understand the computational aspects of Ray Tracing
        - Is there room for optimization?
- Add Shading, Reflection, Refraction, Anti-Aliasing and Texturing.
- Create a brief animation of our scene.
- Work in a team, with a distributed asynchronous workflow.
- Support multiple platforms for development to accommodate different members' needs.

2 ORGANIZATION

2.1 *SOFTWARE ARCHITECTURE*



2.2 *DEVELOPMENT CYCLE*



2.3 *TASK BREAKDOWN*

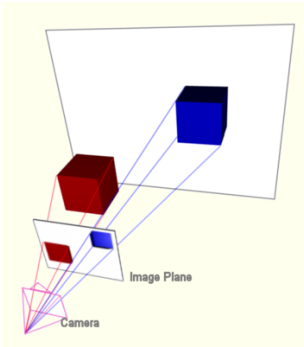
This project is unique both in its scope and its ambitions. Some of us were interested in understanding the graphics pipeline while others in working with the computational challenges that it entailed. As requested, we are providing a tentative breakdown of tasks below.

Name	Tasks
Uthara	Main concept, design, implementation, skeleton tracer, shadows, shading, reflection, look development. Domain architect.
Himanshu	SW Architecture, Skeleton Tracer, Shader, Multiple Object Intersection, Github, Cross Platform Support, Documentation.
Tanmay	Explicitly Modeled Object Parser, OctTree Optimization for complex object rendering, s-t and Normal Interpolation.
Anil	Animation, Refraction, Reflection, Shadow calculation, Debugging, Code Refinement, Documentation, Report generation.
Srikanth	Texture Mapping (Explicit and Procedural), Anti-Aliasing, Website, Animation, Report generation.

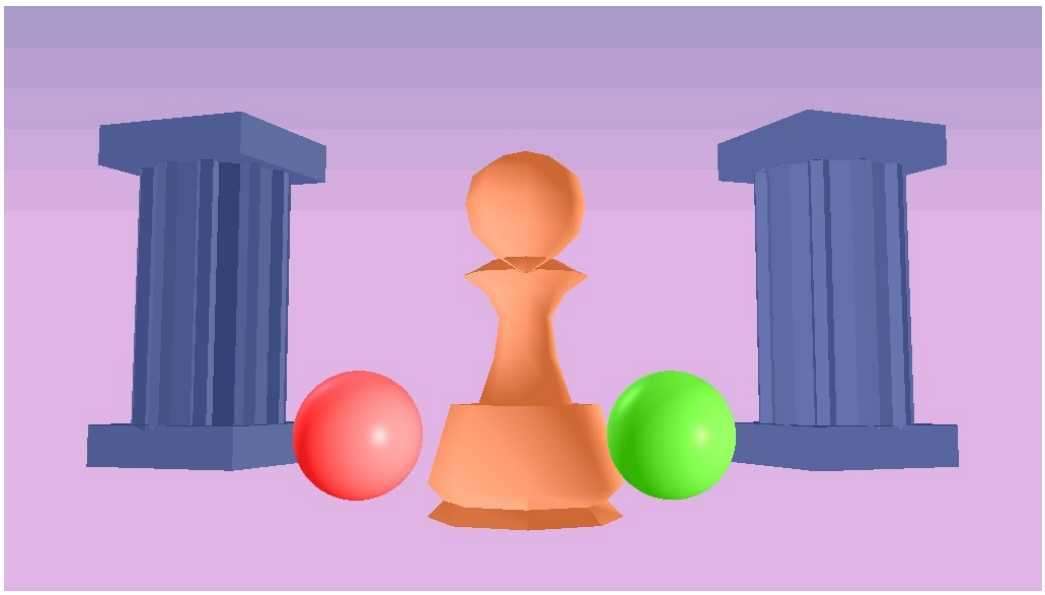
3 IMPLEMENTATION DETAILS

3.1 *BAREBONE TRACER*

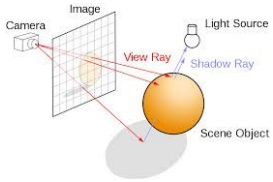
We built a basic ray tracing renderer by following the tutorials from (1). This renderer could handle basic implicitly modeled objects such as a sphere, plane, etc. The points of ray-intersection with these objects were calculated mathematically using standard theorems in 3d geometry and vector algebra (2). We used Phong shading (3) for our color calculations.



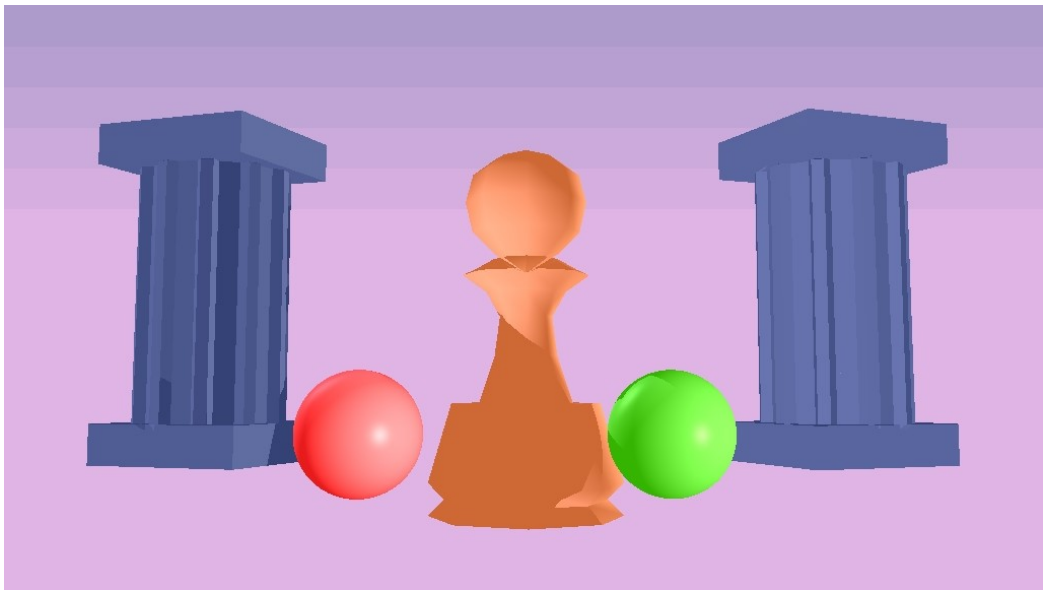
A sample image generated at this is shown below. (Pillars and pawn implemented later using object parser)



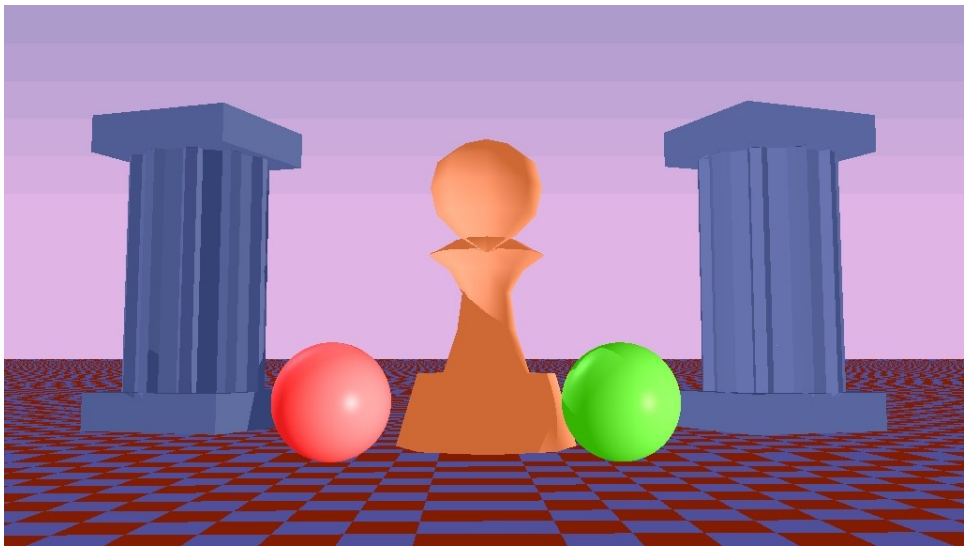
Our shadow calculations also follow directly from (1),



the resulting image with shadows is shown below.



We also included a checkerboard plane as shown below.



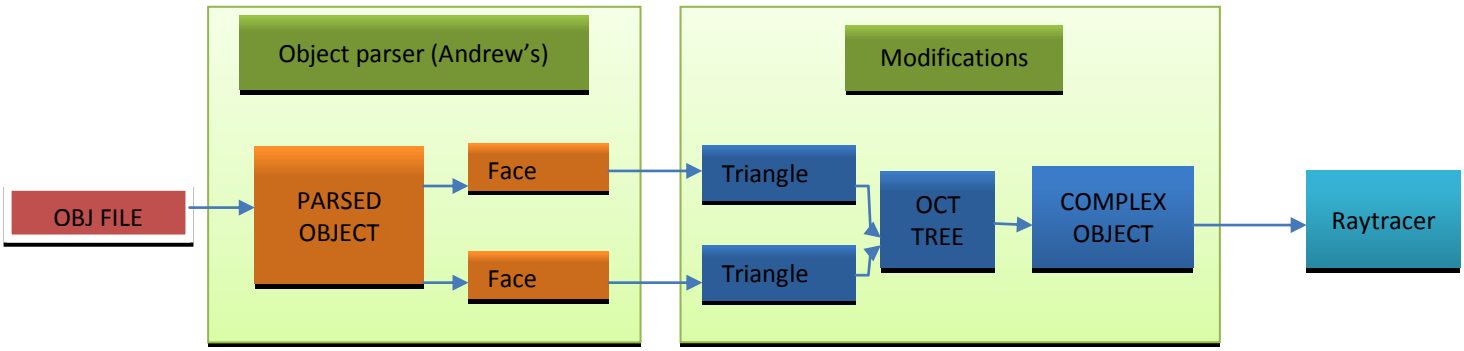
3.2 *IMPLICIT OBJECTS*

We wanted to make use of the bare bone Raytracer immediately, for reflections, refractions etc., so we started with Sphere and Planes created using implicit equations since they were straight forward and very quick to render. We borrowed equations from Andrew Glassner's Intro to Ray-tracing.
We used the basic Sphere and plane equations to find intersection points by substituting values we received from the ray origin and direction, solving the equations and finding the intersection value, intersection point and normal, for points that's visible in front of the image plane, which was later used for all shading calculations.
Basic Plane equation: Ax+By+Cz+D = 0
Basic Sphere equation: (Xs-Xc)^2+(Ys-Yc)^2+(Zs-Zc)^2 = radius^2

3.3 *OBJECT PARSING*

Our next step was to add an object parsing library that could read any obj file containing model space vertices of polygons that make up the complex 3d object. We used the object parser developed by Andrew Goodney (4) and slightly extended it to transform all the input polygons to camera space, using the transformation matrices we learnt from the class home works.

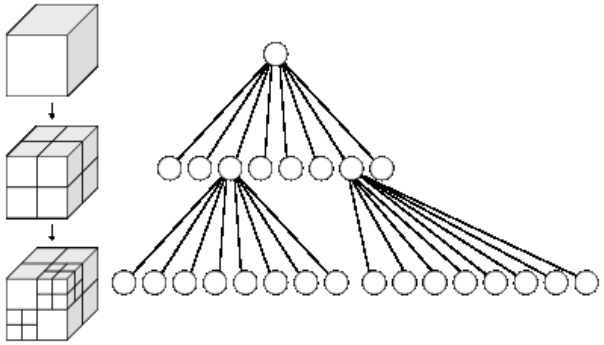
Overview diagram



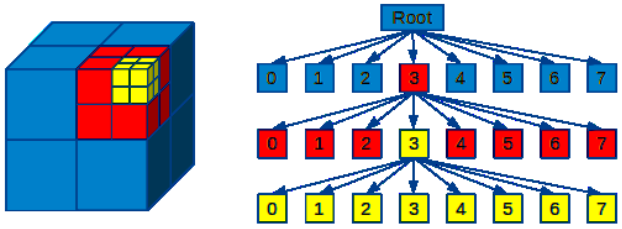
The rendering time with the basic parser however was very limiting; it took more than an hour to render the Utah teapot on our laptops. To avoid this, we added a very efficient optimization algorithm (5) (6) which creates a bounding box for each complex object and also recursively creates eight smaller boxes to occupy the eight 3d quadrants within the bounding box. In the innermost level of recursion, we index all the triangles that lie within each small bounding box. This is similar to creating a large number of positional hash for each triangle in the object; instead of testing for ray intersection with each triangles in the polygon, we only test the triangles that belong to the innermost bounding box that contains the ray. Using this optimization, the rendering time reduced from 1+ hour to ~ 5 minutes.

Triangle Intersection using LEE

d = - N . v0, t = - ((N . Ray_origin) + d) / N . Ray_dir
P = Ray_origin + (t . Ray_dir)
where N=> normal, P => intersction point, t=> intersection distance

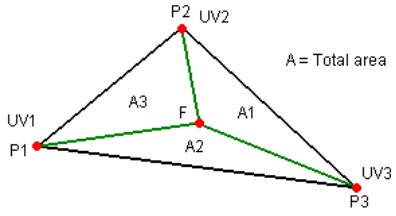


CONSTRUCTION OF OCT TREE



TRAVERSAL OF OCT TREE

UV Interpolation and texture interpolation: using Barycentric coefficients



using the ratio of the total Area to A1,A2,A3 we calculate bary centric coefficients a,b,c which we then use to interpolate uv's and normal

3.4 *SHADING*

This has two components:

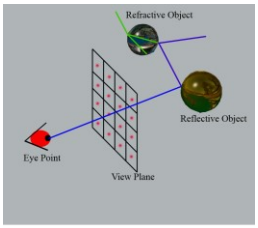
Diffuse and shadow: Iterating over all objects, if N.L > 0, then light hits the object. For shadow, send a new ray with the current object intersection point as the origin and light direction as the direction of intersection of object and the light. Send this ray to every other object to get the shadow indicator. If shadow indicator is set to true, then we do not calculate color.

- Diffuse color is: $ColorDiffuse = Ka * N.L * LightColor$
  $ColorDiffuse = Kd * N.L * LightColor$
- Specular: This component accounts the reflection of the ray.
  $R_d = 2(N.L) * N - \overline{L}$
  We check if: $R_d.L > 0$ then,
  $ColorSpecular = Ks * LightColor * (R.L^{spec_{coeff}})$

3.5 *REFLECTION*

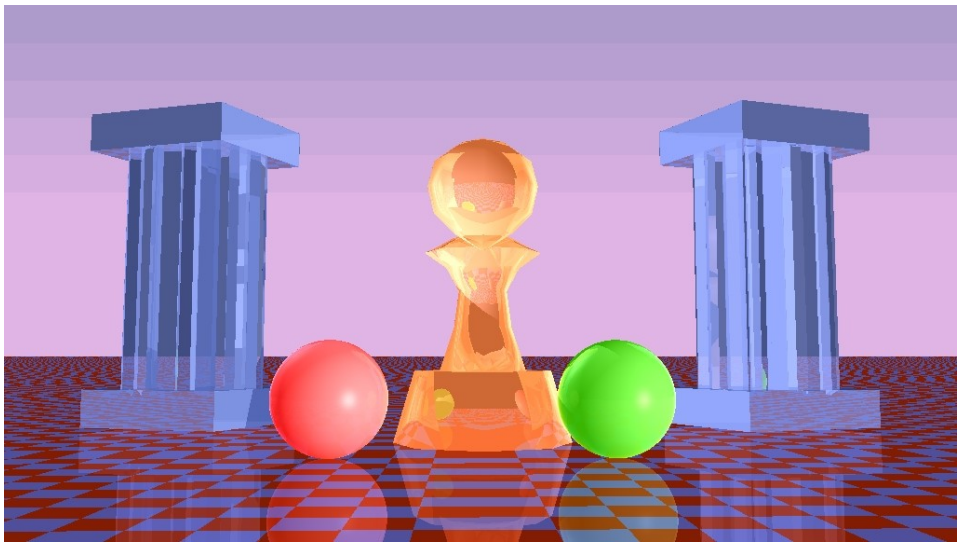
We added reflection using the following equation from (7) and (8).

$$R = I - 2\,(I.N)N$$



Here, R is the reflected ray; I is the Incoming ray and N is the object normal vector. For each ray that is incident on an object, we compute the reflected ray's direction and recursively track this ray to get the color for the original ray. To keep the computational overhead to a minimum, we limit the depth of ray interaction through reflection or refraction to 3.

The result after adding reflection is shown below.



3.6 *REFRACTION*

We currently perform refraction only on "glass" objects (refractive index ~1.5). For refracted ray calculation, we used the following equation due to (8) and (9).

$$T = \left(\frac{\eta_1}{\eta_2}\right) I + \left(\left(\frac{\eta_1}{\eta_2}\right) cos\theta_i - \sqrt{1 - sin^2\theta_t}\right) N$$

Where, T is the refraction/transmission ray; n_1 is the refractive index of medium 1; n_2 is the refractive index of medium 2; θ_i is the angle of incidence and θ_t is the angle of refraction. For each object on which a light ray is incident, we computed the color due to reflection as well as refraction. These are then combined using Schlick's approximation (10) to the Fresnel equations (11).

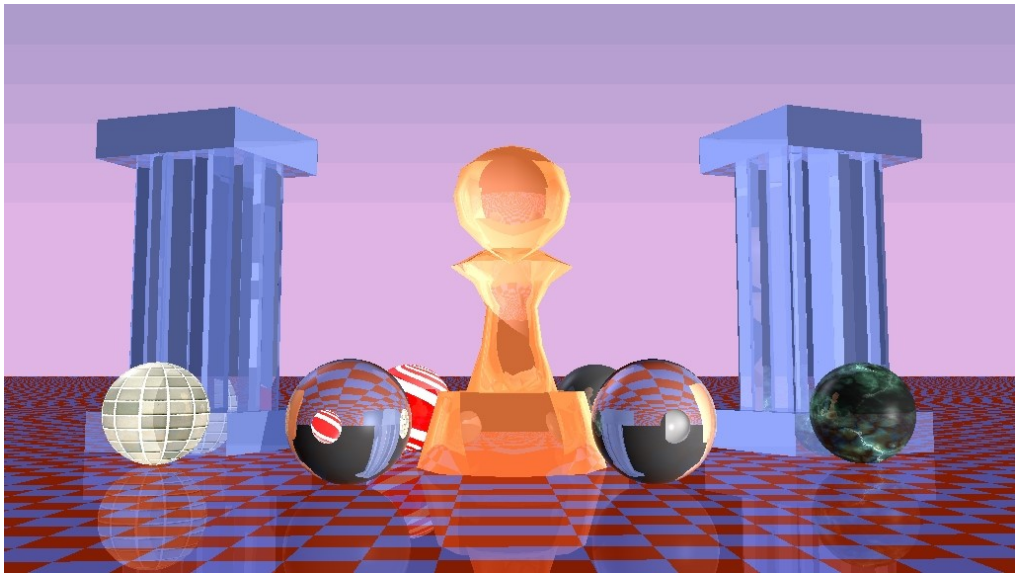
$$R_{Schlick}(\theta_i) = R_0 + (1 - R_0)(1 - \cos\theta_i)^5$$

$$with, R_0 = (\frac{\eta_1 - \eta_2}{\eta_1 + \eta_2})^2$$

We limit refraction only to simple implicitly modeled objects. Since time was limited, we could not get refraction working with implicitly modeled complex objects but we do note that the underlying mechanism is very similar to implicit objects such as spheres: Calculate two points of intersection for a ray with the object, perform refraction at each of these and combine results to get the final color.

Sample refraction in real life

The final image with all the features enabled is shown below.



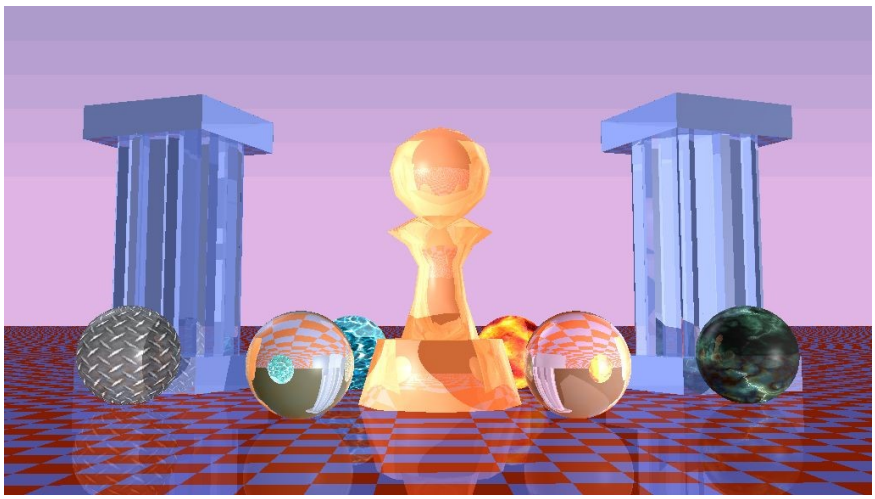
3.7 *TEXTURING*

We limit texturing only for implicitly modeled objects such as spheres but note that we easily extend this to complex objects. The UV mapping for spheres was found using (12) and various textures were mapped on to multiple spheres as shown in the scene. For each point P on the sphere, we calculate the unit vector from the center of the sphere to P. Assuming that the sphere's poles are aligned with the Y axis, the UV coordinates in the range for P are then given by the following equations,

$$u = 0.5 + \frac{arctan2(d_z, d_x)}{2\pi}$$

$$v = 0.5 - \frac{\arcsin(d_y)}{\pi}$$

These values are then mapped to the texture image to calculate the corresponding color.
Sample spheres with texture mapping are shown below



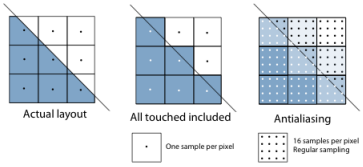
3.7.1 PROCEDURAL TEXTURING

We added procedural texturing to the sphere by adding normally distributed and skewed 2d noise (13), as shown in the final scene.

Sample Noise pattern on complex object

3.8 *ANTI-ALIASING*

We set out to do anti-aliasing using super-sampling, which involves casting several regularly spaced samples per pixel and using their average for the final pixel intensity. To render the pixel, primary rays are cast through each of the indicated sample points. The intensity value of the pixel is the average of all the samples. Because of insufficient time and longer duration of render tests we were unable to fully implement this feature.



3.9 *ANIMATION*

Finally, we animated our scene by generating several output images while gradually moving the camera the transparent spheres. The output images were combined using ffmpeg to create an mp4 video file that we report.


out.mp4

4 CHALLENGES

4.1 CHOOSING THE RIGHT PROBLEM

As outlined in Section 1, we looked at quite a few problems, including noise generation, generating interesting geometry, as demonstrated in http://paulbourke.net, or syntactic L-trees as shown on http://en.wikipedia.org/wiki/L-system. Although these problems seemed beautiful, in the end what drove our decision to choose Ray Tracing was the simple elegance of the intuition behind Ray tracing. This, taken with the fact that ray tracing is typically slow; we wanted to investigate potential bottlenecks to exploit.

4.2 EFFICIENT RENDERING OF EXPLICIT OBJECTS

One of the biggest challenges we faced was rendering explicitly modeled complex objects since this entailed finding intersection of each ray with every polygon of the object which was exceedingly slow. To avoid this, we implemented a clever optimization trick due to Kay and Kayjia that uses bounding boxes to greatly reduce the number of ray-triangle intersections performed. This resulted in roughly a 10 fold increase in the rendering speed for complex objects.

4.3 CROSS PLATFORM DEVELOPMENT

Each person in our team was used to his/her own programming environment and as a result, there was a need to make the application run across platforms. This posed several specific challenges such as the unavailability of certain low level flags across compilers, lack of the board support package (bsp), which contains default initializations for the tracer, in VC++, etc. Resolving these was an interesting experience for us.

4.4 VISUAL STUDIO DEVELOPMENT ENVIRONMENT

Perhaps the most troubling and time consuming bug was an issue with the file writing routine on Windows. When saving the output in .tga file, the output was coming out distorted (<<todo>> paste examples here). Initially, we debugged this from the angle of an issue with the renderer main loop itself. But this was not showing up on the OS X and Cygwin environments. After some debugging, we realized the file had more bytes than expected. We could not debug why this was happening, we were able to verify the input to fprintf was same in both the working and non-working cases. We added the functionality to generate .ppm files and this issue went away on windows.

4.5 TIME CONSTRAINTS

There were several additional features such as refraction with complex objects, anti-aliasing, soft shadows(distributed ray-tracing), Perlin noise, Turbulence noise, Bump maps etc. that we set out to implement but could not do so in time due to limited time and conflicting class schedules among us. We dealt this by making the code as seamless as possible and even though we split our work according to functionality, we ended up working together helping each other and improving the project as a whole and we compromised a little bit of quantity to provide quality.

5 NEXT STEPS

Apart from solving the challenges we mentioned above, there are several important additions we can make to this system such as:

- Add refraction to complex objects.
- Add turbulence with procedural texturing.
- Add Caustics/Photomapping/Distributed Ray-tracing.
- Use GPU/CUDA for faster calculations and rendering.(multithreading)
- Parallel processing: Reducing the render time further by dividing the workload.
- With GPU calculations, we could add real time interactions to the renderer.

6 CONCLUSION

We built a fully functional 3d rendering framework using ray tracing from scratch with several important features such as shadows, reflection, refraction, textures, complex object rendering, etc. This framework can be used a useful starting point for anyone who wishes to build more advanced features such as quick, interactive animation, etc. This was also an excellent experience for all of us since we learnt several core algorithms in ray-tracing and also gave us an exposure to several software engineering processes.

7 REFERENCES

1. http://www.scratchapixel.com/old/lessons/3d-basic-lessons/lesson-1-writing-a-simple-raytracer/implementing-the-raytracing-algorithm/. *scratchapixel.com.* [Online]

2. http://www.scratchapixel.com/old/lessons/3d-basic-lessons/lesson-7-intersecting-simple-shapes/. [Online]

3. http://en.wikipedia.org/wiki/Phong_reflection_model. [Online]

4. http://www-bcf.usc.edu/~saty/edu/courses/CS580/f14/hw/FinalProject/misc/OBJ_Parsers/parser_Andrew/index.html. [Online]

5. http://tavianator.com/2011/05/fast-branchless-raybounding-box-intersections/. [Online]

6. http://www.siggraph.org/education/materials/HyperGraph/raytrace/rtinter3.htm. [Online]

7. http://www.cs.jhu.edu/~cohen/RendTech99/Lectures/Ray_Tracing.bw.pdf. [Online]

8. http://graphics.stanford.edu/courses/cs148-10-summer/docs/2006--degreve--reflection_refraction.pdf. [Online]

9. http://www.flipcode.com/archives/reflection_transmission.pdf. [Online]

10. http://en.wikipedia.org/wiki/Schlick%27s_approximation. [Online]

11. http://en.wikipedia.org/wiki/Fresnel_equations. [Online]