

*Take Infinity*

Uthara Thelagar

Tanmay Patil

Himanshu Joshi

Anil Ramakrishna

Srikanth Madhava

*CS580*

Infinity Ray Tracer

# TABLE OF CONTENTS

Chapter 1 Introduction #

Section 1.1 Motivation #

Section 1.2 Objectives #

Section 1.3 Task Breakdown #

Section 1.4 How to run the program #

Chapter 2 Organization #

Section 2.1 Architecture #

Section 2.1.1 Graphics Pipeline #

Section 2.1.2 SW Architecture #

Chapter 3 Detail #

Section 3.1 BareBones Tracer #

Section 3.2 Object Parsing #

Section 3.3 Ray Casting #

Section 3.4 Lighting #

Section 3.5 Reflection #

Section 3.6 Refraction #

Section 3.7 Procedural texturing #

Chapter 4 Conclusion #

Section 4.1 Results summary #

Section 4.2 Conclusions #

Section 4.3 Next Steps #

# 1 Introduction

In this section we review our motivations for choosing Ray Tracing and our objectives. This is wrapped up with a breakdown of tasks assigned to individual members.

## 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 from model space and render in screen space with different effects such as texturing, procedural texturing, anti-aliasing etc. Coupled with the professor’s enthusiasm and experience we were very motivated to explore various topics for the project.

For this project, we considered L trees, Geometry from Paul Burke’s website and various youtube videos on gears, character animation etc. Finally, we found inspiration in the simple elegance of the intuition behind the ray tracer. We named our team ‘*Take Infinity*’ due to the infinitely many recursive possibilities that arise out of this approach to rendering. We wanted to understand how to construct this renderer from scratch with the possibility to explore any apparent bottlenecks.

## 1.2 Objecttives

With the above motivations, we had the following objectives:

* Develop the ray tracer pipeline from scratch
  + Understand the computational aspects of Ray Tracing
    - Is there room for optimization?
* Adding features such as reflection, refraction and anti-aliasing.
* Work in a team, with a distributed asynchronous workflow.
* Support multiple platforms for development to accommodate different.
* Support multiple output formats.

## 1.3 Task BreakDown

We understand this project was unique both in its scope and its ambitions with different folks interested in different aspects of the problem. While someone was interested in understanding the graphics pipeline whereas someone else was interested in understanding how the computation is distributed to understand avenues for improvement. As requested, we are providing a tentative breakdown of tasks, listed in order of flow of modules.

|  |  |
| --- | --- |
| Name | Tasks |
| Uthara | Main concept, design, implementation, skeleton tracer, shadows, shading, reflection, look development. Domain architect. |
| Himanshu | SW Architecture, including skeleton tracer, initial shader, multiple object intersection, github, cross platform support, Documentation. |
| Tanmay | Object Parser, OctTree optimization, s-t and normal interpolation |
| Anil | Refraction, debugging, code refinement |
| Srikantha | Textures, anti-aliasing, website |

## 1.4 How to run the program

The project is available on:

After downloading the program from the website, please untar it and you can use your favorite IDE to run the project.

### 1.4.1 Command Line

Just run the makefile:

cd Infinity\_Tracer

mkdir obj

make

./TracetoInfinity

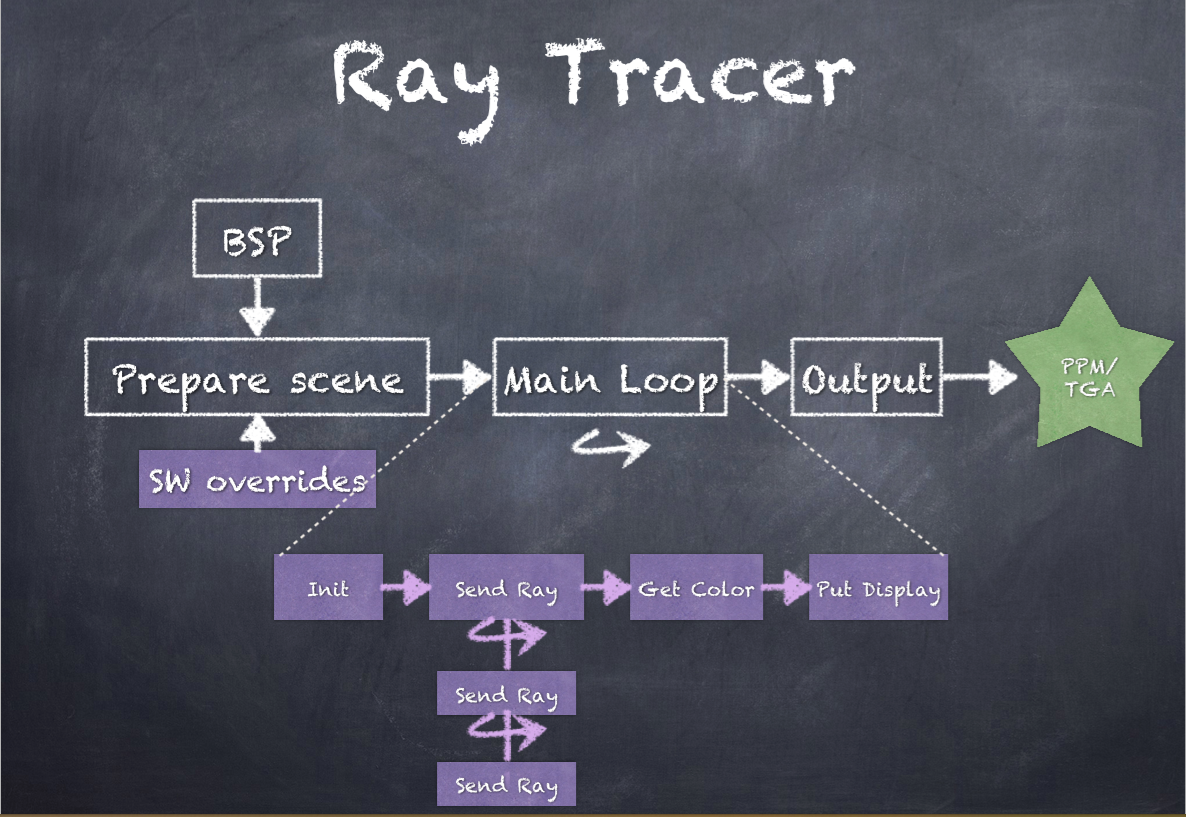
open render01.ppm

### 1.4.2 Visual Studio/XCode

just open the appropriate .sln or Xcode Project file. Press “Run” to generate the render01.ppm file.

# 2 Architecture

The architecture is as shown below:



## 2.1 Rationale

The rationale behind the architecture is to distribute the computation such that it can be parallelized. The scene is setup using the Board support package which includes the information about lights, any objects to be rendered via geometry equations etc. Any of the BSP contents can be overwritten via SW overrides in the init routine.

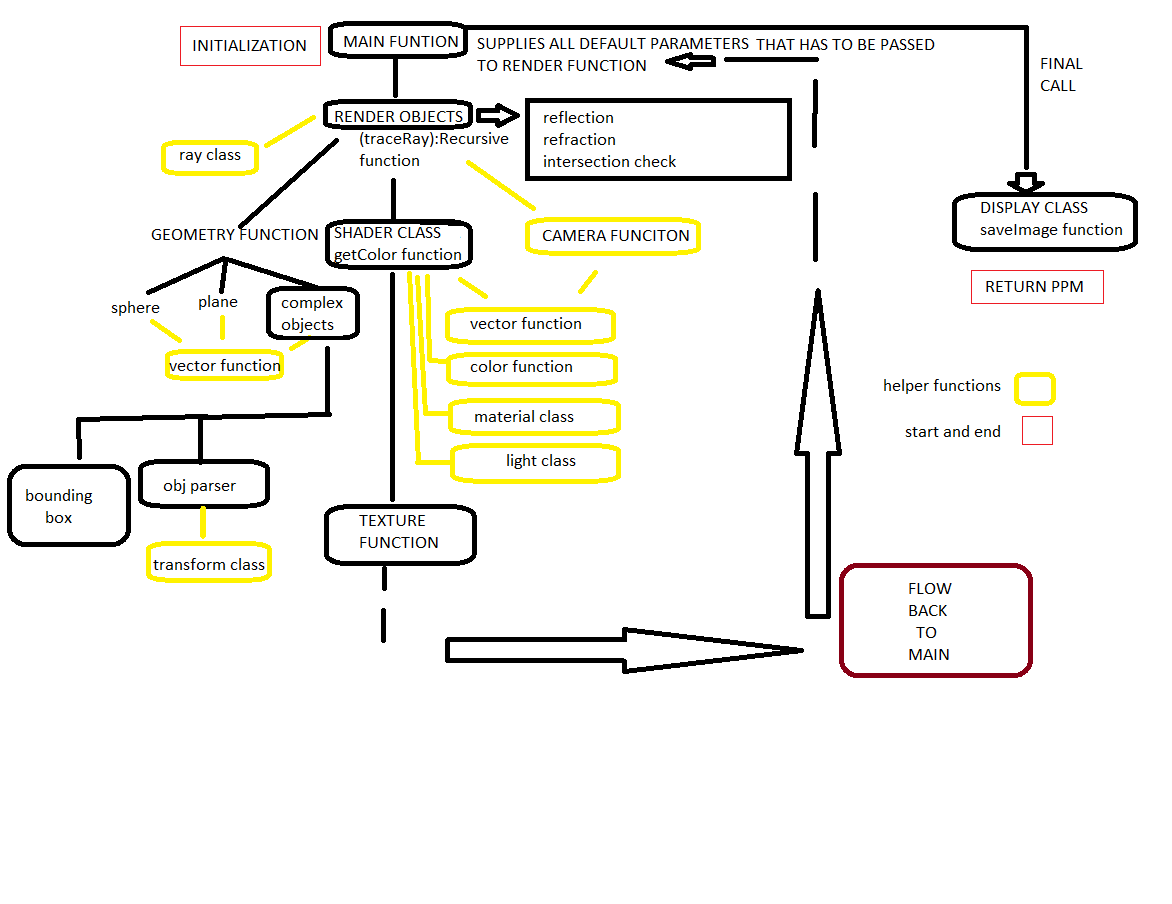
### 2.1.1 Graphics Pipeline

Once the scene is setup, the actual renderer main loop involves an iteration over all pixels in the screen space. Light is cast onto the scene and is then bounced off recursively on objects for reflection. Once it is determined what surface we wish to color, the shading equation is implemented in the get Color module which calculates the rgb values for the pixel. This is then put in the display buffer. The buffer is flushed to a ppm or a tga file once all the pixels are processed.

### 2.1.2 SW Architecture

# 3 Details

The SW architecture diagram is as shown below:



## 3.1 Barebones tracer

Basic frame work can be deconstructed in terms of each class:

* Render class
  + Implements the main renderer loop that determines which objects to render and makes calls to calculate color.
* Display class
  + save tga/ppm images to disk
* Camera
  + from given lookat,up(0,1,0) and position calculate X,Y,Z,
* Material
  + class for specifying color, diffuse, spec, ambient coefficients, reflection,refractive index
  + also extra parameter for texture files
* Geometry
  + super class from which is inherited by all implicit and parsed objects
* Vector operations
  + dot product, cross product, normalize, magnitude calculation
* Color check
  + checks rgb values to be between 0-1

## 3.2 Object Parsing

## 3.3 Ray casting

Send a ray from from the camera for every pixel center.

For perspective correction, convert the raw pixel value to NDC space -1to1 based on aspect ratio and 1/d(tan(FOV/2))( the distance to plane from camera).

Transform each ray from the camera such that it fits in the NDC space, finally the ray from camera in CamZ direction is shifted horizontally and vertically by necessary amount which is computed by scaling X and Y.

ray origin is the camera position:

horizOffset = camera->camX.scalarMult(horizScaleCeoff);

vertOffset = camera->camY.scalarMult(vertScaleCoeff);

primaryRay.origin = camera->position;

primaryRay.direction = (camera->camZ.addVector(horizOffset)).addVector(vertOffset);

primaryRay.direction = primaryRay.direction.normalize();

Use this ray to find intersection

Send the ray to calculate object intersection, once you get the intersection value for all objects that fall on the path of the ray, compare the values and object that wins is the one with lowest intersection value.

This is the object that will be visible in the final image, for which we will be doing shading calculations(Note: intersection value has to be > 0.0000001)

The trackRay function recursively calculates the intersection of ray with objects. At the top level this happens for the light ray sent from camera. For the recursive levels, this happens for the rays sent by the objects. This is used to calculate reflection and refraction. The depth of recursion is limited by a static variable to 2.

## 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 :

* Specular: This component accounts the reflection of the ray.

We check if: then,

## 3.5 Reflection/ Refraction

Reflection is calculated with this equation:

Here, I is the incident ray and it can be the primary ray or a reflected ray (ie. lower in the recursion hierarchy).

## 3.6 Texturing

<<todo post examples or pointer to the image with textures in it>>

## 3.8 Anti-aliasing

An antialiasing scheme similar to what is implemented for HW6 is used. After sending multiple rays we obtain a weighted average of the color of each pixel. This increases render time. We have not been able to get this to work completely and is disabled in the program.

## 3.9 Scene Setup

The scene is setup using a board support package (BSP). This lends itself to easy testing. Anything from the scene can be removed or deleted via a quick change to a .h file. We tested with various camera positions, implemented various transformations. We tested with multiple object files back and forth between maya and our raytracer. This helped us fix the position of the objects to decide the look and feel of the final renderer.

# 4 Challenges Faced

## 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 Ray Tracing Objects from OBJ files

Most of the ray tracing algorithms we found dealt with sending rays through geometry – such as spheres or planes – we wanted to efficiently render a scene using triangles obtained from an OBJ parser. <<@todo add more details about optimizations>>

## 4.2 Scheduling with multiple classes

Coordination between multiple members with varying schedules was a problem which we anticipated. We dealt with this by using a distributed development model enabled via the asynchronous git workflow. Additionally, the core ray tracer was developed with the SW architecture principles of mechanism & component reuse to reduce coupling. Thus, OBJ parser, Reflection and Refraction could be developed concurrently, once the skeleton application and relevant API were ready.

## 4.3 Development Environment

Due to the various platforms people were using, and the specificities of the way the C++ compiler behaves in various environments, we faced an unanticipated challenge. The issue was several standard C defines from the math library were not usable in VC++!

INFINITY, M\_PI were unavailable on VC++ while abs was unavailable on the OS X LLVM compiler. Finally, the board support package (bsp) which contained default initializations for the tracer would were not compilable in VC++ but worked as expected in both the OS X XCode’s LLVM compiler and the gcc compiler under Cygwin and OS X. This set us back by a day or so.

## 4.4 Visual Studio Development Environment

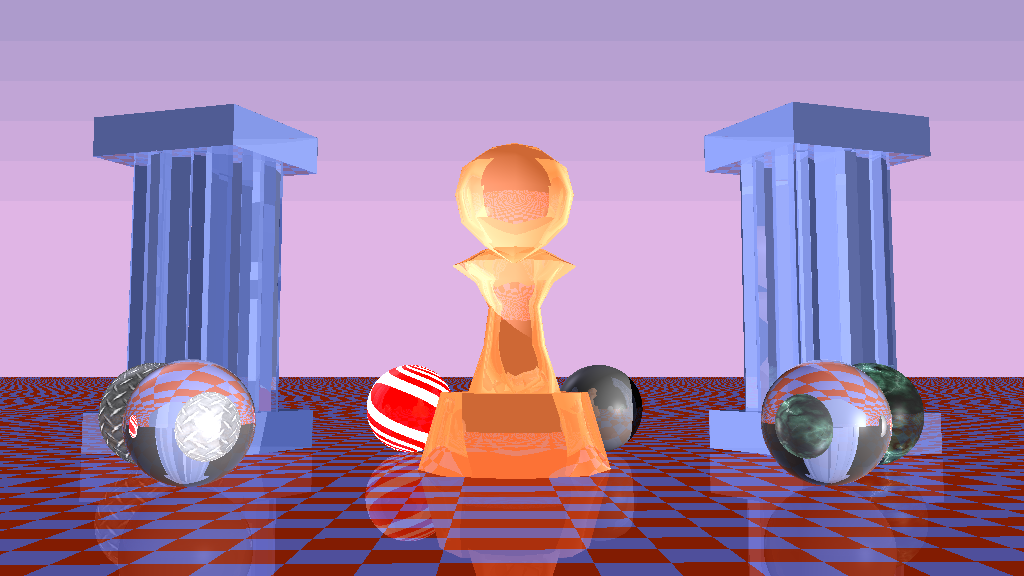
Perhaps the most troubling and time consuming bug was an issue with the file writing routine on Windows. When saving the output in .tga file, the output was coming out distorted (<<todo>> paste examples here).

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.

# 5 Conclusion

## 4.1 Results Summary

This is how the final image looks like:



An animation was generated using transforms and this video was present in class. It is available at the project website.

## 4.2 Next steps

This project has multiple possible avenues it can branch. Here we are listing the ones we feel are important. Our project was developed using git and will be made available online for people to extend.

* Parallel processing: Reducing the render time further by dividing the workload.
* Procedural texturing: Generating noise for use as texture file.

The quote is bold and distinctive. The galleries include items that are designed to coordinate with the overall look of your document.

# References

**First reference.**

Scratch Pixel tutorial on Noise http://www.scratchapixel.com/old/lessons/3d-advanced-lessons/noise-part-1/pattern-examples/

**Additional references.**

Ray Casting calculation

http://www.macwright.org/literate-raytracer/

Scratch Pixel tutorial on Ray Tracer http://www.scratchapixel.com/old/lessons/3d-basic-lessons/

Geometry calculations for Sphere and Plane Andrew Glassner Intro to ray tracing

Reflection http://www.cs.jhu.edu/~cohen/RendTech99/Lectures/Ray\_Tracing.bw.pdf

Refraction

http://graphics.stanford.edu/courses/cs148-10-summer/docs/2006--degreve-- reflection\_refraction.pdf