408 Assignment 1 Report

James Drown : 300387844

March 2021

Whitted Ray-Tracer

Core

I implemented a Whitted ray tracer algorithm in c++. The program is very simple, and when run will develop render a scene (comprised of planes and spheres). The resulting image will be output as a .ppm file.

TO RUN THE APPLICATION:
Please include the "ext" folder (found in the "WhittedRayTracer" folder) in included directories. The program should be able to run, simply by running in release mode. There will be two output images called "WhittedCore.ppm" and "WhittedOutputAA.ppm", which will be located in the ".". Open the image with GIMP, Photoshop, or you can view it online at : `http://www.cs.rhodes.edu/welshc/COMP141_F16/ppmReader.html` (Beware this convert's the file to .png, I have not experienced any problems with this however)

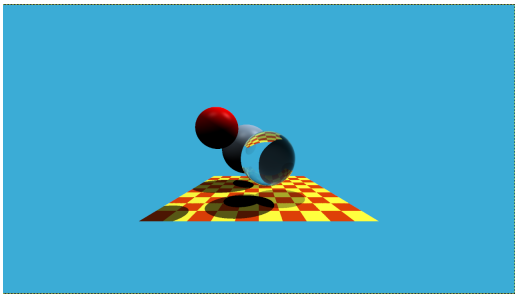
Currently the only objects that work with the ray tracer are plane's and sphere's. If you wish to alter the scene, please refer to the "DefineScene" function and options struct in Main.cpp. I followed the following tutorial to get a general understanding of the work, however I implemented all of the work in my own way.

`https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-overview/light-transport-ray-tracing-whitted`

Libraries used:

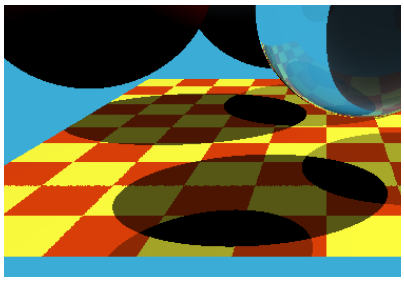
- Stdio.h: Standard output
- Vector: Creating lists
- FStream: Outputting files
- GLM: For general math functions e.g. vec3();

Results (Note: This is a screenshot of the render, rather than the actual file due to conflicts with Latex):

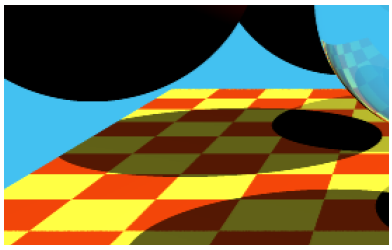


Challenge

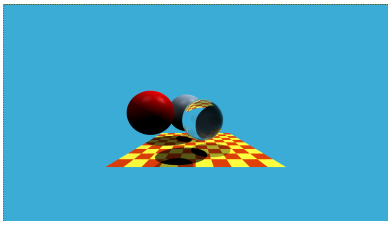
I also implemented one section of the challenge, which was Anti-Aliasing. I did this simply by offsetting the rays direction by 2.5 in each direction and taking samples of each. I then average the values. The results are as follows:



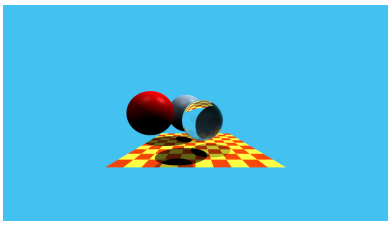
(a) Core Zoom



(b) AA Zoom



(c) Core Standard

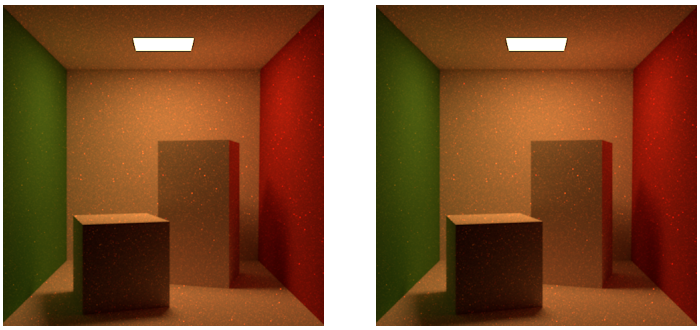


(d) AA Standard

Figure 1: Core vs. Anti-Aliasing

PBRT Profiling

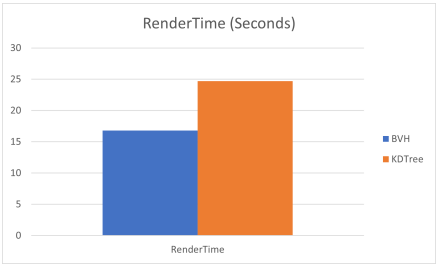
I ran three separate iterations of the cornell-box pbrt scene. I began by running the BVH acceleration structure, which is the standard for PBRT. I then ran the KDTree acceleration structure to compare the two.



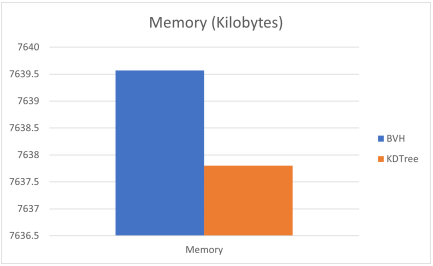
(a) BVH (b) KDTree

Figure 2: Image results of different acceleration structures

The first thing to note is that there were no significant changes in appearance of the two renders, as we would expect from only changing the accelerators. Two factors that were affected were the Memory and Render Time. While the memory stored using BVH went up very slightly (less than 2 kilobytes) it greatly improved the speed of the rendering process.



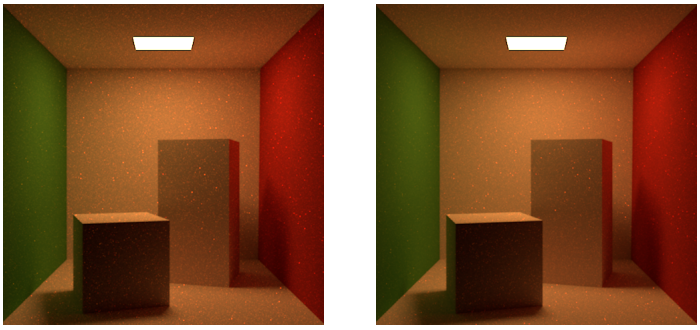
(a) Render time comparison of BVH and KDTree Accelerators



(b) Memory comparison of BVH and KDTree Accelerators

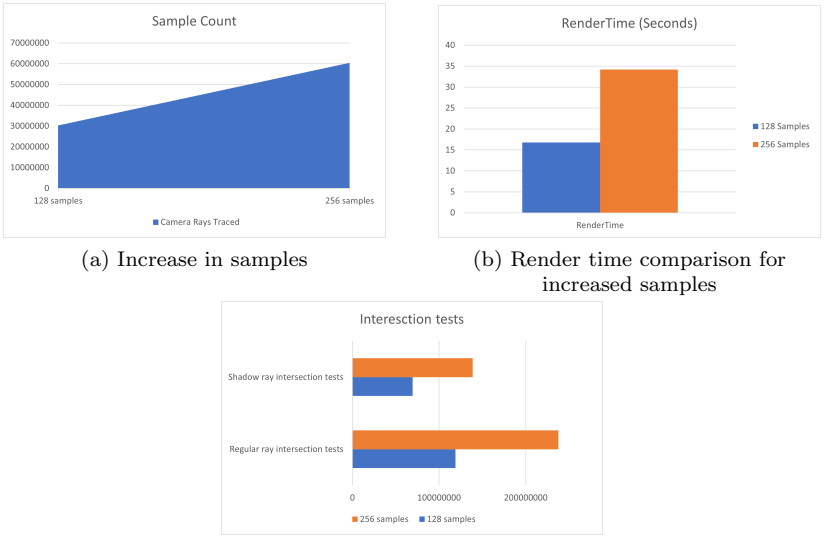
Figure 3: Comparison results between acceleration structures.

After this first comparison, I increased the number of samples for the render from 128 samples to 256, while keeping the BVH accelerator. This resulted in a slight change in clarity/less random light artifacts, though it is not immediately noticeable from a distance.



(a) 128 samples (b) 256 samples

Figure 4: Image results of different sample rates



(a) Increase in samples

(b) Render time comparison for increased samples

(c) Intersection count comparison for increased samples

Figure 5: Render time comparison between sample count.

As shown in figure 4, The increase in samples directly affected the render time and the intersection count. For this particular scene, it would be my recommendation to use the PBRT's default accelerator, BVH, with a sample count of 128, unless the image was going to be viewed in a close proximity. This would give a balance between rendering time and quality.