Ray Tracer

CS488 Final Project

Hao Pan

hlpan

20618961

December 3, 2018

Overview

For this project, I have implemented a ray tracer with several extensions. I have completed most of my objectives, although the project did not meet the level of completion that I had hoped.

I have implemented a scene depicting a sword being used by an unknown user to slice orange semi-refractive spheres. A weapon from a popular video game is also shown in the background.

Statement

For my final project, I wish to use ray tracing techniques to render an indoor scene with objects of varying material. The primary purpose of this project is to display everything I learned in this course.

I want to show that I understand the different graphical methods taught, and that I am able to choose the most suitable ones for my work. Additionally, I want to show my creativity and sense of design by modeling a scene of my own choosing. Lastly, I want to show my love of games through this project by displaying iconic objects from some of my favorite games.

Communication

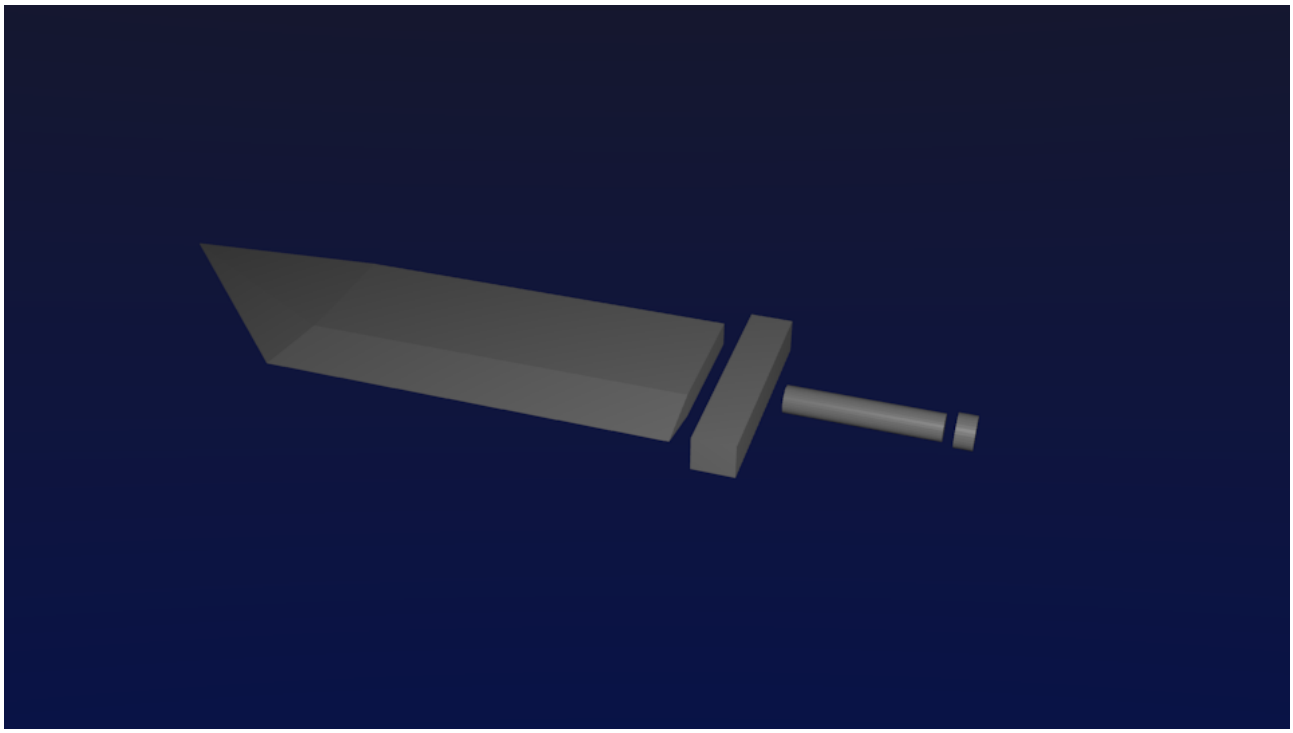
Input: A lua scene file

Output: A rendered image in png format

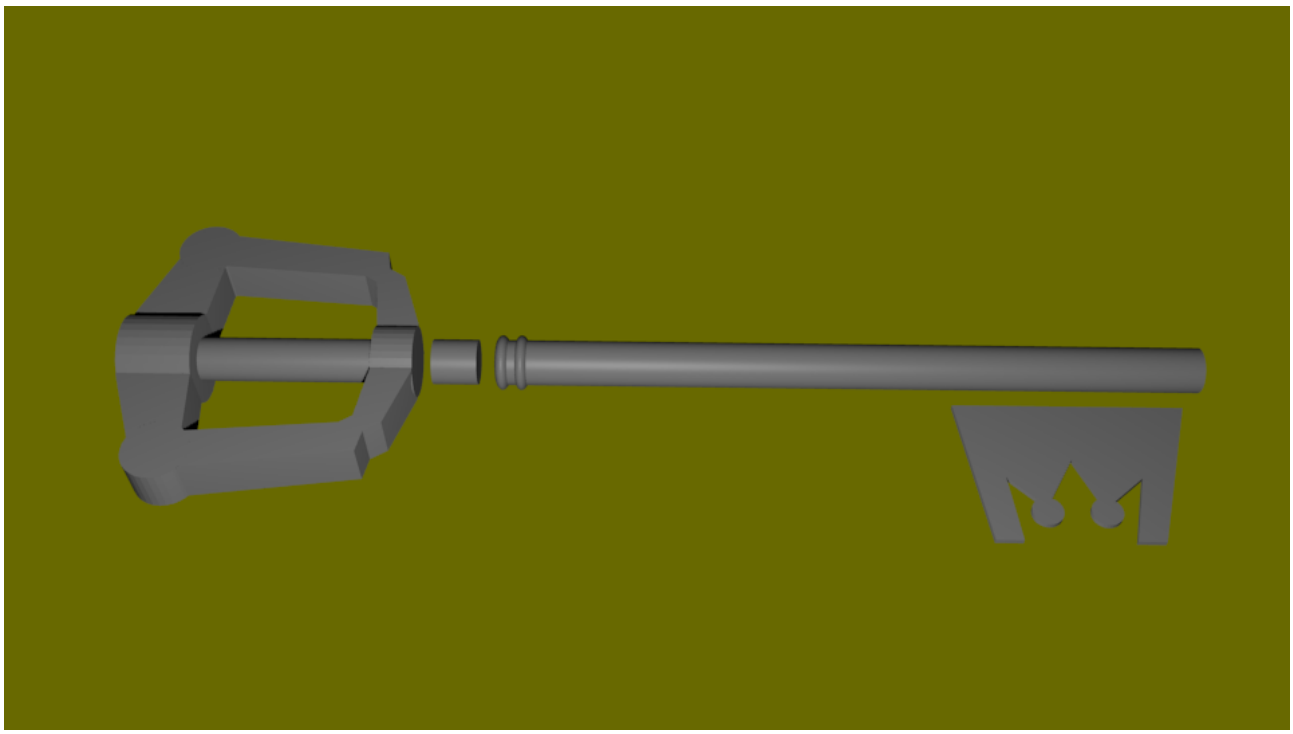
Technical Outline

Modeling Objects

Although my original goal was to model three swords, I only modelled 2 due to time constraints. The first sword I modelled was the *Buster Sword* from a popular video game *"Final Fantasy VII"*. I modelled different parts of the sword as different objects in Blender and hierarchically placed them together in my scene. The Buster Sword consists of a grip, a guard, the blade itself, and a decoration placed at the end of the grip (pommel). I had trouble rendering the blade object with my ray tracer, so I replaced it with a rectangle of similar dimensions. The different parts of the sword can be seen below:



I also modelled a weapon known as a *Keyblade* from the game *"Kingdom Hearts"*. It consists of a handle, the handle grip, the pole (divided into two parts), and the teeth of the key. Its parts can be seen below:



Multithreading

I implemented multithreading in order to render the images more quickly. All available threads will be used to render. Each thread will retrieve a pixel and will attempt to render it. When it has completed rendering the pixel, it will add the result to the appropriate data structure (matrix for supersampling, or image otherwise) and retrieve the next pixel to render.

Adaptive Anti-Aliasing

Adaptive anti-aliasing was implemented to remove the jaggedness of images. My implementation of supersampling does not do anti-aliasing well because it does not sample enough unique points. My implementation of adaptive anti-aliasing is done after the image has been fully rendered. Pixels are compared with their neighbours, and if the colour difference is sufficiently high, more rays are cast to the pixel with (with slight variance) and the average result is used. I used a simple formula to calculate colour differences (sqrt((r – n.r)^2 + (g – n.g)^2 + (b – n.b)^2), and if the colour difference is greater than 0.1, more points will be sampled.

The sampled points are not chosen on a random basis. I used the same points outlined in [TODO]. My program supports 1, 2, 4, 8, and 16x anti-aliasing.

Soft Shadows

To make my scene look more realistic, I implemented soft shadows. They were outlined in [1], as well as in Peter Shirley's notes from the course homepage[3]. In this implementation, the light source is randomly deviated from its original position and cast to a point. This is repeated a sufficient number of times, and the resulting colours are averaged to get the colour of the surface point.

Reflection

A basic reflection was implemented to reflect objects off of reflective surfaces. When the eye ray hits a reflective surface, it will be reflected about the normal and the resulting ray is cast from the surface point. If it intersects with a surface, the surface colour will be the reflection colour. Note that reflection is not sampled recursively, and only the surface colour and shadow will be considered towards the reflection colour. Once the reflection colour has been retrieved, it will be averaged with the surface colour via the surface's reflection constant.

An additional lua command (set_reflectiveness) has been implemented to set a reflective constant for an object. It takes in one number.

Refraction

Refraction was very difficult for me to implement. I looked through various papers, including a pixar one. I couldn't get the described implementation to work. Eventually I was able to implement refraction using the method in Peter Shirley's book [3]. When a ray hits a refractive surface, it will bend according to the ratio of the two medium's refractive constants (n/nt), where n is the refractive constant of the current space, and nt is the refractive constant of the entering space. I keep the refractive constants in a stack by pushing every time I enter a new space and popping when I exit the space.

An additional lua command (set_refractiveness) has been implemented to set a refractive constant for an object. It takes in one number as input.

Gloss & Metallic Reflection

By sampling more surfaces (with a little offset) during reflection, I was able to implement glossy reflection. When a ray hits a reflective surface, several more rays will

be randomly cast from the intersect location. The average results of those ray casts will be used for the reflective colour of the surface.

Metallic reflection is done by turning a colour into gradiant by using a colour intensity formula (0.2126r + 0.7152g + 0.0722b). This resulting number will be the r, g, and b value for the colour.

An additional lua command (set_metallic) has been implemented to set a metallic constant for an object. It takes in one number for whether or not an object is metallic (1 for yes and 0 for no).

Photon Mapping

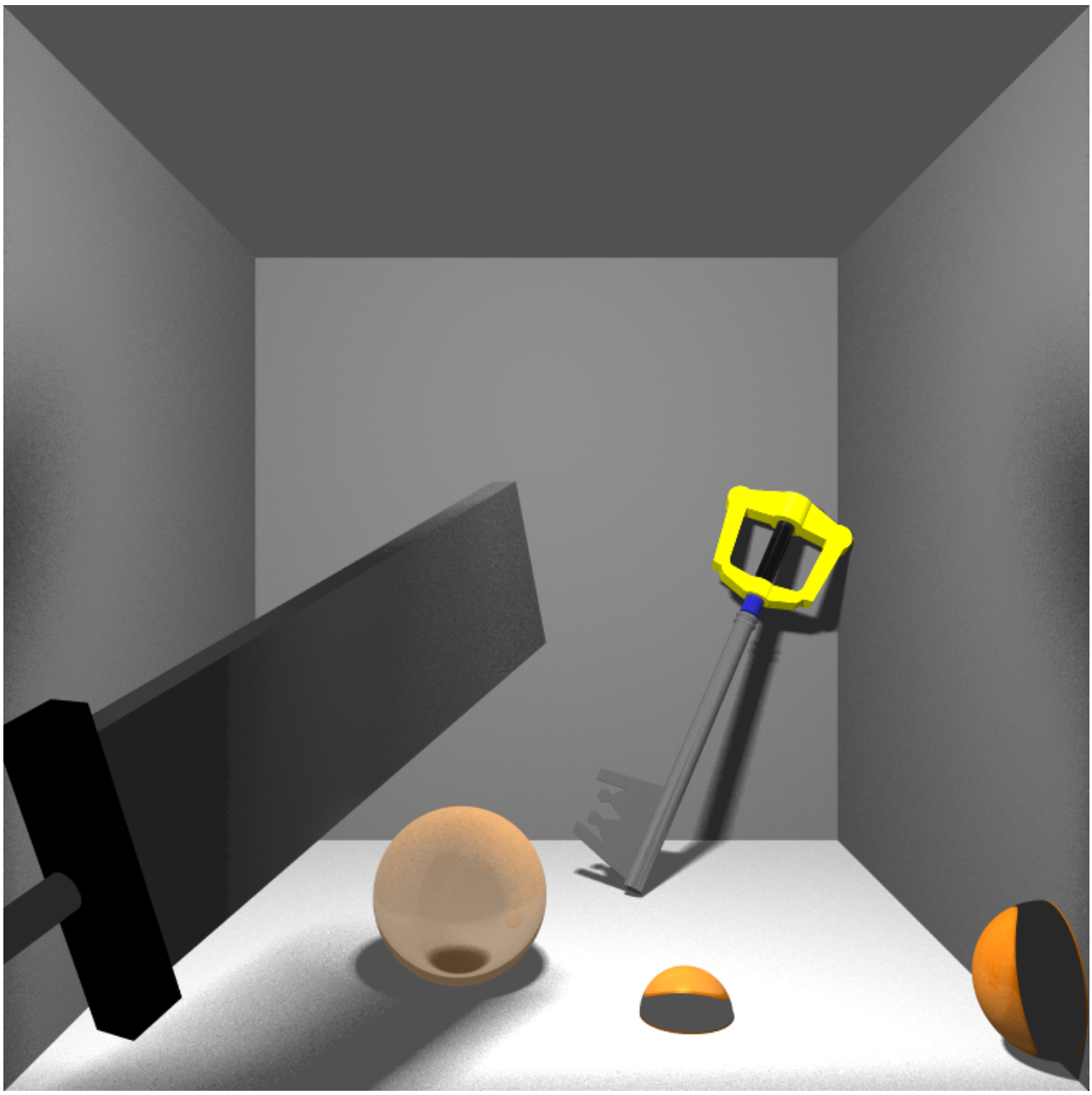
Although I did not finish implementing photon mapping, I was able to implement the first pass by tracing photons from the light source. I used Russian Roulette to determine a photon's path when it hits a surface, and photons will only bounce a maximum of 10 times. I followed the pixar paper [6] to do the photon tracing, and the photons are stored in an unordered list whenever it hits a surface.

I also added the photons to a kd-tree, although I didn't implement tree search, so the tree is unusable.

I was not able to implement the second step, which is rendering an image with the photons because the formulas I found did not work.

Final Scene

The last step was to create the scene, shown below:



NOTE:

- Some sources have information for multiple objectives:

- Peter Shirley's tutorial [3] that's linked from the course homepage describes gloss and refraction.

It also has a section explaining soft shadows.

- Lecture slides from Carnegie Mellon University [4] have detailed diagrams for a lot of ray tracing

techniques including Adaptive supersampling, glossy reflection, and soft shadows.

Bibliography

[1] Voica, A. (2015, August 13) Ray tracing for beginners. Retrieved from http://www.embedded-computing.com/embedded-computing-design/ray-tracing-for-beginners

[2] Pluralsight. (2015, September 4) Understanding caustics for a higher level of realism in your renders. Retrieved from https://www.pluralsight.com/blog/film-games/understanding-caustics-higher-level-realism

[3] Shirley, P. (2016) Ray Tracing: The Rest of Your Life. pp. 201-235

[4] Treuille, A. (2009) Advanced Ray Tracing. Retrieved from https://www.cs.cmu.edu/afs/cs/academic/class/15462-s09/www/lec/13/lec13.pdf

[5] Schindler. G. (2007) Ray Tracing and Photon Mapping. Retrieved from https://www.cc.gatech.edu/~phlosoft/photon/

[6] https://graphics.pixar.com/library/HQRenderingCourse/paper.pdf