CSE 167 (Fall 2021) Raytracer Final Project Guide – I

In this final project topic, you will write a ray tracer. Raytracers can produce some of the most impressive renderings with high quality shadows and reflections. The guide is broken into two parts as two separate possible directions.

- Modify the scene in your HW3 so that it becomes a ray tracer.
- Ray tracer used to be one of the homework in some of the previous CSE167. You can also follow a similar guide modified from the previous years (we provide it in the Raytracer Final Project Guide – II).

For both routes, you will write a ray tracing program that renders a scene.

This document focuses on the first route, where we build a scene in a similar style as your HW3.

For the second route, we provide test scene described in a plain text format. We provide a starter code for the second route that features a basic scene parser. We also provide a few reference screenshot for you to compare the result.

Guide I Introduction

In this guide, we expect you to write a ray tracer from scratch. The basic ingredients are given in the course slides. In particular, the rasterization-based OpenGL rendering is not needed. However, if you want, you may use GLUT to access the framebuffer to store your rendered result and let it show on the screen.

Expectation

Develop a raytracer that features Page-12 of the slides https://cseweb.ucsd.edu/~alchern/teaching/cse167_fa21/7-1RayTracing.pdf

That is, the lighting of each intersection has

- the Lambert diffuse light similar to HW3 (run through all the light and take dot product between the normal and the light direction) *including the visibility test by a shadow ray* (see page 10).
- and a recursive mirror reflection.
- You can add ambient term like in HW3 to fake the global illumination.

Your rendered image should feature shadow and mirror reflected image on each shiny objects (like the bottom-right image of page 12).

A bonus point direction is to include global illumination. In that case, you don't need the fake ambient light. Instead, follow page 61 of the slides. Note that the recursion depth is corresponds to the number of bounces of each ray. You need to sum the colors with result from all possible number of bounces like described in page 65. You may just present the result given by the sum of 1 bounce, 2 bounces and 3 bounces (and ignore all the terms for larger number of bounces). Or you may apply the Russian Roulette method to obtain the sum of all numbers of bounces. For more detailed information, we refer you to the bonus point guides for Gude—II.

Turning HW3 framework to a ray tracing framework

OpenGL is a rasterizer. It doesn't feature ray tracing pipeline. So we will abandon most part of HW3 codebase. In fact the whole program will not use GPU. Everything is programmed in C++ and is run on CPU.

Although we will not use much of HW3 framework, many C++ classes can be reused (with modifications if needed), such as Camera, Scene, Geometry (in particular Cube and Obj for the

.obj file reader), and so on. You may need to create additional class such as Ray, Intersection, and geometric elements such as Triangle and Sphere.

For a ray tracer, you want to be able to run through each pixel to shoot ray into the scene. And for each ray we want to be able to check through all the geometries in the scene to perform the intersection test. So, a simple setup is to have

- an array of all pixel (the image) to store the resulting color.
- a list of all triangles (and/or other elements such as spheres, if any) so that you can iterate through them during the ray-scene intersection test.

Traversal of scene graph

In ray tracing, we want to prepare the list of all triangles to be draw. In HW3, the scene is described with a scene graph. Back then when we were using OpenGL-based graphics in HW3, a depth first search subroutine visits each geometry and draw each geometry individually. In ray tracing, we can still reuse this depth first search to visit each geometry to collect triangles into our "array of all triangles to be drawn" at the initialization stage. Once we have the list of triangles, our ray tracer can perform ray-scene intersection test by iterating through the list.

Showing the final image

You can look at the Screenshot.h code which is how we turn array of color data into an image file. You can use a similar code to export your ray-traced image into a .png or .jpg file. You may also store the color data into a texture. Then, use OpenGL to render a simple square, on which you put the texture to show the result. There are also ways you can directly write data into the framebuffer, so that the screen directly shows your image.