Satwiko Wirawan Indrawanto - 6201539
Basar Oguz - 6084990

Assignment 1: Whitted-style Ray Tracer documentation

Introduction

This project has been developed to function as an extendible and reusable framework for ray tracer programs. The framework was built upon and tested with the template provided in class, that typically handles a linear pixel frame buffer, plotting on screen and game loops.

The delivered project consists of the basic geometric primitives (plane, triangle, sphere) that could be placed in a “scene” and tested for intersections with a ray, point light sources, a pinhole camera implementation and a Whitted-style ray tracer, that accounts for shadows, reflections and transparency. The project provides a basic UI to control the free camera at run-time and the shading calculation on the primitives are done according to a basic but extensible material implementation.

The main functionality of the program is to render a scene consisting of the test bed primitives, point light sources and a camera, using a physically-based Whitted-style ray tracing algorithm. It was built to be extended and improved by acceleration structures for achieving a real-time performance.

All implementation details we unanimously decided upon and implemented together. Although the coding workload was divided, as Basar has implemented the primitives and intersection routines, and Satwiko the Whitted-style ray tracing algorithm and camera manipulations.

Architecture

The most basic element in the ray tracer test bed is the Ray class that consists of a three dimensional origin position and a direction vector, as well as a float indicator, that signifies a point (origin + t*direction) on the ray. Another high level abstraction is a Scene, that contains a pointer to a Camera and a std::vector of pointers to Primitives and Lights in the scene to be rendered.

The Primitive base class, that is extendible by any number and complexity of primitives, is inherited by Triangle, Sphere and Plane, and contains virtual ray-intersection methods, normal calculation and a Material instance. The Material class contains information about the surface specifications, such as a base color, and shader type (diffuse, mirror or glass). The Light class consists of a position and a base color. Triangles are defined by the positions of their corners and a normal vector. The triangle-ray intersection method uses the Möller-Trumbore algorithm that calculates uses barycentric coordinates

The Game class initializes a Scene and a RayTracer instance that takes in the information of the scene (primitives, camera and lights) and renders the scene on the screen surface in every Tick using the Plot function of the Surface class.

Camera class

The Camera class conceptually consists of a position, a direction that it's “looking at”, positions for the screen center and the three screen corners. The camera can be translated in run-time using (W,A for manipulation in the z-axis, A,D in the x-axis) and rotated about the x-axis (Down-Up keys) and the y-axis (Left-Right keys). The Camera class generates and stores normalized rays for every pixel on the screen according to the predefined screen width and height, also taking the aspect ration into account. Also included are a number of helper functions for calculating the screen corner positions and updating the position. The generated and stored rays are tested for intersection and then traced recursively until (and if) they intersect with a light source, and the respective pixel of the originating ray is colored (on the screen) by the RayTracer::render() function.

List of possible inputs:

- W, A, S, D, Q, E = translating the camera in various directions
- Arrow Keys = rotating the camera about various axes
- + and - keys = zooms in the camera

RayTracer class

The main functionality of actually tracing a ray, including visibility calculations and Whitted-style ray tracing on diffuse and reflective surfaces are implemented in the RayTracer class.

The render() method calls the GetColor() method that returns a float3 value as a color, and this value is cast to an integer, clamped, and plotted on screen.

The GetColor() method takes a Ray as an argument and loops through the primitives in the scene to find intersections, and operates on the nearest intersection (using a z-buffer like structure). Whitted style shading is applied on the nearest intersection point, depending on the Material of the primitive that has been hit.

If the material that has been hit is “diffuse”, DirectIllumination() method is called. If the point hit has a direct unoccluded path to a light source, it is shaded using the Material base color, by taking the angle of incidence and the inverse square of the distance to the light source into account.

In the case that the Material hit being a “mirror”, the ray is reflected using a helper member function Reflect() and traced again, recursively, with a call to GetColor().

However, if the hit Material is a “glass”, namely a dielectric material, a ray is refracted, and an additional ray is reflected. Then these two rays are recursively traced by the GetColor() method, that generates two different color values. These two different color values are blended by a Fresnel coefficient that is calculated by the Fresnel() method, that takes into account the angle of incidence and the surface normal.

External libraries and dependencies

This is a cross platform project (Windows and Mac OS), for this we use GLM for vector and matrix math. FreeImage is used in the template for image loading. SDL2 is used for low-level window handling and key detection/interaction. QuarticSolver headers by Jonathan Zrake, NYU, were used to solve intersection equations between a Ray and a Torus. tinyobjloader by syoyo is used to load wavefront obj files.

Additional points

Loading objects from file and support for meshes are implemented in Scene class using tinyobjloader.h. Reads an obj file and converts the vertices to triangles stored in primitive list. For instance, the white box in the example scene shown is an obj file loaded. Bigger obj files can be imported but has significant performance issues.

Refraction is also implemented, currently using 1.52, considering the physical properties of a “glass” as the default refractive index.

Complex primitives implemented, having cylinder and torus.

Multithreading currently done when generating rays and drawing on screen using #pragma omp parallel. More is planned to be implemented in the future as the current thread.h it is now not compatible with Mac OS. Cross platform compatibility.