Advanced Graphics - Assignment 1 : Whitted-Style Ray Tracer Framework

Dustin Meijer(s5726328) & Luuk van de Wiel(s4088212)

TL;DR

Basic Functionality

- Generic and Extensible architecture implemented
- Camera can rotate around, but can only translate according to world coordinates
- There is a list of input commands available at the end of this report
- Planes, Spheres and Triangles implemented
- Material class exists and is used to store colour information and has a type to indicate if it is DIFFUSE, MIRROR or GLASS(start for Dielectrics)
- We have multiple scenes that you can switch using a preprocessor directive in Scene.cpp
- Whitted-Style Ray Tracer using Direct Illumination and Full Reflection is implemented

Extra Features

- For-loops were annotated with OpenMP # pragma commands to start parallelization (WARNING: Only works well in Release and not in Debug mode, turn off if you want to compile debug mode)
- Triangle primitive is implemented
- UI

Introduction

For the course Advanced Graphics at the University Utrecht we focus on studying methods for creating physically accurate images from virtual environments and how to make the process of creating these images run at an interactive frame-rate. As a start for doing this, we were assigned to create a Whitted-Style Ray Tracer together with a framework, so extend this work can be extended for other techniques.

The basic functionality required was the following :

- Implementing a generic and extensible architecture for a ray tracer.
- A ‘free camera’ with configurable position, orientation, FOV and aspect ratio.
- A basic UI or controls to control the camera at run-time.
- Support for at least planes and spheres.
- A basic material class.
- A basic scene consisting of a small set of these primitives(spheres & planes).
- A Whitted-style ray tracing renderer to demonstrate and test the architecture.

There was also the option to attain additional points by implementing some or all of the following:

- Support for triangle meshes, e.g. using ‘obj’ files to import scenes.
- Support for complex primitives.
- Texturing.
- Flexible lights: point lights, spot lights, IES profiles.

- Refraction and absorption.
- Efficient and generic multi-threading.
- Performance tuning.

The rest of this document will discuss what functionality we decided to implement, how it was implemented and what difficulties we faced. We will first discuss the architecture of the project, then the ray tracer that is implemented, followed by what was done for the camera. Then we will dive into the primitives that are implemented, as well as the material class and scene(s) made. Afterwards, we will go through what bonus features we decided to implement. At the end of document a list of input commands can be found together with the work division of this project.

Basic Functionality
In this section we will discuss what basic functionality was implemented.
Architecture
For the architecture of this framework, we were handed a starter project by Jacco Bikker that contained the raw implementation of getting text, lines and single pixels to a window. We could create the main loop in the Game::Tick(float dt) function. We added a Camera, Renderer, Primitive, Sphere, Plane, Triangle, Light, Scene, Material, and Ray class to this starter project.
The basic initialization flow is as follows :
Game::Init -> Creates Scene and Renderer, Scene::Init creates all the primitives of the scene and the camera, Renderer::Init asks Camera to generate the initial set of rays, these are stored in Camera.
The basic loop is as follows :
Game::Tick -> Renderer::Render ->

For every pixel Renderer::Trace -> If primitive hit has Diffuse material, check for DirectIllumination for every LightSource (Renderer::DirectIllumination), else if primitive hit has a mirror material, send out a new Trace.

For every pixel in the pixel buffer Surface::Plot -> Plot the pixel to the screen

For extensions, the Renderer::Render function can have more cases added to handle different types of materials (or maybe even combined materials). Primitives can easily be added by just sub-classing Primitive. The way Lights are handled might have to be changed slightly to support different types of lights.

Ray Tracer
The Renderer class is basically the Ray Tracer for us. The Ray Tracer we have implemented supports Direct Lighting and Reflections. This functionality can be found in the Renderer::Render and Renderer::DirectIllumination functions, at the time of writing Dielectrics were not yet implemented.

Camera
The Camera class is responsible for being our representation of our eyes in the virtual environment. Our camera class gets initialized at (0,0,0) with a viewDirection of (0,0,1). A virtual

screen to target our rays is then constructed in the direction of our viewDirection at a distance of d. Modifying d will allow us to simulate different Fields of View. The camera can Rotate around all axises in Local Space and is capable of translation along World Axises. Rays are generated once during the initialization and are kept updated whenever the camera moves.

Primitives

We created a Abstract Primitive class to serve as an interface between primitives and other areas of the framework. The primitive class requests a child of it to implement a bool CheckIntersection(Ray* ray) and a vec3 GetNormal(vec3 point) function. It also comes with a default red Diffuse material.

The primitives we implemented were :

- Planes
- Spheres
- Triangles

It is easy to implement other primitives by just subclassing Primitive.

Material Class

We have a very basic material class that has an enum to denote what MaterialType it is and contains a color vector. This class can be easily be changed to support different functionality if so required.

Scene(s)

We created a basic Scene class that contains a list of primitives, a list of lights and a camera. Currently we have a set few scenes hard-coded in the scene constructor, but this can easily be changed to expect the primitives, lights and camera. For us, it is currently just a container.

Bonus Features

In this section we discuss the bonus features we implemented.

Triangle Primitive

Our choice of primitive to add was the triangle, since this the most widely used primitive in all 3D applications.

UI

We added a FPS counter as an indication of what the performance is of the ray tracer. We also added information about the current resolution, d value, camera position and view direction.

OpenMP - For loop parallelization

With OpenMP we did a very basic and general parallelization of our most used loops in the program. (locations : Camera::GenerateRays, Camera::UpdateRays, Renderer::Render)

Key Commands Available

LeftArrow & RightArrow keys -> Rotate camera left & right

UpArrow & DownArrow keys -> Rotate camera up&down
LeftControl & RightControl keys -> Rotate camera with a roll
W & S keys -> Translate camera forward & backward over world axis
A & D keys -> Translate camera right & left over world axis
LeftShift & RightShift keys -> Translate camera up & down over world axis

Work Division
Basic Architecture - Dustin & Luuk
Primitives - Luuk
Ray - Luuk
Renderer - Dustin & Luuk
Camera - Dustin
UI - Dustin & Luuk
Basic Controls - Dustin & Luuk
Report - Dustin & Luuk
Material - Luuk