Aalto University
ELEC-A7151 - Object-oriented programming with C++

Project plan:
Path tracer

Eemeli Forsbom, 1012259
Aleksi Kalsta, 1018350
Eetu Reijonen, 908911
Henri-Mikael Suominen, 100547997

October 30, 2023

Contents

1 Scope of the work **1**

2 Software structure **2**

3 External libraries **4**

4 Division of work and responsibilities **4**

5 Schedule and milestones **5**

1 Scope of the work

Our mission with this project is to create a basic path tracing tool that takes in a file containing the needed information to render a three-dimensional scene and outputs a PNG or other visual file format of this scene to be displayed to the user. The tool will include at least the following features:

- Shadows
- Reflections
- Spheres and triangles as geometry in the scene (these objects can also be light sources)
- Support for .obj files (triangle meshes)
- Environment Lighting
- Glossy and diffuse materials
- At least one camera with modifiable parameters like resolution, depth of field, field of view, and position.
- A file loader that gets all information of a scene from a file (possibly YAML) and creates a scene from all the retrieved information.
- Monte Carlo integration

These features will be implemented before any additional features are to be implemented.

Also, we plan to implement the following extra features if time allows:

- A graphical interface that allows the user to change different parameters regarding the scene, like the depth of field or the number of light bounces, and show a real-time preview of an image with these parameters
- Parallelized rendering on all CPU cores and/or GPU and possibly other optimisation related improvements
- Paths for object movement (animation)
- Specular materials

The basic form of the program should be used as follows. First, the user must input the address of a scene file to be rendered. The scene file has to be edited separately (in a text editor) to create the desired scene. Parameters of the camera and other objects may be able to be changed in a graphical interface in the final build but this is not yet certain. After this, the user can run the program and wait from seconds to hours, depending on the desired quality, for the program to render the scene. When the program has finished

rendering, the final image of the scene will be shown in a window. The program also outputs an image file for the user to save for later use.

The logical structure of the program works in its basic form as follows. When the program starts running, an Interface-object is created. This object should then create the FileLoader-object which asks the user for a scene file to be read and creates a Scene-object based on the file. This Scene-object is given to the Interface-object that forwards it to the Renderer-object. The Renderer-object then renders an image matrix based on the properties of the scene and gives this for the interface to display. It also creates an image file for the user of the rendered scene.

2 Software structure

The core functionality of our program relies on accurately modeling the interaction between various types of light rays and different materials and examining how these materials influence the properties of light. To achieve this, it is convenient to incorporate a few commonly used structures into our program. We plan to use at least the data structures shown in Figure 1 in our program.

Figure 1: Required structures

Material structure captures the most basic properties that are needed to model essential materials and Color structure is used to represent RGB-values. We will utilize a Triangle structure to store information belonging to TriangleMesh objects that will be introduced subsequently. A ray is a simple structure designed to store information regarding the origin and direction of light, and Hit structure may be used to analyze the hitting points of the rays. The Camera structure is designed to encapsulate essential parameters required for generating an image of the scene.

A preliminary plan for the program class structure is presented in Figure 2. Note that, since we are in the initial stages of familiarizing ourselves with the topic, it is highly probable that both the plan and class structure may undergo changes later. Additionally, at this moment, the focus is on capturing the fundamental idea of the program structure and surely some of the important functions and attributes aren't present.

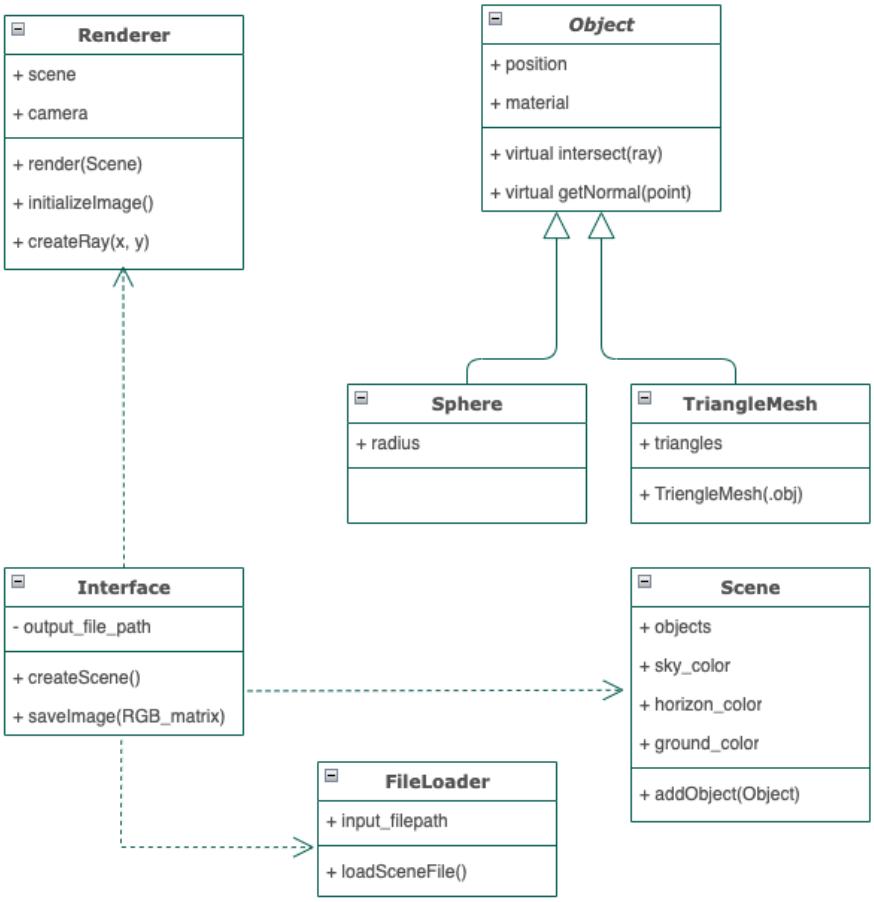

Figure 2: Class structure

Starting from the Interface class, its main purpose is to take necessary inputs from the user, create a Renderer, Scene, and FileLoader objects, and save the generated image when the Renderer is ready. We wanted to keep the interface as a separate class so that it is easier to implement some kind of GUI if we later decide to do so. In the FileLoader, we will implement the basic functionality required to load scene information from a file. Scene class captures all the information about the scene and includes all the functions that we need to adjust any scene-related things. As the name suggests,

Renderer is the class that will perform the actual rendering and include all the helper functions that are needed in the rendering process. Object represents an abstract class, which will have all the common attributes and methods needed for all the objects. Sphere and TriangleMesh are inherited classes from the abstract Object class.

3 External libraries

Implementation of the core features discussed in the previous chapters benefits from the use of external libraries. We chose libraries that make it easier and faster to perform linear algebra calculations, output image files, and input scene files. For the more advanced features, such as a graphical user interface and GPU acceleration, further libraries are required.

In its most basic form, a path tracer is a program performing calculations to simulate the behavior of light with different materials. These calculations can be efficiently represented with linear algebra. For this reason, we chose the recommended Eigen library [GJ+10]. It contains the basic data structures, such as a three-dimensional vector, as well as the basic functions, such as dot product and cross-product, to perform essential linear algebra calculations.

The second crucial feature of the program is to be able to output the result of the light calculations in a human-readable format. In concrete terms, one has to create an image file from a matrix of RGB values. For this purpose, we plan to use the recommended SFML library [SFM23].

For the final basic feature, scene file input, a YAML, XML, or JSON parsing library is required, since we are going to use one of these file formats to represent the scenes. yaml-cpp [jbe23] is a possible option for parsing YAML-files, RapidXML [Fe-23] or pugixml [zeu23] for XML-files, and json [Nie23] or RapidJSON [Ten23] for JSON-files. The final choice will depend on the file format we choose and the specific requirements of the library which will become clear as the project progresses.

The implementation of a graphical user interface (GUI) for our program is not yet planned in detail at this stage of the project. However, a possible option for creating the GUI is OpenGL. According to our current understanding, this library can also be used for implementing GPU acceleration.

4 Division of work and responsibilities

We agreed to divide the work and responsibilities of the project as equally as possible. Of course, at this stage of the project, it is quite hard to divide the whole project into parts but we agreed on some initial features each one of us would get started on. Eetu agreed to start on the Renderer class as he

already implemented a small demo. Eemeli will start working on the Scene and Object classes. Henri is going to start developing the Fileloader class and he is also going to investigate possible file formats for our scenes. Lastly, Aleksi will work on the Interface class and he will also focus on the build of the project.

As we finish implementing these core classes we will then divide the more advanced features depending on who finishes their parts first and also according to personal preference. We also decided on a weekly meeting time on Mondays from 14-16. These meetings will primarily be in person but if we decide we do not have much to discuss some particular week we can also arrange the meeting remotely. Now that we have the initial division of work decided it will then be easy to divide the work in the future during these meetings.

5 Schedule and milestones

We agreed on a rough schedule to get the basic features done at the latest on 13.11. and then we will move on to the more advanced features. We will try to develop as many advanced features as we have time for but the first things we are going to start on are the triangle meshes and parallelization. Then we will implement more advanced materials with different properties like refraction and subsurface scattering. We will also try to optimize our path tracer as well as possible along the way. The last feature we will implement if we have time, is a GUI for our program.

This is the rough schedule we decided on during the first two meetings and of course, we will set discrete deadlines as the project goes along during future meetings. We can also adjust the schedule in the future depending on how fast or easily we can implement certain features.

References

[GJ+10] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*. 2010. URL: `http://eigen.tuxfamily.org`.

[Fe-23] Fe-Bell. *RapidXML*. 2023. URL: `https://github.com/Fe-Bell/RapidXML`.

[jbe23] jbeder. *yaml-cpp*. 2023. URL: `https://github.com/jbeder/yaml-cpp`.

[Nie23] Niels Lohmann. *json*. 2023. URL: `https://github.com/nlohmann/json`.

[SFM23] SFML. *SFML*. 2023. URL: `https://www.sfml-dev.org`.

[Ten23] Tencent. *RapidJSON*. 2023. URL: `https://github.com/Tencent/rapidjson/`.

[zeu23] zeux. *pugixml*. 2023. URL: https://github.com/zeux/pugixml.