ELEC-A7151 Project

Path Tracing

Group: cpp-pt-4
Dang Ly
Teodors Kerimovs
Verneri Hakkarainen
Lauri Wilppu

Advisor: Markus Säynevirta

I. Scope of the work

Path tracing is a technique that performs physically accurate simulation of light in order to generate ultra-realistic images of virtual 3D environments. It is the fundamental method behind the creation of modern graphics in applications like architectural visualizations and CGI in movies, and laying the foundation for many other advanced techniques that are used in real-time rendering. The core idea behind path tracing is rather straightforward: you have light sources, objects that interact with light, and a camera. Instead of following the real-world light path, which is from light source to camera, path tracing casts rays in the reverse direction to reduce computation complexity. If a ray reaches a light source after some iteration, the color of the ray and the corresponding pixel can be established based on the properties of the light and the objects it encounters. Because of this simple and unbiased characteristic, path tracing is used to generate the reference image that researchers often use to compare their implementation with.

A typical session can start by loading a scene file from disk or creating a blank session. The program will display the raster preview of the current world. Users can move the camera around, add and change objects' properties, or change the camera settings. After users are satisfied with the current setup, they can configure and start the path tracing process. Ideally, the path tracing will run with GPU acceleration and the program will show the current progress by displaying the rendered section of the image. After the rendering process is completed, users can redo everything if the result is not satisfying or save the result as an image file. The modified scene can also be saved for further usage.

Below is the list of planned features in the project:

1. Basic feature list
    - File IO: read/write scene files, generate common image file formats
    - Various simple light sources, ambience light, volumetric light.
    - Common simple geometry: sphere, plane, triangle mesh, boolean geometry
    - Various materials characteristic
    - Custom camera settings: position, direction, FOV, focus, etc.
2. Additional feature list
    - Monte Carlo integration
    - Area light
    - Support for common file format
    - Advanced materials: texture, normal maps, metallicity
3. Advanced feature list
    - Accelerated geometric data structure (BVH)
    - Importance sampling
    - GPU acceleration
    - Advanced material: dielectric, fog, subsurface scattering
    - Real-time rendering preview

II. Program structure

Our program consists of 4 main modules: the GUI, data structure, path tracer, and helper library. The GUI is expected to have previewing, object editing, and various settings functionalities. This module has 2 mains component: class *Display* would display the raster preview of our model before performing path tracing and support navigation of the camera; class *GUIElement* and its realizations let user changing configuration, parameters and performing various tasks. *Dialog*, a child of *GUIElement*, can be used to display information and error notifications. We designed this diagram with OpenGL and Qt in mind; however, that can be changed in the future if we found other libraries that are simpler and fit our planned functionalities better.

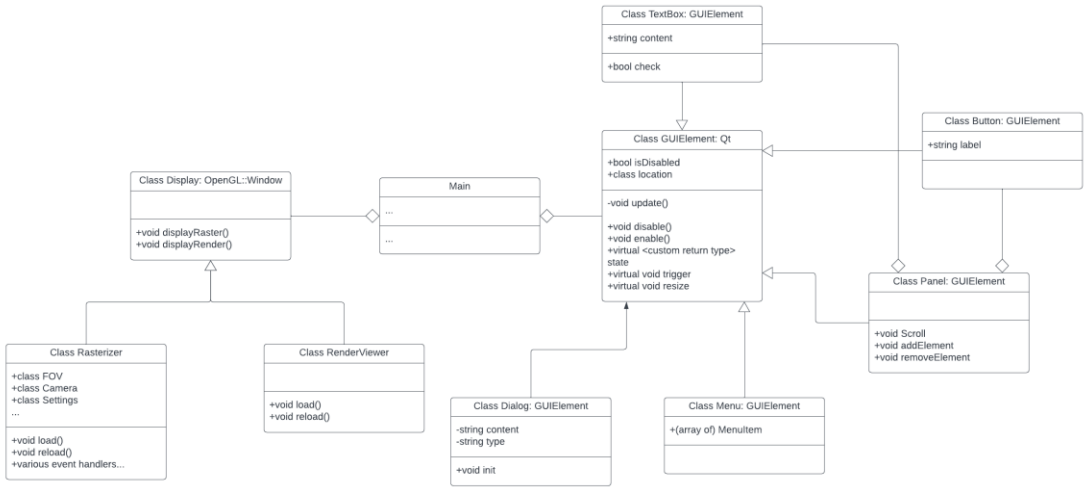

Figure 1: GUI module classes diagram

The data structure module stores all information about the world's objects and related objects' functionalities. The GUI's *Rasterizer* uses this data to render preview before tracing, and this data can also be edited from the GUI. The path tracer, obviously, relies on this module to complete its tasks.

The main structure in this module is class *Object*, representing all possible 3D objects in the program. Each object has 2 components: *Geometry* and *Material*, which are in their respective classes. *Geometry* has 2 virtual functions that all its children must implement: "hit" function, to check if a given ray intersects with object; and "next" function, return the next ray to continue the tracing algorithm. *Geometry* will have various children; we only mention a few sample classes here. *BooleanGeometry* is a special class based on other primitive geometry, but supports set operations: union, intersection, set difference, etc. *Material* has 1 virtual function that returns color after a ray hits the surface. This class will also have a vast number of children. *RenderBuffer* will contain the current tracing progress that the GUI module can display. The use of inheritance will make the module very expandable with new types of shapes and material.

Since this module is performance critical, additional structures may be implemented, for example BVH.

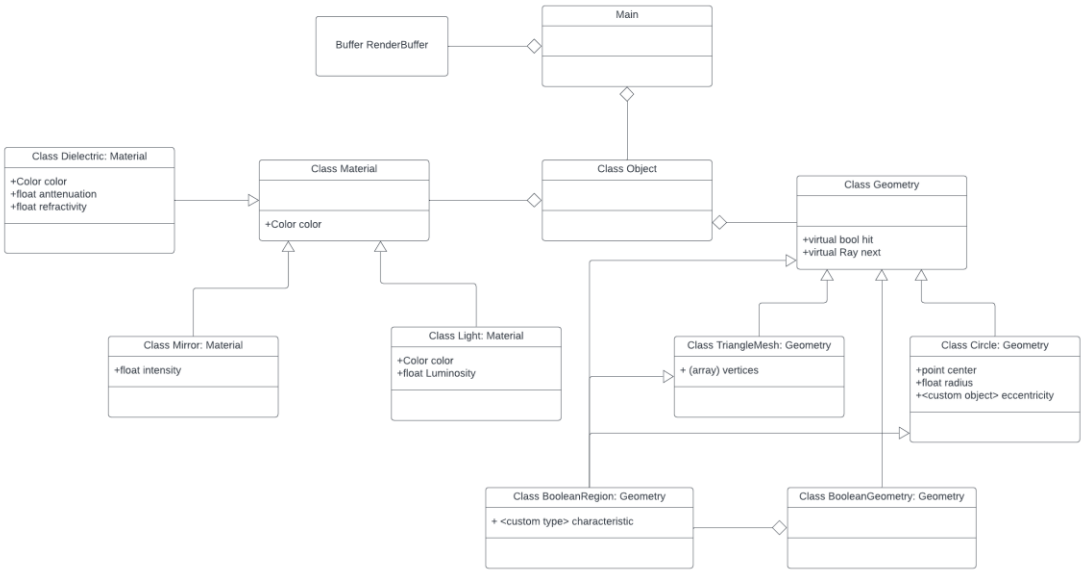

Figure 2: Data structure module class diagram

Path tracer, the most important part of this program, is planned to follow the functional programming paradigm. The reason is that this module does not quite follow the inheritance expansion; each new functionality will be independently created and added. Also, since this module is performance critical, too complex OOP structure may hurt rendering time severely, especially if many virtual classes and functions are used.

Helper library module implements other structures that are necessary for the program. It has 2 main components: rendering-related objects and functions (*vector*, *color*, *camera*, *FOV*, etc.) and other helper functions: file IO, image encoding, error handling, etc.

III. External libraries and technologies

Below are some libraries and technologies that we plan to use in the software. CUDA, OpenMP, OpenGL and SFML have already been supported on Aalto Linux machine. Others will be statically linked to our program. This list may change in the future according to our needs.

Library	**Category**	**Documentation**
SFML	GUI	https://www.sfml-dev.org
OpenGL	Graphics/GUI	https://www.opengl.org
Eigen	Linear algebra	https://eigen.tuxfamily.org
stb	Image processing	https://github.com/nothings/stb
CUDA	GPU programming	https://developer.nvidia.com/cuda-toolkit
Sample code	Sample code	https://github.com/rogerallen/raytracinginoneweekendincuda
OpenMP	Parallel programming	https://www.openmp.org
GoogleTest	Unit testing framework	https://github.com/google/googletest

IV. Division of workload

In the first phase, everyone is going to work on the code base, with the goals being creating a runnable path tracing program. Each person will implement a subset of path tracing features presented in the book "Ray Tracing in a Weekend" by Peter Shirley, with necessary code documenting and unit testing. The detailed plan is presented in the following section.

Later there will be larger blocks of parallel tasks and we will specify our roles – two main branches are advanced path tracing features and GUI development.

V. Planned schedule

Below is the plan of the first phase of the project. Each work requires implementation, code documentation with doxygen, and unit testing.

Date	Person	Work	Note
1.11	Teodors	Basic vector and ray implementation	Follow chapter 2 and 3 in the book
3.11		Project plan review	Receive feedback from TA about our project plan
3.11	Teodors	Image exporter	Export frame buffer to a common image format: JPEG, PNG, etc.
3.11	Dang	Simple path tracing	Implement spherical geometry, shading, and anti-aliasing
6.11	Verneri	Materials	Implement various types of materials, based on chapter 7-9
6.11	Lauri	Camera and focus	Implement camera positioning and defocus blur
17.11	To be announced	Monte Carlo integration Advanced light type	
17.11	To be announced	Basic GUI functionalities Scene file reader	
20.11	Everyone	Complete basic and additional features requirements	Advanced features and milestones will be decided in the following group meeting