Aalto University

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

Path tracer, Documentation

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

December 10, 2023

Contents

1 Overview

Our software is a basic path tracing tool that takes in a file containing the needed information to render a three-dimensional scene in YAML format and outputs a PNG image file of this scene to be displayed to the user. The software is used through a simple graphical user interface or via the command line. The tool includes the following features:

- Input of a scene file in YAML format.
  - Informative custom exceptions for invalid input parameters
- Output of a PNG image file
- Different geometries:
  - Sphere
  - Box (3-dimensional)
  - Rectangle (2-dimensional)
  - Trianglemesh (loaded from .obj file)
- Area lights (any geometry above)
- Different materials:
  - Diffuse
  - Reflective
  - Clear coat
  - Refractive
- Shadows
- Reflections
- Any number of objects in the scene
- Modifiable camera parameters:
  - Position
  - Resolution
  - Field of view
  - Depth of field (for simulating defocus/lens aperture)
  - Focus distance
- Example scenes that demonstrate the features

- Monte Carlo integration (shooting multiple randomized rays through each pixel)
- Parallelized rendering on all CPU cores
- Bounding volume hierarchy geometry acceleration data structure
- GUI with real-time preview rendering

2 Software structure

The basic classes and structures used in the program can be seen in Figure 1.

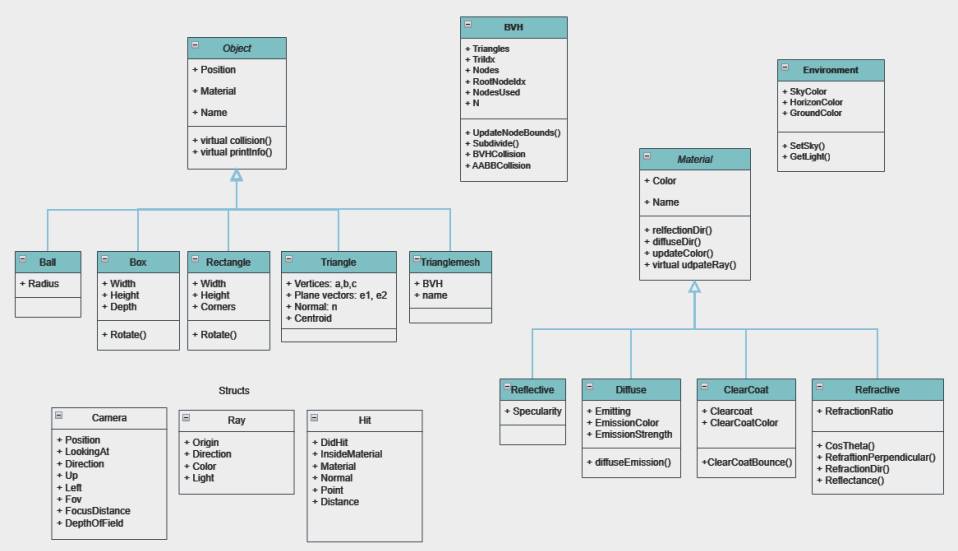

Figure 1: Basic classes and structures

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. The Environment class includes the necessary information about the background of the scene to be rendered.

The Object class is an abstract class describing the basic properties of our objects. Different kinds of objects are implemented as inherited classes of Object which include their own version of the virtual collision() and printInfo() functions. The five different objects we have implemented are Ball, Box, Rectangle, Triangle and Trianglemesh. The Trianglemesh utilizes a Bounding volume hierarchy geometry acceleration class (BVH) to store a

collection of Triangles for efficient collision detection. An instance of the Trianglemesh class is created from a .obj file.

The Material class is also an abstract class that describes how Rays will relfect, collect light and change color when colliding with a material. We have implemented four different materials: Relfective, Diffusive, ClearCoat and Refractive as inherited classes of Material. Relfective is a mirror-like material, Diffusive is a smooth material, ClearCoat is a smooth material with mirror-like finish, and Refractive is a transparent material.

These are the basic classes and structures that are needed to implement our path tracing software. They are then used by the classes seen in Figure 2 to implement a functioning program.

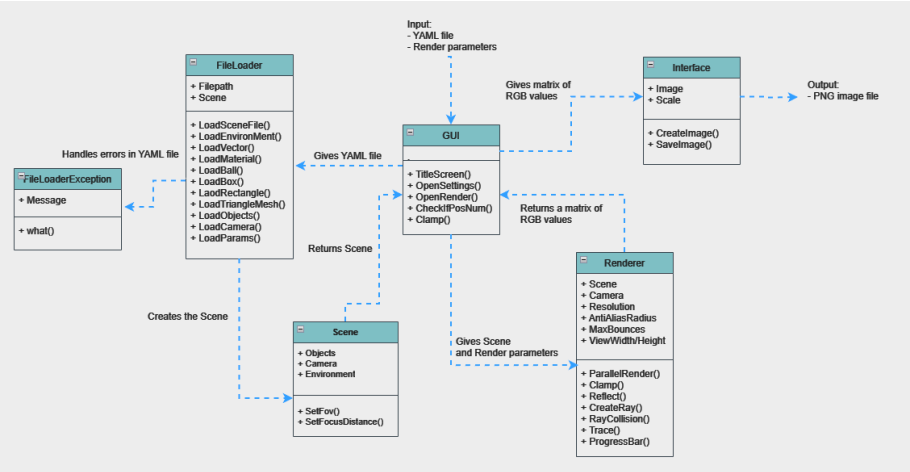

Figure 2: Program structure

When the program is started it takes the path of a YAML file containing the information of the scene, and the render parameters, such as resolution and number of samples, as inputs either inside a GUI or via command line arguments. Then the FileLoader class reads the YAML file, checking for any exceptions i.e. input errors in the file, and creates an instance of the Scene class. This class includes a list of all the objects in the scene, the camera and the environment, which were all extracted from the YAML file. The scene is then returned to the GUI which then calls the Renderer class with the given render parameters and the created scene. The Renderer object then does the actual path tracing with the given parameters and returns a matrix containing RBG values of each pixel in the image. Finally this matrix is given to the Interface class which then creates a PNG image from the matrix and saves it for the user.

We have used the following external libraries to implement the program:

- Eigen (for linear algebra)
- tiny_obj_loader (for loading .obj files)
- SFML (for GUI and PNG generation)
- YAML-cpp (for reading YAML files)
- OpenMP (for CPU parallelization)

Eigen and tiny_obj_loader are both simple header file libraries so their files are included directly in the /libs folder. For OpenMP, the user needs to ensure that they have a compiler supporting OpenMP, i.e., the environment variable OpenMP_ROOT is correctly set. GNN is an example of a compiler supporting OpenMP and for Mac libomp can be installed with Homebrew to enable OpenMP for clang. The other libraries are fetched by CMake from their Github repositories. Therefore, there should be no need for the user to install any of these libraries separately.

3 Instructions for building and using the software

3.1 Building the software

As mentioned in the previous section, we have used CMake to generate Makefile that can be easily used to compile our program. All you should need is CMake with version 3.19 or newer and OpenMP, such that CMake is able to find it. In order to get our program running, first step is to download the project from GitLab. You can either download the .zip file from our GitLab repository: PathTracer or simply clone the repository with "git clone". After getting the repository to your computer, open the project root where the CMakeLists.txt is located and enter the command

```
cmake -S . -B build/ -DCMAKE_BUILD_TYPE=Release
```

which will fetch needed dependencies from github, create a build directory and generate Makefile with other necessary files for building the project. Note that downloading the dependencies might take a while. CMake will also give you a Deprecation Warning for yaml-cpp library, since it only requires version 3.4 in its own CMakeLists.txt, but this should not cause any problems in current CMake versions. Next you can proceed to the build directory and build the project using make as follows:

```
cd build && make
```

This will build the executable ./PathTracer and unit tests simualtaneusly. After building the project there is two different ways to use the program: via command line or a graphical user interface.

3.2 Using the software

If the user wishes to use the GUI, simply typing ./PathTracer in a terminal in the build folder is sufficient. The program first asks to enter a file name for the YAML file containing the scene information. We have included several example scenes in the folder ../scenes, which can be used to test the program. After obtaining a valid scene file, the program then asks for the rendering parameters the user wants to use. These parameters are the horizontal resolution ResX, vertical resolution ResY, field of view FOV (in degrees), depth of field, focus distance and number of samples used. The only required inputs are the resolutions, amount of light bounces and the number of samples as all the other parameters have default values set for them. There is a preview button at the bottom that can be used to test the given parameters with a fast render that is displayed in the GUI. When the user wants to start the actual rendering they can just press Enter and the program starts. The rendered image will be shown and updated after each sample. When the program is finished it shows the final image in the GUI and also saves it to the build folder. It's recommended that the user unselects any currently selected input space, by pressing the escape key, before pressing enter and starting the render.

The other option is to use the command line with the required parameters given as arguments after ./PathTracer. The required arguments are in order: string for the file path, horizontal resolution, vertical resolution, number of samples, number of bounces and image filename (with path) as a string. This will print information about the scene and a progress bar of the rendering process to the command line as shown in Figure 3. When the rendering is finished, the image will be saved to the build folder (or to relative path from build folder given as an input). Here are some example commands for rendering our example scenes.

./PathTracer ../scenes/bigBalls.yaml 1280 720 10 20 bigBalls.png

./PathTracer ../scenes/glassBalls.yaml 1920 1080 10 20 glassBalls.png

./PathTracer ../scenes/legacyBalls.yaml 1500 1200 20 10 legacyBalls.png

./PathTracer ../scenes/mirrorRoom.yaml 1280 720 15 30 mirrorRoom.png

./PathTracer ../scenes/objectScene.yaml 2160 2160 20 10 objectScene.png

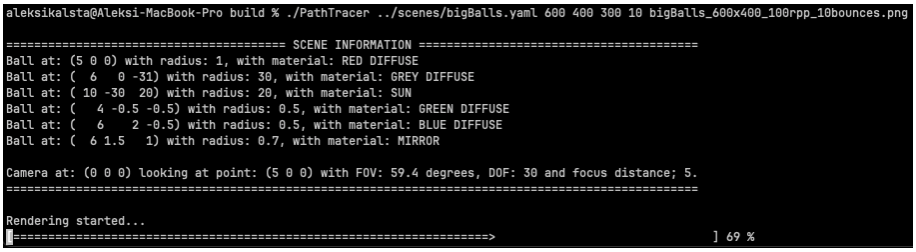

Figure 3: Example output about rendering a scene from the command line

3.3 Scene files

The YAML files containing all the information of the scenes have three different nodes. The first one is the camera node, which has the following format:

```
Camera:
  Position:
    - 0
    - 0
    - 0
  LookingAt:
    - 5
    - 0
    - 0
  Fov: 0.33
  DepthOfField: 30
  Angle: 0
  FocusDistance: 5
```

where the three position values are the x, y, and z coordinates of the camera as floats, the LookingAt values are the x, y, and z coordinates of the point the camera is looking at as floats, Fov is the field of view as a percentage of π radians, DepthOfField is the strength of the defocus effect (aperture size in a real camera), Angle is the orientation of the camera about the image plane normal in degrees and FocusDistance is the focus distance of the camera.

The second node is the environment node:

```
Environment:
  SkyColor:
    - 0.2
    - 0.5
    - 1.0
  HorizonColor:
    - 0.7
```

```
    - 0.8
    - 0.8
  GroundColor:
    - 0.1
    - 0.1
    - 0.1
```

which specifies the colors of the environment as RGB values. The third and final node is the Objects node which contains all the objects in the scene and has the following format:

```
Objects:
  - Object:
      Type: Ball
      Position:
        - 5
        - 0
        - 0
      Radius: 1
      Material:
        Type: Diffuse
        Color:
          - 1
          - 0
          - 0
        Name: RED DIFFUSE
  - Object:
      Type: TriangleMesh
      Filepath: ../objects/knight.obj
      Scale: 1
      Position:
        - 9
        - 1.5
        - -1
      Rotation:
        - 0
        - 0
        - 55
      Material:
        Type: ClearCoat
        Specularity: 0.8
        ClearCoat: 0.2
        Color:
          - 0.39
          - 0.19
```

```
    - 0
Name: LACQUERED WOOD
```

where for each object the Type (Ball, Box, Rectangle or TriangleMesh), Position, and Material (Diffuse, Reflective, ClearCoat or Refractive) must be specified. In addition, different types of objects have the following parameters that need to be specified:

- Ball: Radius (float)
- Box: Width (float), Height (float), Depth (float), Rotation
- Rectangle: Width, Height, Rotation
- Trianglemesh: Filepath, Scale, Rotation

The Rotation node contains three different values which are the x-, y-, and z-axis rotations of the object in degrees. The Filepath for the object is a string to the file location of the .obj file from the build folder. The Scale is a float that determines how the size of the Trianglemesh is scaled from the .obj file.

All the materials needs to have color specified as an RGB value, but otherwise different materials have a little bit different parameters. Diffuse materials can work as a light source by providing it an emission strength and emission color. By default diffuse materials do not emit any light. Reflective material can be given a specularity parameter, which specifies how good the reflection will be. Basically value 1 means a perfect mirror and value 0 looks like a diffuse material. Clear coat material can be given a clear coat strength, clear coat color and specularity, where clear coat strength represents a probability of clear coat bounce, clear coat color specifies a color of these bounces and specularity works exactly like in reflective material. In addition to color, refractive material needs a refraction ratio, which in this case means the ratio from vacuum to material, so for glass this is approximately 0.7 and for diamond 0.42. There is a lot of different examples in the example scenes on how to define different kind of materials, so easiest way to get started is to look models from there.

To modify the example scenes the user can simply change the specific values in the given YAML files. To create new scenes the user must implement the three nodes described above in a new YAML file.

4 Testing

Most of the time when implementing new features, the testing had to be done by rendering some example images to see the results of the program. For example, when implementing the properties of refractive material it is really

hard to evaluate the correctness of the program any other way than inspecting the output image since the interactions between the rays and objects are complex and non-deterministic. Of course, after getting the properties to work as desired it is possible to create some tests to evaluate that the program works similarly in the future after possible changes. Furthermore, some of the properties and methods of the program can be tested by simple unit tests, which is exactly what we have done.

For unit tests, we have used the Google Test open-source testing library. Google test offers different kinds of macros to enable easy testing for different kinds of values. In addition, Google Test implements a "ready to go" main program to run all the defined tests and offers different kinds of parameter choices for more sophisticated testing. In practice, we have included Google Test to our main CMakeLists.txt, such that it fetches the library from GitHub as shown in figure 4 and builds own executable for the test program.

```
31  # Unit tests
32  FetchContent_Declare(
33    googletest
34    GIT_REPOSITORY https://github.com/google/googletest.git
35    GIT_TAG        release-1.8.0
36  )
37  # For Windows: Prevent overriding the parent project's compiler/linker settings
38  set(gtest_force_shared_crt ON CACHE BOOL "" FORCE)
39  FetchContent_MakeAvailable(googletest)
40  set(TESTS unittests)
41  enable_testing()
42  add_executable(${TESTS} tests/AllTests.cc)
43  target_link_libraries(${TESTS} PRIVATE gtest_main)
44  target_link_libraries(${TESTS} PRIVATE sfml-graphics)
45  target_link_libraries(${TESTS} PRIVATE yaml-cpp)
46  include(GoogleTest)
47  gtest_discover_tests(${TESTS})
```

Figure 4: Google Test library in CMakeLists.txt

Unit tests are implemented to own files for each class and included in one AllTests.cc file from where the actual test program is built. In this way, the tests stay organized and it is easy to add more tests for new classes in the future. After building the project, unit tests can be run by simply going to the build directory and entering the command "ctest" in the command line, which runs all the defined tests. If some of the tests fail, it is possible to get additional information about the reason by providing command line arguments "–rerun-failed" and "–output-on-failure" together with the command "ctest". All the tests are categorized by class, so if you wish to run unit tests for Environment class, you can simply enter the command "ctest -R ENVIRONMENT" and the output looks something like in figure 5.

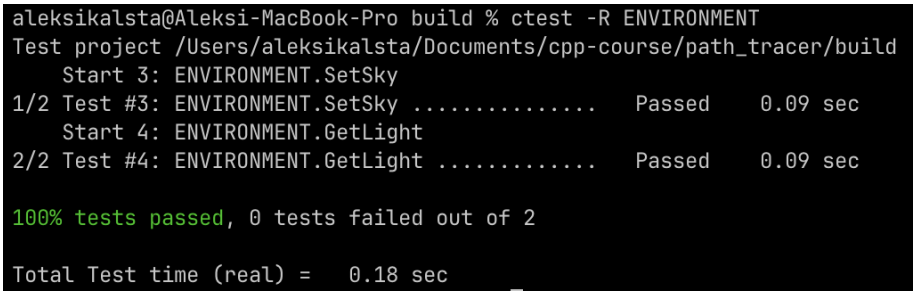

Figure 5: Unit test output for Environment class

In general, we got the tests we implemented to work well and we got a good idea of how the testing could and should be improved especially in larger software projects. Considering this project, it was quite challenging to create good unit tests that serve their purpose. That is because during this short period, we were learning a lot about path tracing and after every new idea, when we made significant modifications to our program, we also had to rewrite many of the unit tests. Nevertheless, we can see the importance of testing for the future.

4.1 Bugs

During development, we were able to catch and fix most bugs in our code, often immediately after making some changes and noticing the rendered scenes did not look right, seeing failed unit tests or failed building, etc. However, there were some long-standing issues, one of which we were not able to fully solve during this short project. Firstly, the camera system did not work properly regarding the defocus effect and the position and the orientation of the image plane. These problems were due to some incorrect linear algebra calculations in the Renderer code. They were only observed once we started to experiment with different camera positions and defocus parameters towards the end of the project. Fortunately, these bugs were quite easy to fix after understanding them. The camera system saw a refactoring during the last weeks of development and now works robustly with easily changeable camera parameters.

The second major bug is related to the rendering itself. After the CPU parallelization was added, over time we started noticing that on some renders black or colored lines or blocks started appearing. An example of this is presented in Figure 6.This problem is especially pronounced when using the GUI to render images. With the GUI, the lines appear almost 50% of the time, making the images unusable. Conversely, when using the program through the command line, the lines appear very rarely, maybe in 1% of the rendered images. Despite our best efforts, we have not been able to definitively track down the source but the culprit is very likely CPU parallelization. The lines

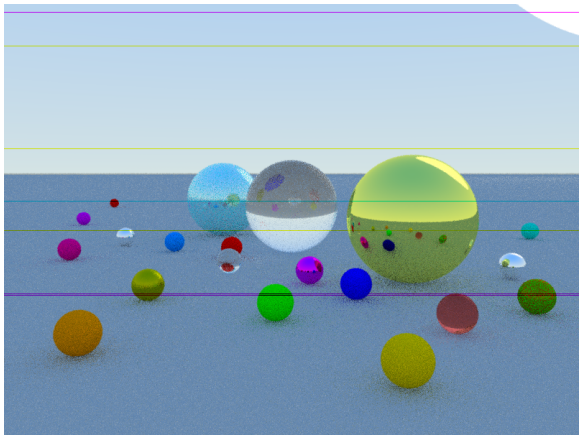

Figure 6: Colored lines appearing in an image rendered using the GUI

might be due to synchronization issues between the threads or data races. Concurrent use of the SFML library (in the GUI) seems to increase the likelihood for the lines drastically. Due to poor reproducibility of the bug and the high-level abstraction of SFML and OpenMP, troubleshooting has been very difficult. With more time, this bug should be able to be fixed. One would probably learn a lot about the inner workings of OpenMP along the way, too. However, even the troubleshooting part has been very insightful and a good lesson in seeing how challenging bugs can be.

5 Work log

Table 1 contains how many hours were used on the project each week by each group member.

Table 1: Hours used on the project by each group member

	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7
Aleksi	13	12	3	9	13	12	16
Eemeli	10	6	8	12	16	8	8
Eetu	15	8	10	12	8	6	7
Henri	8	12	14	6	9	5	5

5.1 Week 0

Everybody: familiarizing with the topic, and project plan formulation during the meeting

Eetu: Implementation of a Julia demo

Aleksi: Established the project repository to GitLab, Studied Makefiles and CMake, and started to search for information about graphics libraries.

Henri: Started looking at a suitable file format to store information of scenes and finding a library that reads chosen format

5.2 Week 1

Everybody: Choosing and getting familiar with external libraries, writing the project plan

Eetu: Finished the Julia demo.

Aleksi: Got the SFML library to work with CMake and got the CMake file working seamlessly

5.3 Week 2

Eetu: Renderer and partial Object, Ball and Scene implementations

Aleksi: Interface class implementation and bug fixes, fighting with cross platform compatibility.

Henri: Partial FileLoader implementation and creating template for .yaml files containing scenes

Eemeli: Research on triangle collisions and .obj files

5.4 Week 3

Eetu: Renderer improvements, RandomGenerator, Environment, print methods

Aleksi: Merged different branches.

Henri: First iteration of working FileLoader

Eemeli: Research on loading .obj files and started working on triangles

5.5 Week 4

Eetu: OpenMP implementation, cmake improvements - loading from GitHub

Aleksi: Searching information about unit testing and established unit testing system with Google Test.

Henri: Updated FileLoader and .yaml scene files to work with renderer improvements

Eemeli: Implemented Triangle and TriangleMesh classes

5.6 Week 5

Eetu: Implemented box object and clearcoat material.

Aleksi: FileLoader improvements (unit tests, custom exceptions etc), Refactored multiple parts of the program to make memory management better.

Henri: Implemented a first iteration of working GUI

Eemeli: Implemented bounding volume hierarchy for trianglemeshes and triangle collision optimization

5.7 Week 6

Eetu: Implemented rectangle class and created example scenes. Implemented command line argument interface for the program.

Aleksi: Refactored materials to own classes and added refractive material as a new feature. Other necessary modifications to the program accordingly.

Henri: Added more parameters for the user to change in the GUI and made a sample by sample view of the rendered image

Eemeli: Implemented LoadObject method to read .obj files from YAML files and rotation of Trianglemeshes. Started to work on documentation.

5.8 Week 7

Eetu: Fixed bugs relating to camera defocus effect and designed example scenes. Made command line interface more robust.

Aleksi: Fixed/modified camera movement system, designed example scenes, wrote documentation, cleaned the source code directory, and fixed unit tests for FileLoader.

Henri: More bug fixes on GUI and looking into "colored lines" -bug but to no avail

Eemeli: Worked on documentation, and created the presentation for the demo.