ELEC-A7151 - Object oriented programming with C++

Path Tracer - Project Documentation

Fall / 2018

Group 4
Aleksi Hämäläinen 425287
Jan Lundström 345448
Ville Leppälä 353854
Jesse Miettinen 350417

1. Overview

The goal of this project was to produce a software that can render a background and a set of shapes given as an input.

A very simple description of a path tracing algorithm would be that we simulate multiple (millions) of light rays. The simulation is being done backwards; from camera to the light source. We shoot multiple rays through every pixel in our resulting photo and determine the color of the pixel by averaging over all the rays color values that went through the pixels. An example result of our path tracer is shown below. It includes different materials and textures as well as a random amount of spheres.

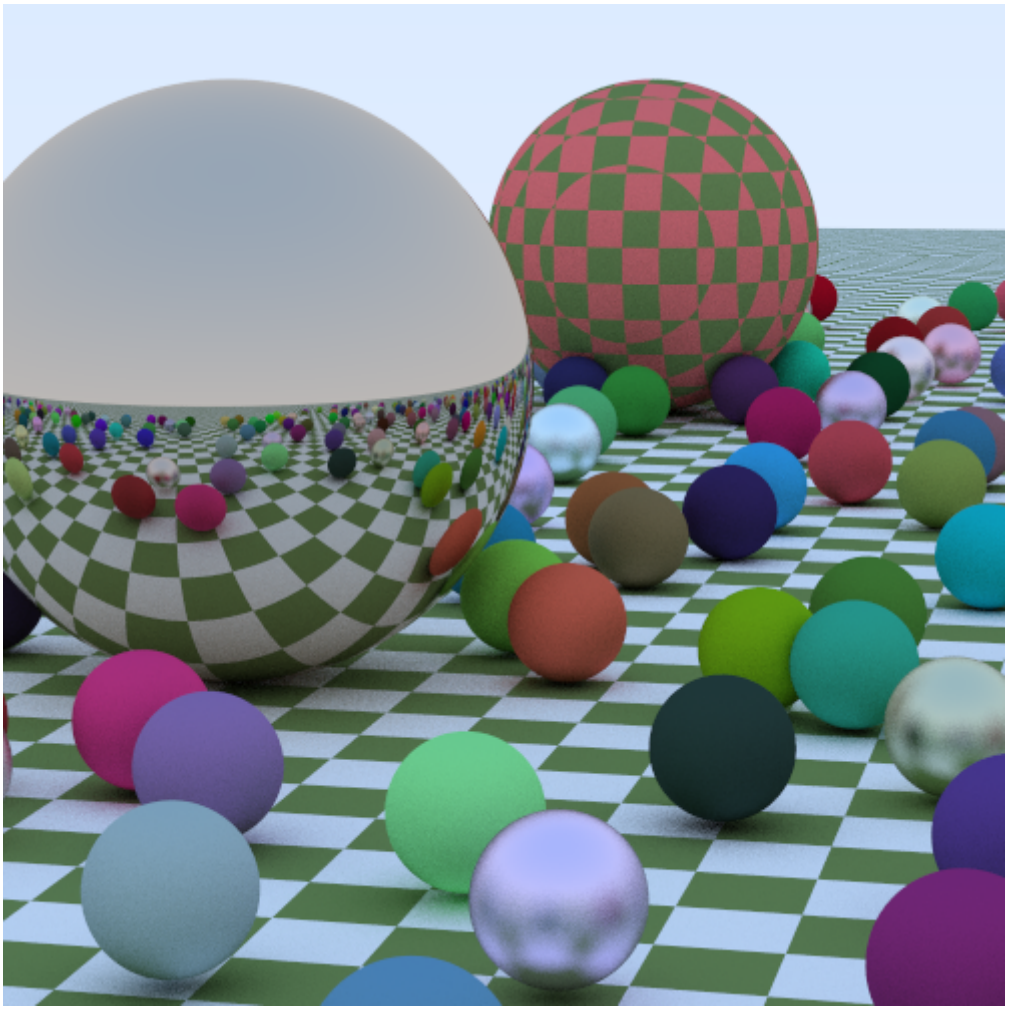

Figure 1. Output of our path tracing software

The path tracer supports physical geometries, precisely spheres, rectangles (xy, xz and yz planes) and boxes. Each object can be built with different material models like dielectrics (e.g. glass, water), solid textures (e.g. RGB colors and textures like checker) as well as metal (albedo). In addition, each object can be assigned as diffuse material meaning that it emits light to the scene. Also, a normal map is available. Images are constructed with minimal noise by anti-aliasing meaning that rays are sent multiple times through pixels and the result is averaged. The amount of anti-aliasing can be controlled by the user. The camera can be moved to all directions as well as rotated. In addition, focus distance, field of view, aspect and aperture can be modified.

Performance-wise rendering is accelerated by a bounding volume hierarchy. The software is also parallelized to enable computing with all cores of the CPU.

From the user perspective, scenes can be loaded from a file while rendered images are outputted as ready PNG files. To showcase the capabilities, example scenes are created in dark and light conditions with a random geometry and material generator.

2. Software Structure

The main structure of the program is described in figure 2.

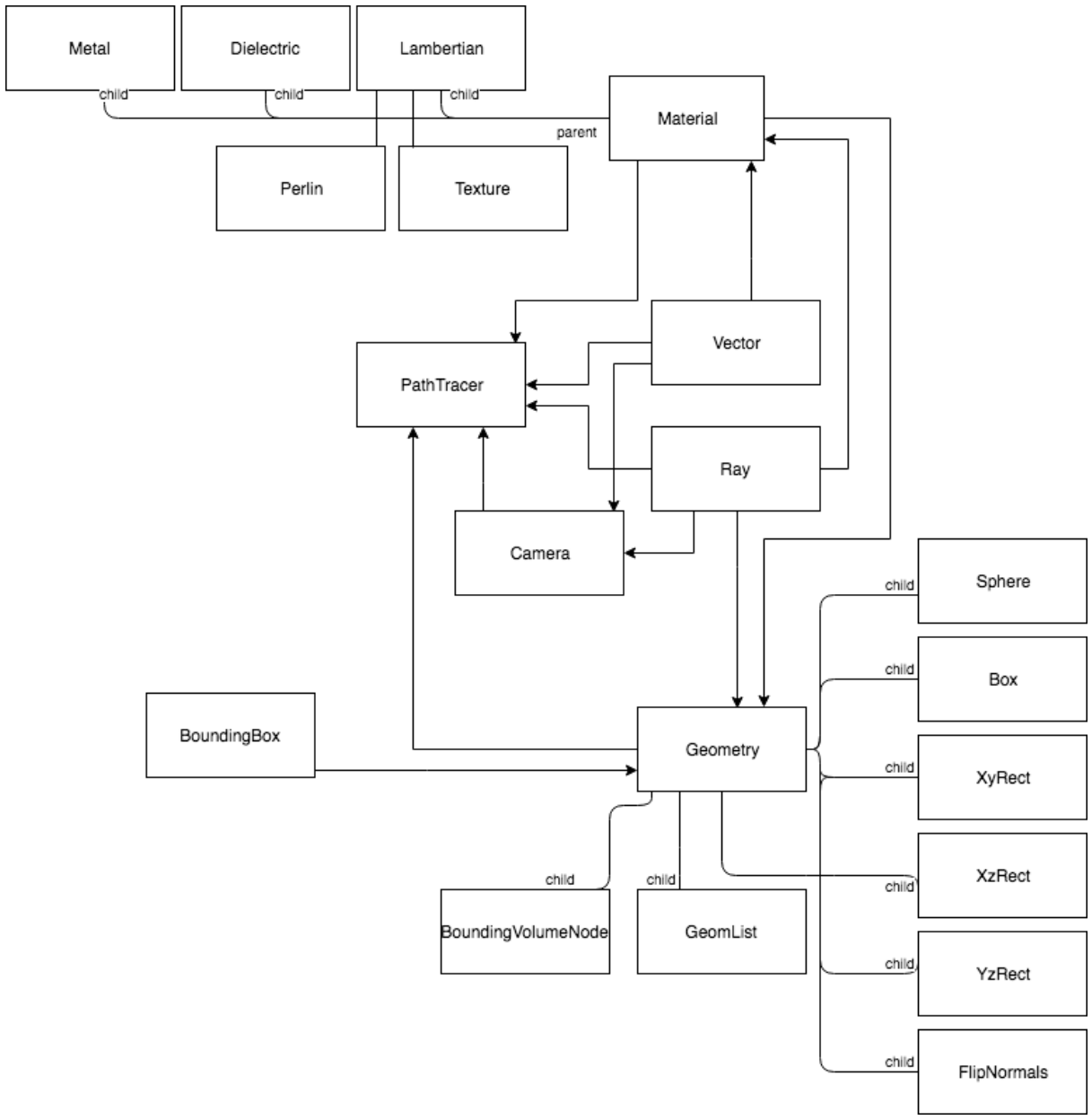

Figure 2. UML chart

A scene is created through the following steps (popularized description)

1. Create a object list for geometries
2. Create Texture object(s)
3. Create a Material object(s) and insert the texture(s) to it
4. Create desired Geometry object(s) and insert the material(s) to it
5. Create a Camera object with desired parameters
6. Create a Geomlist object and insert the object list into it
7. Render the scene
    a. The rendered loops through all pixels in the desired scene
    b. Multiple rays are “shot” randomly through each pixel [anti-aliasing]
    c. The color of each ray is determined in a Color function that checks whether there is a geometry on the specific coordinate
    d. The color of each pixel is averaged from the values obtained from the rays processed in b and c.
    e. Color values are normalized as BGR values
    f. The image is created based on the created BGR values

In more detail, geometries including their size and position, materials, textures, camera parameters, light and the amount of anti-aliasing samples can all be modified by the user. The software can be runned through a robust command line UI.

The ray-object intersection is the main time-bottleneck in a ray tracer, and the time is linear with the number of objects. It is a repeated search on the same model, so the renderer is accelerated by making it logarithmic search in the spirit of binary search. Because we are sending millions to billions of rays on the same model, we can do an analog of sorting the model and then each ray intersection can be a sublinear search. Objects are divided into subsets. Any object is in just one bounding volume, but bounding volumes can overlap. A good introduction to the subject can be found from reference [2].

Computation time decreased remarkably with parallelization and bounding volume hierarchy. However, rendering high quality images is time-consuming and for more performance, importance sampling was also implemented in an own branch. Unfortunately, due to few flaws and time limits, it was not merged to origin master.

A created scene can be saved with SaveWorld()-function. The objects and the current parameters of the camera in the scene are saved as a JSON-file to the location given in the execute program keywords arguments. SaveWorld takes care of the output stream and opening and closing the output file. SaveWorld() also calls for ToJson() function that takes care of index generation for objects in the scene and calling for each objects ToJson() functions. Each object then calls for their own materials’ ToJson-function which call their own textures ToJson function. ID and the json file are given as references in all ToJson()-function’s parameters.

Example of an object saved to JSON-format:

```
{
   "object0": {
     "material": {
       "texture": {
         "color": {
```

```
        "x": 0.8999999761581421,
        "y": 0.8999999761581421,
        "z": 0.8999999761581421
      },
      "type": "constant_texture"
    },
    "type": "lambertian"
  },
  "position": {
    "x": 70.0,
    "y": 60.0,
    "z": 400.0
  },
  "radius": 60.0,
  "type": "Sphere"
```


When LoadObjectList is called, it is given a vector of shared pointers. LoadObjectsList takes care of opening the input file and parsing the JSON-formatted text. It then calls for each object's own Loader functions that all return a shared pointer to a created object. LoadObjectsList() then pushes

The C++ standard library is used widely. Dynamic memory allocation was needed extensively and therefore shared pointers were used for memory management due to their safety. The vector class was implemented with templates in order to give flexibility for optimization purposes later on. Now the implementation was made with floats but with templates this can be changed later on if needed. All containers used are from the standard library.

From external libraries, SFML is used for scene loading. This provides a straightforward way to output image files and works also with Linux. OpenMP is used for running the program with multiple cores. This was seen as a easy way to implement parallelization and the result works well. JsonCpp was included for scene loading. It provides a firm help for loading JSON files. One of the first external libraries used was Gtest which was used for testing. This proved as a great way to create tests already during the course and thus it was an easy choise for this project as well. The command line UI was created with the help of the Boost argument parser. This makes it easy to parse commands and it also handles exceptions which was seen beneficial. Finally, CMake was used for linking libraries together and building the make file.

More information from the following links.

- https://www.sfml-dev.org/
- https://www.openmp.org/
- https://github.com/nlohmann/json
- https://github.com/google/googletest
- https://cmake.org/
- https://theboostcpplibraries.com/boost.program_options

The code was written based on the Google C++ style guide [1] in order to create a clear and easily readable structure. Class and variable names were standardized, all dynamic memory allocation was agreed to be done with shared pointers. Information hiding is used and most of the code is

implemented in cpp files while hpp files are used more as headers. Also, class inheritance is used widely and most parent classes are therefore abstract.

3. Instructions for Building and Using the software

Detailed instructions can be found from the readme file. In a nutshell, following procedures have to be done.

Install the following libraries if not yet installed.

- install SFML
- install openMP
- install Gtest
- install CMake
- install Boost

Run CMake, and build the Make file. The command line UI can be run with for example the following command: ./path_tracer -r 1 -o 500 -s 200 -f randomnew.png

This created a randomscene with 200 anti-aliasing samples, 500 spheres and saves the image in randomnew.png file.

All necessary libraries are installed at the computers of Paniikki. The software is tested with those and should work fluently.

4. Testing

The testing principle was to test everything what was created before merging it to the master branch. In the initial phase of the project, all base classes and methods were extensively tested with the Gtest library which was included to our project. Gtest provided an easy and straightforward way to test the functionality of classes like Ray and Vector. It was also seen important to make a solid ground work since bugs related to these base features might have been difficult to track later on. After going further, more testing was made visually. Materials could be inspected visually quite well. Normal maps were utilized in inspecting geometries since it reveals visually where rays reflect from objects. Performance optimization was tracked with execution times. In addition, all merges were reviewed by another developer from our team and all flaws were corrected before merging to master.

5. Work log

Work was distributed evenly throughout the project timeline as well as developer-wise.

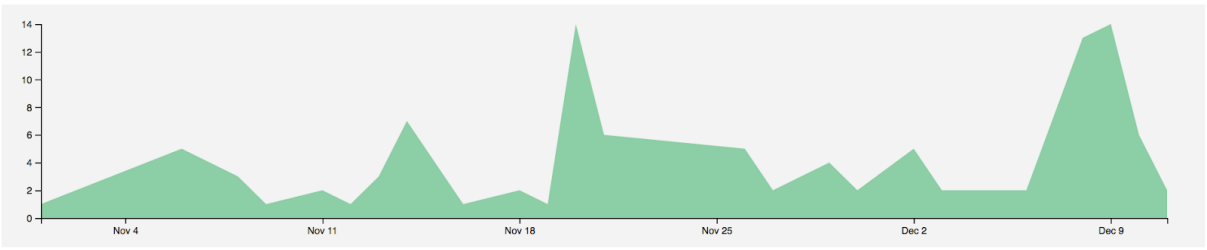

Figure 3. Git commits in a timeline

Chart 1 describes responsibilities and workload on a weekly level. As planned Aleksi, took more responsibility on project management while others focused purely in coding. Besides the features described in the charts, all contributed in reviewing code of co-developers and bugs were fixed often together.

Chart 1. Weekly contributions and work load

Week	**Aleksi**		**Jan**		**Jesse**		**Ville**	
44	src structure, Cmake, Gtest, vectors	10h	Project Plan	4h	Project Plan + Sphere	10h	Ray	9 h
45	OpenCv, helping team to get the environment and coding to start, merger reviewing	10h			Sphere +Normal mapping	10h	Simple path tracer	10 h
46	Code integration and maintenance, merger reviewing	10h	Simple materials	8h	Geomlist	10h	Anti-aliasing	7 h
47	Bounding Volume hierarchy, merger reviewing	15h					Movable Camera	11 h
48	Bounding Volume hierarchy, Fixing bugs, merger reviewing	20h			Scene input/output	5h	Random scene	7 h
49	OpenMp integration and parallelization , merger reviewing	20h	Advanced materials	20h	Scene input/output, keyword arguments	10h	Rects, Boxes, Light source	20 h
50	SFML, Ubuntu integration, Command line	15h	Importance sampling	20h	Scene input/output	20h	Project Report	10 h

	interface, helping others, solving bugs, merger reviewing							

6. References

1. https://google.github.io/styleguide/cppguide.html#Naming
2. https://www.scratchapixel.com/lessons/advanced-rendering/introduction-acceleration-structure/bounding-volume-hierarchy-BVH-part1