ELEC-A7151 - Object oriented programming with C++

Path Tracer - Project Plan

Fall / 2018

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

1. Project Scope

The goal of this project is to produce a software that can render a background and a set of shapes given as an input. Rendering will be done with our own implementation of a path tracing algorithm. [1] The shapes will be given in an input file that describes their position, geometry and a material. A more-in-depth description of architecture and design decisions from the corresponding sections x and y below.

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. The result of a very sophisticated path tracer would be something like below.

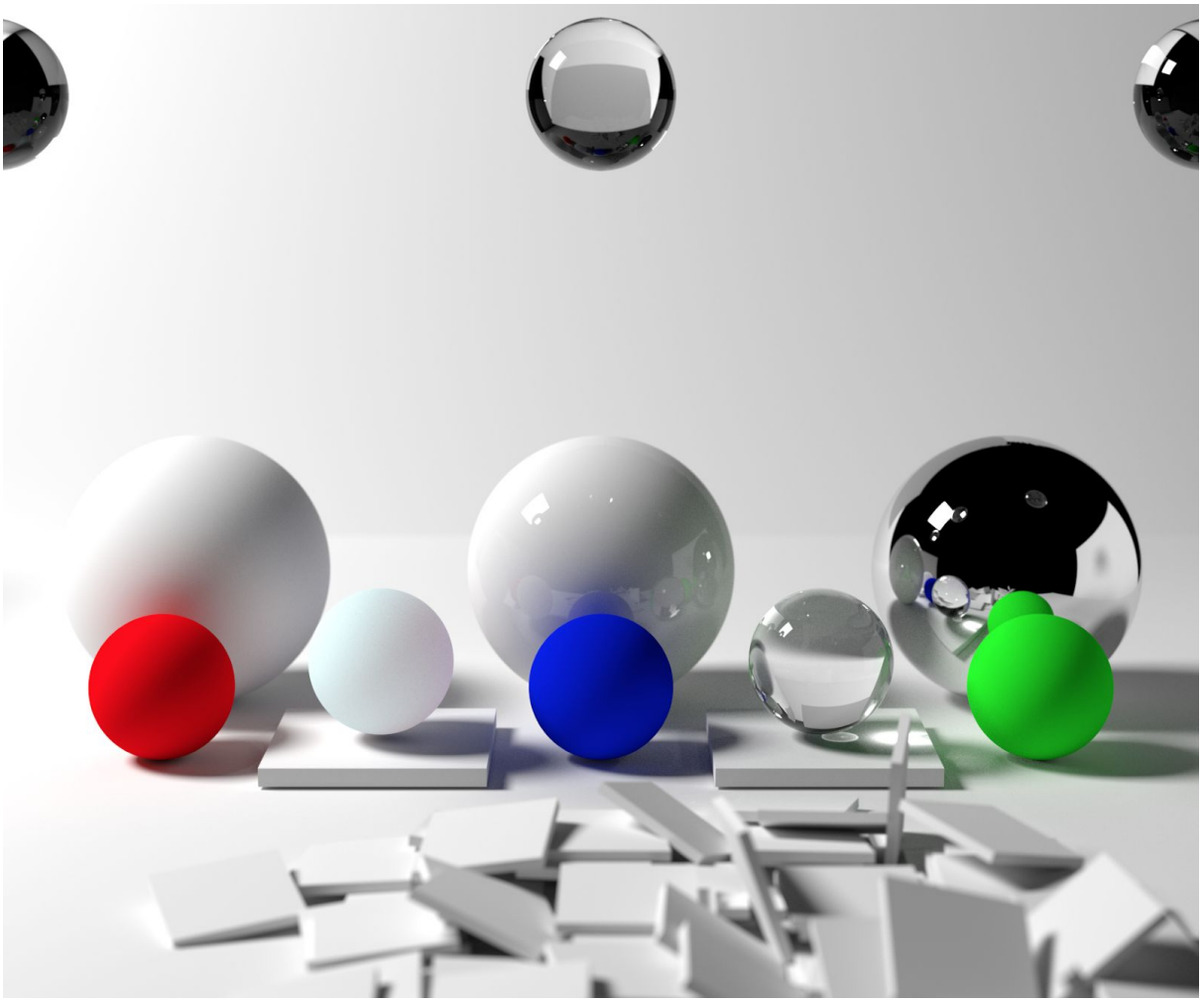

Figure 1. Output of a good rendering software using path tracing. [1]

1.1 Resources and deliverables

For this project we are also considering to use the suggested libraries: SFML [2], Eigen [3] and Json [4] and XML [5] parsers. We are not yet sure if we are using these at all. For example Eigen might not be needed at all, since we think that the vector operations are pretty straight forward.

After all, our vectors are only position vectors in 3D-space, so there shouldn't be a need for efficient matrix calculations that Eigen offers.

Deliverables through this project's timespan are: Project plan, Project Code, Documentation and Final Demo. This exact document is the project plan. The whole point of this plan is to make us sit down and agree on our goals, roles, distribution of work and other preliminary key features of this project. Project code is naturally a deliverable. Without going too much in detail, it will include the following:

- CMake
- cpp.-files
- hpp. -files
- main.cpp

The deliverables will be pushed to GitLab with documentation. This documentation will include unit test files and a READ.ME -file. Also one key deliverable will be the project demo in mid December.

1.2 Tasks and group's way to work

Some of the biggest tasks in this project worth mentioning are: growing domain knowledge, planning the architecture of the software and actually implementing the decided architecture. Some recurring tasks as testing, documenting and updating the plans will also be done during the project. We started growing our domain knowledge by looking at this awesome Disney-video: https://www.youtube.com/watch?v=frLwRLS_ZR0 and also reading this book: https://github.com/petershirley/raytracinginoneweekend/releases/ that Aleksi found.
We have also done some preliminary architecture plans, those will be reviewed below in part 2.

The recurring tasks will be divided as evenly as possible throughout the project. The goal is to let everybody participate both in coding and testing. Also the way people are assigned to code is determined. We have a meeting every Wednesday 6pm-8pm to decide what each of us will implement during the next week and to decide on the implementation assignments guidelines. The guidelines should include at minimum desired inputs and outputs and even some unit tests, if there is enough time on Wednesdays. On Tuesdays 8:30am we meet and review these results.

Also before Tuesdays, if possible we will go through each others codes and comment them. One person will look another's codes. Then we might even do some extra testing if the already passed tests are not sufficient to evaluate the software. Last if everything seems to be alright, we will merge the implemented and tested softwares to the main branch.

2. Major architectural decisions

Based on what we learned from the ray tracing book [6], we planned a preliminary architecture for this path tracing software. This UML-diagram is still a rough estimate of the classes and doesn't yet give much information of the relations, attributes or methods of the classes. However, some major guidelines will be discussed below the diagram.

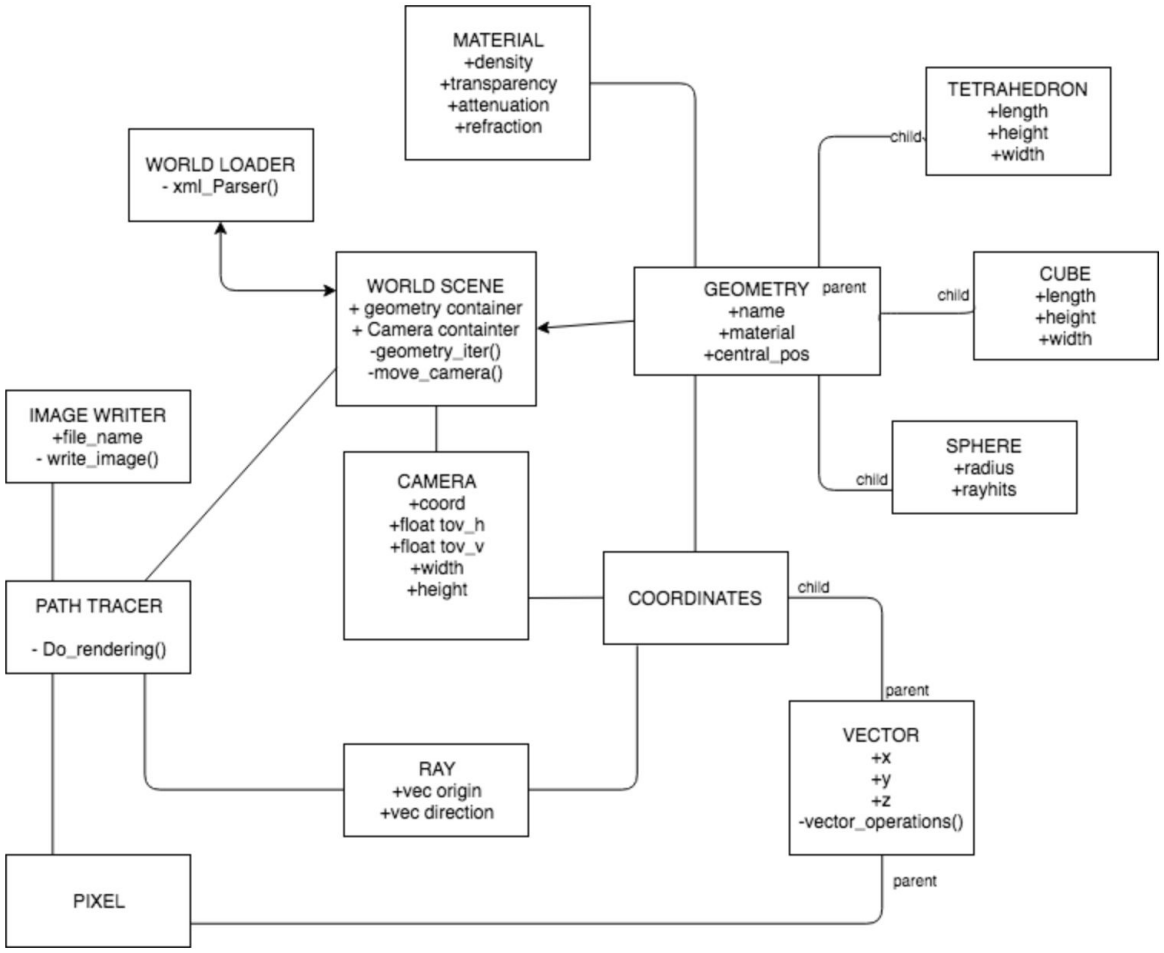

Figure 1. UML-diagram which reveals the planned class structure and relations.

The backbone of the path tracing algorithm is always vector math. Positions, are always indicated by a vector from origin. A ray that we use to simulate one ray of light uses two different vectors. First one to calculate its origin and second one its direction (more of this later). Path tracing cannot work without a Path Tracer class that shoots the rays through all the pixels. It plays a fundamental part in the rendering process. Also we need shapes which we'll implement by Geometry parent class and a few sub classes. The shape also holds a material object that will be used to determine how the ray acts when it hits the shape.

The geometries are held in World Scene class that is basically a big container class, where the World Loader inputs the shapes to be rendered. Path Tracer also uses this information in World Scene to check for ray hits and also to get material parameters of the geometries in the container. Camera object is also contained in World Scene.

3. Design rationale

At this point much of our detailed design is still undone, especially in Path Tracer class. However we have a mutually agreed that the path tracer works by "shooting" rays through pixels and

calculating the average color of a pixel. Then again some of the design choices have already implemented.

We considered making two different classes of vectors. One for Coordinates and one for pixels. The reason for distinguishing them was that there might be a risk inserting pixels into a place where coordinates were expected. We decided that this distinguishment would result in a lot of unworthy code and decided to go with simpler solution. Now there will be a vector class that can hold both RGB values and float values for XYZ values and also calculate all the necessary vector operations. The vector is then used in Pixel and Coordinates classes.

We decided to go with the books [6] way of implementing a ray. Each ray has an origin A and a direction B. We can calculate all the points of in a ray's path with the following equation: p(t) = A + t*B, where t is a parameter. We really didn't think of any other simple solution that could have suited our need better.

4. Preliminary schedule

The plan is to have the software in such a condition that something is already rendered in Mid-term meeting. Other deadlines for our group follow the courses schedule:

- Final demos Dec 10th-14th, where the software has to work.
- Final version with all little tweeks pushed to Git Dec 14th 11:55pm.

5. Distribution of roles

Very shortly:

- Aleksi Hämäläinen
    - Group manager
    - Develops & tests little less than everybody else, only a little.
    - Allocates a little time for planning weekly Wednesday's assigning and Tuesday's review sessions.
- Ville Leppälä
    - Developer
    - Develops & tests as much as everybody else
- Jan Lundström
    - Developer
    - Develops & tests as much as everybody else
- Jesse Miettinen
    - Developer
    - Develops & tests as much as everybody else

6. Algorithms for some "harder" problems

1. We will use the following algorithm in case a we need to test if a ray hits a sphere:
    a. The center of a sphere is C = (cx,cy,cz)

   b. The points on ray's path are p(t) = A+tB
   c. The surface of a sphere is (x-x_0)^2 + (y-y_o)^2 + (z-z_0)^2 = R^2
   d. Now vector from C to p(t) is p(t)-C
   e. so we can use dot((p(t)-C),p(t)-C)) = R^2
   f. Which leads to:
      i. t*t*dot(B,B)+2*t*dot(B,A-C)+dot(A-C,A-C) - R^2 = 0
   g. By checking if this function of t's discriminant is > 0, we see that there is a collision.
2. Making a material look like matte. When a ray hits the surface of a geometry that has material classified as matte. The ray reflects to a direction that is shown by two points: the hitpoint on the material and another point. The another point is randomly selected from a sphere that has a radius of a unit vector and is centered at the length of one unit vector in the direction of the collision normal.
3. Monte Carlo integration is used to smoothen the boundary pixels between an object and background.

7. References

1. https://en.wikipedia.org/wiki/Path_tracing
2. https://www.sfml-dev.org/
3. http://eigen.tuxfamily.org/index.php?title=Main_Page
4. https://github.com/nlohmann/json
5. http://rapidxml.sourceforge.net/
6. https://github.com/petershirley/raytracinginoneweekend/releases/