Path tracing c++

FINAL DOCUMENTATION

Overview

The app is a simple path tracer used for rendering realistic looking images. Scenes are loaded from an input file in a json format. The user is able to move the camera freely and see rendering in real time. It's possible to save the rendered image to a file.

Features

Basic

1. Input of scene file - JSON format used
2. Output of image file
3. Light sources
    - Implemented as geometries (as below) with emission
4. Geometry object
    - Sphere
    - Triangle
    - Parallelogram
5. Material model
    - Color
    - Diffuse material
    - Specular material
    - Refractive material (additional)
6. Shadows
7. Reflections
8. Other
    - Any number of objects in the scene
    - Freely modifiable camera parameters: (position, field of view, resolution)
    - Example scene demonstrating all the features

Additional

1. Monte Carlo integration
2. Real-time preview
3. Real-time camera movement
4. Refractions
5. Multithreading

Software structure

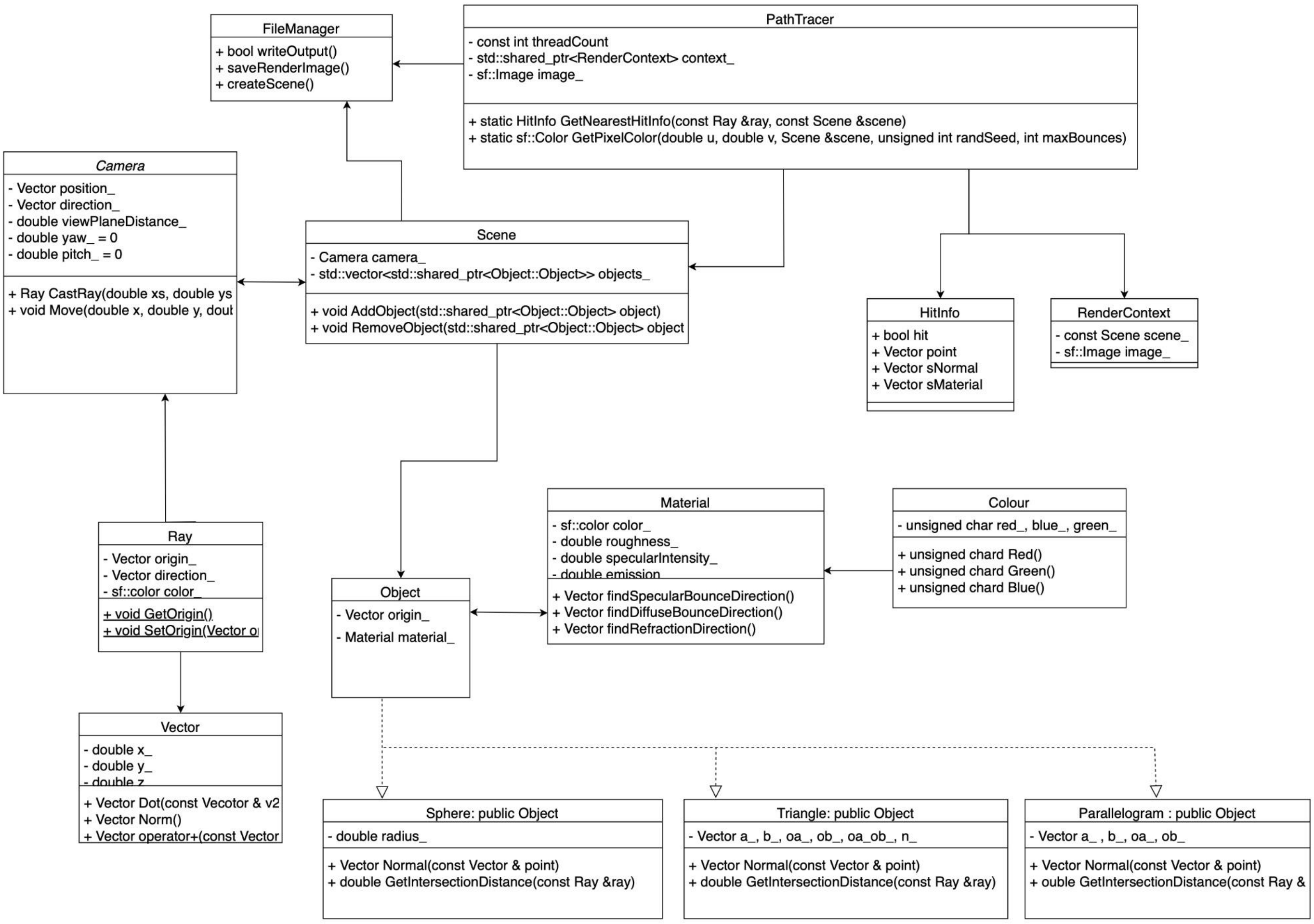

Image 1. Overall architecture, and class relationships diagram.

Instructions for building and using the software

Building and running

- Most of the team worked in Clion, so it's the recommended IDE. It is possible to make it work on VScode, but there might be issues with installing correct libraries for SFML.
- The recommended OS is either Mac or Windows
- CMake is used for building the project. The target is called "app".
- SFML might not be initially installed, so to solve the issue we suggest using:
  - git submodule init
  - git submodule update

Using the software

- The app usage is straightforward - it starts rendering immediately after launch.
- Keeping the scene still for a longer time results in sharper images.
- It is possible to move and look all around the scene using keyboard or mouse.
- There is no option to change the scene in the app, but one can change it by matching the filePath variable in the main function to one of the files in the files directory.

Moving in the scene

- **up arrow**: increase pitch (look up)
- **down arrow**: decrease pitch (look down)
- **left arrow**: increase yaw (look left)
- **right arrow**: decrease yaw (look right)
- **w**: move in the looking direction (forward)
- **s**: move in the opposite of looking direction (backward)
- **a**: move left (sideways)
- **d**:move right (sideways)
- **q**: move to the direction of positive y-axis
- **e**: move to the direction of negative y-axis
- **u**: increase the amount that yaw and pitch change when pressing arrows
- **j**: decrease the amount that yaw and pitch change when pressing arrows
- **o**: increase moving speed
- **l**: decrease moving speed

Creating scenes

The scenes are defined in JSON files. If the user wants to create their own scenes, it is advised to use the example scene file as a model, since it showcases all possible parameters.

Camera has six parameters:

- position - list of 3 doubles
- viewPlaneDistance - double
- yaw - double
- pitch - double
- target - list of 3 doubles
- fov - double

Material has at max seven parameters:

- color - list of 3 ints
- roughness - double
- specularIntensity - double
- specularColor - list of 3 ints
- emission - list of 3 doubles
- n (refractive index) - double
- name - string

Sphere has three parameters:

- origin - list of 3 doubles
- radius - double
- materialName - string

Triangle and **Parallelogram** have four parameters:

- origin - list of 3 doubles
- a - list of 3 doubles
- b - list of 3 doubles
- materialName - string

Comments

- Any material parameter can be omitted, they will be assigned a default value. It is not advised to omit the name parameter.
- Parameter materialName must refer to a name defined in Material

Testing

Testing is done using the doctest library. It is lightweight and easy to use. Most classes have their own test files, each consisting of several test cases that check if functions work as intended. It's possible to run all functions in a file at once. There isn't full coverage, but testing is extensive nonetheless. This solution was an improvement over our first test file implementations that were built using make files.

Work log

The workload was divided fairly across all members. Each task was picked up by a member according to their personal interests and capabilities. At first Aleksi was assigned to update the meeting notes, but eventually we realized it's easier for each member to update their part individually. Meeting dates were decided at times when most members could participate, although it wasn't always possible.

Meeting 31.10.2023 12:00

For the past weekend, each of the members have researched the topic and familiarised themselves with the project concepts.

Hours spent on research:

- **Johannes** ca. 5 hours
- **Markus** ca. 10 hours
- **Aleksi** ca. 10 hours
- **Weronika** ca. 8 hours

On the day before the meeting, Weronika, Johannes and Markus had a quick meeting to discuss the project plan and create UML diagrams.

In this meeting we have:

- Discussed the division of labor
    - We decided that we will divide the classes and topics between two pairs
- Discussed the project structure
- Discussed features
    - Do we want a real time camera control (Yes)
    - Do we need optimization? (Probably)
    - What techniques do we need? (Shadows, reflections, subsurface scattering, etc.)
    - Etc.

- Discussed the schedule
  - In what order do we want to implement the features?
  - What are the milestones
- Implemented the project plan
  - Reviewed, discussed and finalized UML diagram
- Scheduled next meeting

Meeting 08.11.23 13:00

Johannes ca. 15 hours

Wrote Vector class, Ray class, Intersection method for Sphere, Triangle class, simple test files for vector, ray, sphere and triangle.

Markus ca. 14 hours

Wrote FileManager class for interacting with external files. Worked on Object, Colour, and Material classes and tests for creating an image.

Aleksi ca. 10 hours

Set up SFML library for managing windows, put together a simple flat shaded ray tracer, created camera and scene classes, implemented a pipeline for rendering multiple objects and added simple ambient lighting for shading. Made some general changes to vectors and project refactoring

Meeting 17.11.23 12:00

Johannes ca. 14 hours

Worked on Camera class: movement and rotation. Helped with PathTracer

Markus ca. 14 hours

Worked on PathTracer class: Path tracing algorithm. Worked on Material class: Specular, emissive, and diffuse materials and their interaction with rays.

Aleksi ca.

Work on camera class: Fov, movement, rotation and utility. Window resizing.

Weronika ca. 12 hours

Worked on testing using Doctest library. Rewritten most tests and created new ones.

Meeting 24.11.23 12:30

Aleksi ca.
Implemented multithreading for rendering

Weronika ca. 18 hours
Fully implemented tests, refactored FileManager to namespace instead of a class, wrote scene creation function from json input file using nlohmann library
link: https://json.nlohmann.me/

Meeting 02.12.23 12:00

Weronika ca. 9 hours
Made the scene work correctly, refactored all files a little for consistency, split the object into class into multiple files within new directory

Johannes ca. 15 hours

Implemented refractions, added parallelogram object

Features in greater detail

- Geometries
  - Sphere
    - Defined by *origin* (center) and *radius*
  - Triangle
    - Defined by three points (vectors): *origin*, *a* and *b*
    - Direction of the normal vector is calculated as a cross product of vectors *origin-a* and *origin-b*, where e.g. *origin-a* is the vector pointing from *origin* to *a*.
  - Parallelogram
    - Defined by three points (vectors): *origin*, *a* and *b*
    - Direction of the normal vector is calculated as a cross product of vectors *origin-a* and *origin-b*, where e.g. *origin-a* is the vector pointing from *origin* to *a*.
- Refractions
  - Material has field ‘n_’ (refractive index). Probability of reflection is found by calculating reflectance using Snell’s law and Fresnel equations. Based on the probability, ray will either be reflected or goes into the object. Refraction direction is calculated using Snell’s law.
  - Works with spheres assuming they are solid (not hollow).
  - Limitations:
    - Works currently correctly only if one of the media of the interface is free space (air).
    - If a geometry that forms a closed surface is assembled using two-dimensional geometries (triangles and parallelograms), care has to be taken that the normal vectors of the surface point outwards (see Triangle and Parallelogram above).