Computer Graphics, Autumn 2019 Assignment 2 The Path of Light

Alexandros Keros, Kartic Subr

Due date: 04/11/19 (5pm)

Your recent success with the *"Reel to Real"* Studios interview has landed you a position as a computer graphics intern! Your first task is to implement a custom ray tracer, in C++, to augment the company's rendering arsenal. Following the architecture provided, and employing the libraries given, you are asked to populate the necessary classes resulting in a fully functional, photorealistic ray tracer, simulating different cameras, light sources, objects, and shading according to the *Blinn-Phong* model. Using your implementation you should render a scene showcasing the abilities of your custom ray tracer.

Code Architecture

You are provided with class definitions according to the UML class diagram found in Figure 1. The given code supplies all the necessary header files and classes that will dictate your implementation, and it is **compulsory** to use this class structure. The input file should be in *json* format, according to the example of Figure 2, and the output image format is `.ppm` (Refer to "Implementation Details" section for for additional information).

To help you with your implementation, you are given a basic vector library, a json input file parser, and a `.ppm` output file writer. Please read the **Implementation Details** below, as well as the `README.txt` file found in the supplied code archive.

What To Submit

The submitted `.zip` file should contain:

- A **report** explaining the steps taken to implement your ray tracer, including detailed descriptions of feature implementations, and rendered images illustrating each of its abilities. The report should be in `.pdf` format, and explain your work, step by step, with inline images. Figures should be numbered, annotated, referenced and clearly visible. Finally, include a qualitative assessment of your results for each feature of the ray tracer implemented. If multiple cameras and light sources are implemented,

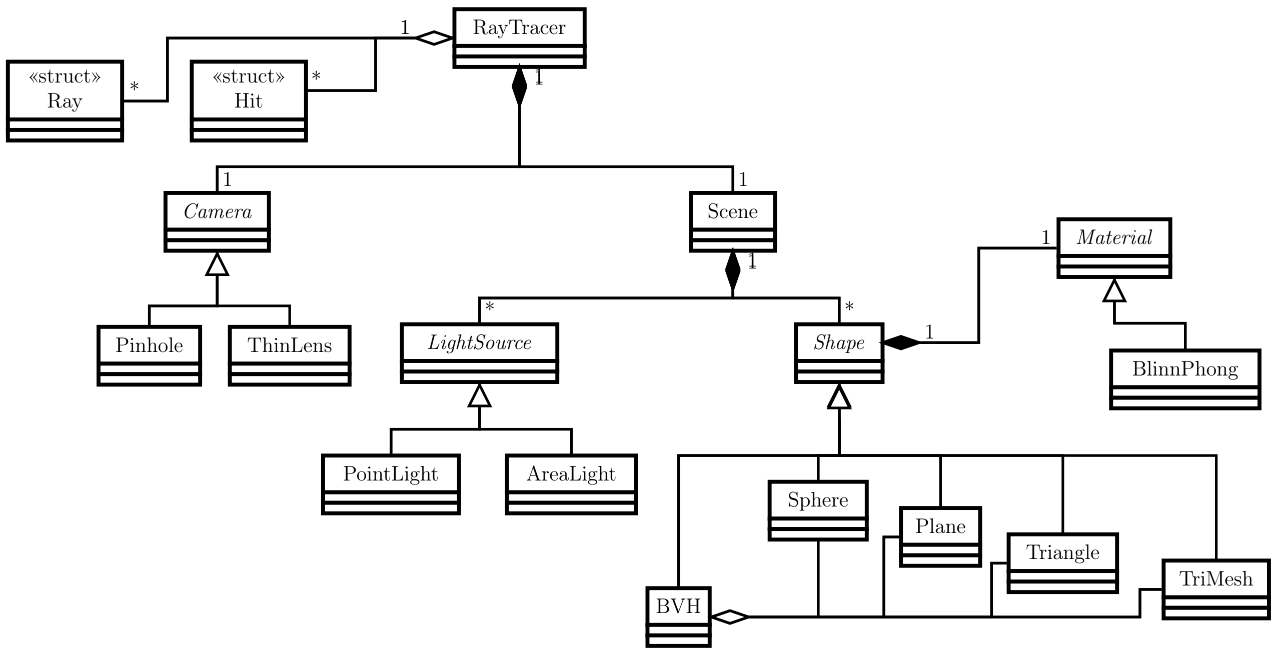

Figure 1: UML class diagram of ray tracer

compare and contrast their output. If optimizations, such as *Bounding Volume Hierarchies* (BVH) are implemented, comment on the computational efficiency of the ray tracer in terms of rendering time/memory requirements, etc.

- A folder containing **your ray tracer implementation**. It should be implemented in C++, and **should follow the class structure provided**. You should implement all functions and include all member variables required. You can add additional classes, as long as their purpose is clearly explained both in the comments and the report. The submitted code should be clean and readable, with indicative variable and function naming, and should contain ample comments describing function's operations and variable roles.

- A **final output image** showcasing the abilities of your ray tracer, in `.ppm` format, along with the **input file(s)** used to generate it.

A submission of only the final output without explanations and intermediate steps will only receive half the credit. **Plagiarism is considered a serious offence**, so please make sure that you supply original code, or clearly reference the sources from which your code originates.

Please name your submission file as `<your UUN>_A2.zip`, and upload it on the *Learn* platform (`https://learn.ed.ac.uk`) as the second assignment for the course.

Marking Scheme

A total of 100 points are assigned for this project, which will then be halved, i.e. its final contribution to your grade will be at most 50%, depending on the marks on your following two assignments. See the course website for clarification. The marking scheme is described below. Numbers in parentheses indicate points of the specific task. Each student is expected to design a unique scene illustrating the abilities of the implemented ray tracer. Provide evidence of having achieved each of the following milestones in your report.

1. Basic ray tracer (Total 30)
    - Ray casting (5)
    - Camera - *pinhole* camera (5)
    - Light source - *point* light source (5)
    - *Shapes* [1]:
        - Sphere (2)
        - Planar quad (2)
        - Triangle (2)
        - Triangle meshes
            - explicit definition (2)
            - `.ply` file parsing (2)
    - Materials & Shading - *Blinn-Phong* model (5)
2. Texture mapping (Total 15)
    - Sphere (5)
    - Planar quad (3)
    - Triangle (2)
    - Triangle meshes (5)
3. *Bounding Volume Hierarchy* (BVH) (20)
4. Distributed raytracing (max. 1 bounce): numerical integration (Total 20)
    - *Thin lens* camera model - *depth of field* effect (5)
    - *Area* light source - *soft shadows* effect (5)
    - Compare random sampling with jittered for above two cases and explain the significance of your results. (5 + 5)
5. Creative modelling for feature demonstration - a scene that demonstrates all the above features (15)

[1]The term *Shapes* is used to refer to scene objects, in order to avoid confusion with C++ objects.

Implementation Details

Supplied code

The code structure supplied is already organized in folders. You are asked to respect the file and class structure, but you can add additional source files in the specified folders, if needed, as long as their addition is justified. To compile your code you will need to use CMake (minimum version v2.8) with the `CMakeLists.txt` file provided. CMake is a tool that aids the build process of your C++ application, and has already been configured for the code and libraries supplied with the assignment. In *unix based* systems, open your terminal and run:

```
cd /path/to/RayTracer
mkdir build
cd build
cmake ..
make
#run your code with
./raytracer <arg1> <arg2> ...
```

Otherwise, you can use your preferred CMake tool to build your project in the operating system of your choice.

UML diagram

Unified Modelling Language (UML) class diagrams provide a useful specification, visualization, and documentation tool for object oriented software systems, with which object/class structure and interactions can be concretely specified in an easy to read graphical format. Specifically, classes, such as the *RayTracer* class, are depicted as rectangular nodes (Figure 3, far left). Inheritance relationship between a parent class (*Camera*) and children subclasses (*Pinhole* and *ThinLens*) are denoted by a hollow arrow pointing from the children to the parent class (Figure 3, second from the left), and signifies that children classes inherit properties of a more generic parent class. Composition and aggregation relationships between classes describe how objects take part in the instantiation of another object (Figure 3, second from right, and far right, respectively). Composition between objects signifies that objects (*LightSource* and *Shape*) exist only with the existence of a third object (*Scene*) and are destroyed when the latter object is destroyed. On the other hand, aggregation is a "part of" relationship, where objects (*Sphere*, *Plane*, etc) come together to form another object (*BVH*), but their existence is not dependent on the latter and they can exist independently.

This class structure, along with the relationships shown in the UML class diagram of Figure 1 and described here, should be present in your implementation.

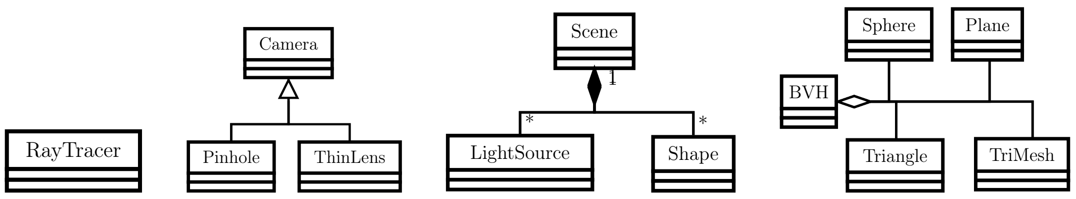

Figure 3: UML relations. Classes are depicted as rectangular nodes, to the left. Inheritance is denoted by a hollow arrowhead pointing from the children classes to the parent class, second to left. The filled-in diamond indicates the composition relation between classes, and the hollow diamond shape indicates the aggregation relation between classes, second to right, and far right. Numbers nearby endpoints dictate multiplicities of the association.

Inheritance

In the supplied code you will find examples of inheritance for two distinct cases. One example is given for the *Camera* base class and its *Pinhole* and *ThinLens* subclasses, where the type of camera is determined at runtime, as parsed by the example input file. The second example provided is that of the *Sphere* object, which is a subclass of the *Shape* class. **You are expected to follow the class structure provided, respecting the inheritance relationships dictated by the supplied code and the accompanying UML class diagram of Figure 1**.

Libraries & methods provided

You are provided with the following libraries, along with examples of their use in the `examples/` folder:

- **Vector library**
  You are provided with a bare-bones vector library that implements all necessary operations between 3 element vectors, `Vec3<T>`, and 4x4 transformation matrices , `Matrix44<T>`, in `math/geometry.h`. A basic how-to example is provided in `examples/vecMatrixExample.cpp`. After the supplied code compilation, you can execute it by running `./vectorexample` from within your build file.

- ***Rapidjson* library**
  Rapidjson (`http://rapidjson.org/`) is a header only json parser/generator implemented in C++, that should be helpful for parsing you tracer's input file. A comprehensive tutorial can be found at `http://rapidjson.org/md_doc_tutorial.html`, and we have included a basic how-to example in `examples/jsonExample.cpp`. After the supplied code compilation, you can execute the example by running `./jsonexample` from within your build file.

- **PPMWriter**
  You are provided with a method to output `.ppm` image files by supplying your final pixel value container, which can be used as:

```
    //int width - image width
    //int height - image height
    //Vec3<float>* framebuffer - pixel value container as
        a Vec3 array, values 0-255
    //char* filename - output image filename
    PPMWriter::PPMWriter(width,height,framebuffer,
       filename);
```

Sampler comparison

For the quantitative evaluation and error convergence comparison of jittered and uniformly random sampling, you are asked to compare the mean squared error (MSE) of the *linear RGB* values (in range $[0, 1]$) of pixels within a region of the output images. Firstly, you render a scene using high sample count (5000 and above, or until convergence has been achieved, i.e. no noise artifacts are visible in the image), which will act as the reference image I_{ref}. Then, you render scenes with stepwise increasing sample counts (10-2000 with step size 50) using both uniformly random and jittered sampling. Finally, for each sampling method and sample count you compute the MSE of the *linear RGB* values over a specified region or the image

$$\frac{1}{N}\sum_i [|R_{i_{ref}} - R_{i_{test}}|^2 + |G_{i_{ref}} - G_{i_{test}}|^2 + |B_{i_{ref}} - B_{i_{test}}|^2],$$

where i is the index of pixels contained within an N-pixel region. $R_{i_{ref}}, G_{i_{ref}}, B_{i_{ref}}$ and $R_{i_{test}}, G_{i_{test}}, B_{i_{test}}$ indicate the *linear RGB* values of the i-th pixel in the reference image and the test image (rendered with either jittered or random sampling), respectively. Plot the MSE against the increasing number of samples in log-log scale for both sampling strategies and comment on your findings.

```
{"nbounces":3,
 "camera":{
        "type":"pinhole",
        "width":800,
        "height":800
 },
 "scene":{
        "backgroundcolor":[0.01, 0.01, 0.01],
        "lightsources":[
                {
                "type":"pointlight",
                "position":[0.1, 0.1, 0.1],
                "intensity":[1., 1., 1.]
                }
        ],
        "shapes":[
                {
                "type":"sphere",
                "center": [0.2, 0.3, 0.4],
                "radius":0.3,
                "material":{
                        "ks":0.4,
                        "kd":0.8,
                        "specularexponent":3,
                        "diffusecolor":[0.4, 0.3, 0.4]
                        }
                },
                {
                "type":"sphere",
                "center": [0.2, 0.5, 0.7],
                "radius":0.1,
                "material":{
                        "ks":0.6,
                        "kd":0.2,
                        "specularexponent":10000,
                        "diffusecolor":[0.4, 0.4, 0.4]
                        }
                }
        ]
 }
}
```

Figure 2: Input file example