Computer Graphics, Spring 2023
Assignment 2
The Path of Light

Chris Lochhead, Amir Vaxman

Due date: 13/03/23 (5pm)

Your recent success with the *"Reel to Real"* Studios interview has landed you a position as a computer graphics intern! They were impressed with your creativity and your capacity to gather real-world skills in such a short time, but before they let you tinker with their renderer, they want you to hone your programming and debugging skills. They have assigned you the task of implementing a custom ray tracer, in C++, as a 3-week training project. Specifically, they want to ensure that by the end of the training program, you can develop code while adhering to a specific design specification. The specs and libraries they want you to use are detailed in this document. You are required to populate the necessary classes so that the result is a photorealistic ray tracer that can shade according to the *Blinn-Phong* model. Finally, they would like to see a customised scene to showcase the abilities of your custom ray tracer. Since the company also values effective communication and collaboration, they would like you to present[1] your work in 5 minutes (on a date that will be announced after the deadline for the assignment).

Note: Trainees of the past have been known to complain, sulk and vent after starting on this assignment too late. It is strongly recommended that you *solve this assignment working regularly, and starting early,* to leave sufficient time for debugging. Also, please bear in mind that this is part of your training and that "Reel to Real" is already impressed with your capability. So, do not be anxious if you cannot solve all elements of this assignment. Happy coding!

1 Workflow

It is suggested that you develop your raytracer in the following sequence.

1. Complete parser
    - Just use printed messages to ensure that all scene components are created, initialised and destroyed correctly.

[1]You may be exempt from this if you have a valid reason. Please contact Amir if applicable.

2. Implement pinhole camera and a pointlight (Figure 1a).
3. Implement ray-sphere intersection (Figure 1b).
4. Implement ray-plane (planar quad) intersection (Figure 1c).
5. Implement ray-triangle intersection (Figure 1d).
6. Implement Blinn-Phong shading (Figure 1e).
7. Implement reflections and refractions (if not already) (Figure 1f).
8. Implement texture mapping in Sphere (Figure 1g).
9. Implement texture mapping in PlanarQuad (Figure 1h).
10. Implement texture mapping in Triangle (not displayed in our example).
11. Implement triangle meshes (not displayed in our example).
12. Implement texture mapping in TriangleMeshes (not displayed in our example).
13. Implement a Bounding Volume Hierarchy (BVH) (not displayed in our example).
14. Implement distributed raytracing (not displayed in our example).
15. Make your own scene to demonstrate the features (As seen in Figures 3, 4, 5 and 6).

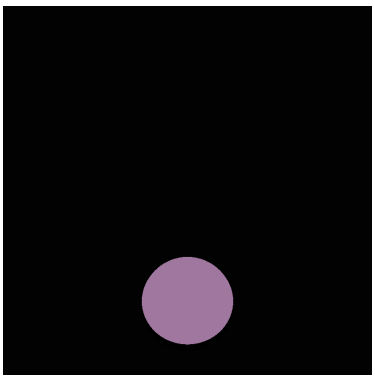
Note: While following the above step-by-step approach, you may need to adjust the provided example.json file accordingly, by removing content that is beyond the scope of your existing implementation. For example, you can remove the textures from the materials until you implement the texture functionality, to make testing the rest of your raytracer features easier. This will minimise the number of corner-cases that you will have to account for in your code, while your project is still in a development phase, allowing you to focus your effort on delivering the required functionality. *Of course, the version of your project that you will submit, should run with any given json file, producing an image in case of correct input, or helpful error messages, otherwise.*

2 What To Submit

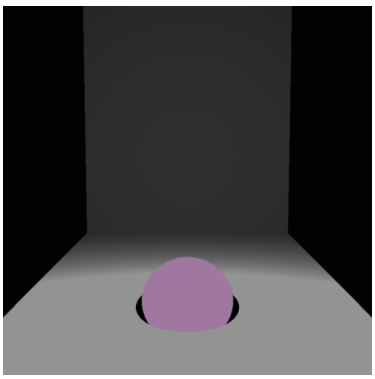
The submitted .zip file should contain:

- A **report** explaining the steps taken to implement your ray tracer, including 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

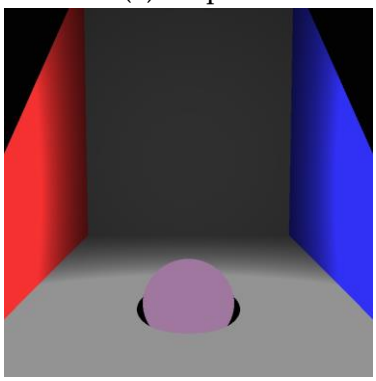
(a) Step 2



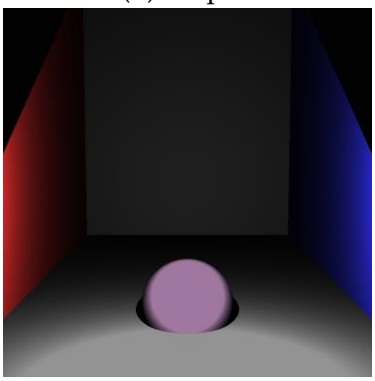
(b) Step 3



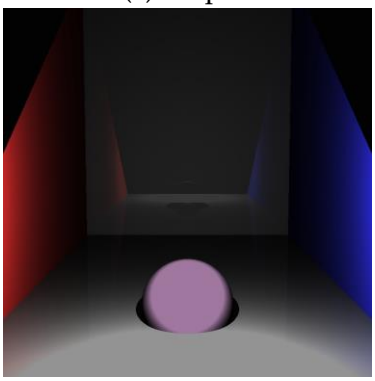
(c) Step 4



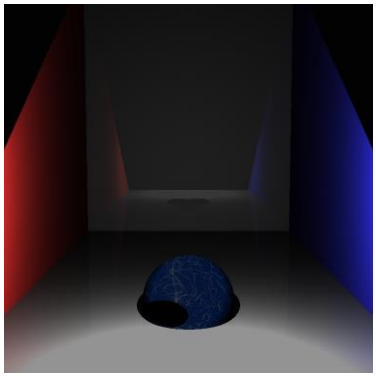
(d) Step 5



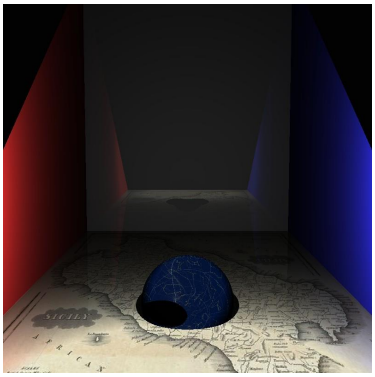
(e) Step 6



(f) Step 7



(g) Step 8



(h) Step 9

Figure 1: Step by step scene rendering

the ray tracer implemented. If multiple cameras and light sources are implemented, 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 the member variables required. You can add additional classes, as long as their purpose is 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 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**. Your submissions will be analysed using automatic plagiarism detection software. If you have copied code snippets from online resources, clearly reference/cite their sources (e.g., comment /* source: <url>, <brief comment> */ when applicable). If code for the main function- ality (marked below) is copied then you will not receive credit for it, but if it was adapted you will receive partial marks. Even though you may not get the marks for them, snippets with citations could allow you to proceed with other steps for which you can receive full credit.

3 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%. 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 25)
    - Ray casting (5)
    - Camera - *pinhole* camera (5)
    - Light source - *point* light source (5)
    - *Shapes* [2]:

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

  - Sphere (1)
  - Planar quad (1)
  - Triangle (1)
  - Triangle meshes
    * explicit definition (1)
    * .ply file parsing (1)
- 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 with a comparison of performance. (25)
4. Distributed raytracing (max. 1 bounce): numerical integration (Total 25)
   - *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 (5)
6. Presentation (5)

4 Code Architecture

You are provided with class definitions according to the UML class diagram found in Figure 2. 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[3]. The *input file* should be in .json format, according to the example of Figures 3, 4 and 5, and the *output image* format is .ppm

To help you with your implementation, *you are given*:

- a basic vector library,
- a .json input file parser, and

[3]The aim is to help you get used to the object-oriented code architecture and to help the markers evaluate your submission fairly.

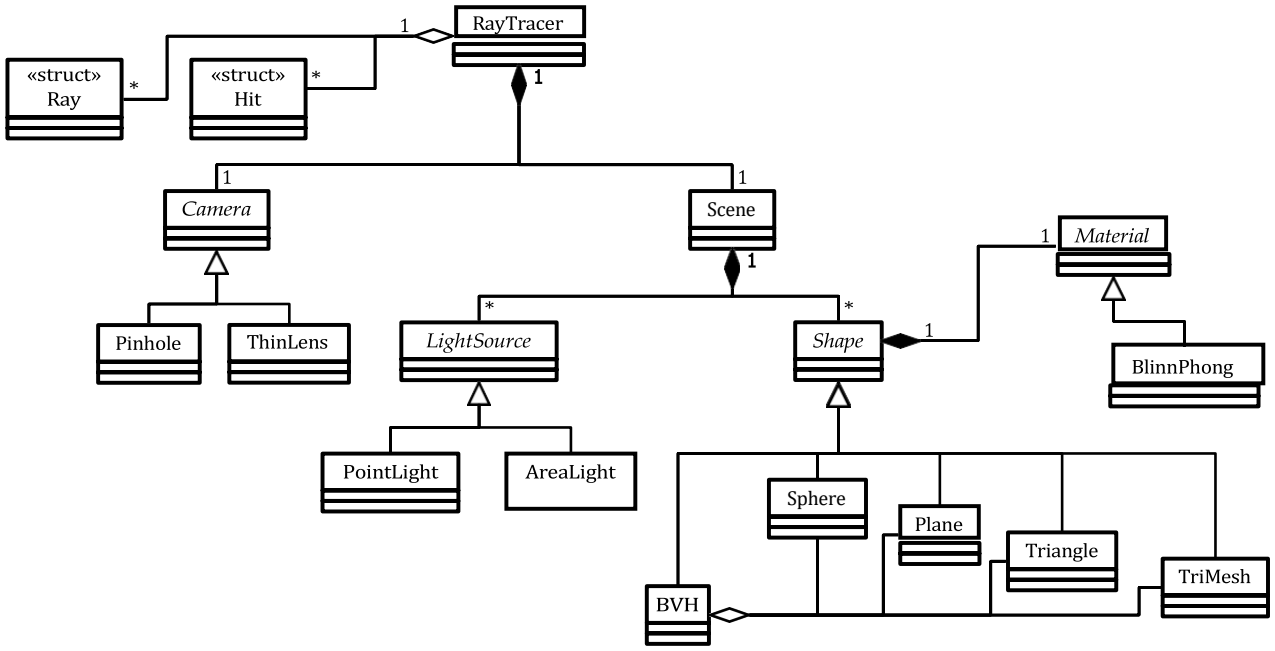


Figure 2: UML class diagram of ray tracer

- a .ppm output file writer.

For additional information, please read the **Implementation Details** and the **Some Tips** (Sections 5 and 6) below, as well as the README.txt file found in the supplied code archive.

```
{
" nbounces":3 ,
"camera":{
    "type":"pinhole",
    "width":800,
    "height":800,
    "fov":35,
    "position": [0.0, -3.5, 2.0],
    "lookat": [0.0, 1.0, -0.1],
    "up": [0.0, 0.0, 0.5]
},
"scene":{
    "backgroundcolor":[0.01, 0.01, 0.01],
    "lightsources": [
    {
        " type ":    " pointlight",
        "position": [0.0, 1.0, 1.5],
        "is": [100.0, 100.0, 100.0],
        "id": [100.0, 100.0, 100.0]
    }
    ],
    "shapes":[
    {
        "type":"sphere",
        "center": [0.0, 2.7, 0.15],
        "radius":0.5,
        "material":{
            "ks":0.4,
            "kd":0.8,
            "    specularexponent":   3   ,
            "diffusecolor":[0.4,  0.3,  0.4],
            "tPath": "../textures/sky-map.jpg",
            "tWidth": 850,
            "tHeight":  480
        }
    },
```

Figure 3: Input file example (a)

```
37      {
38          "id": "back wall",
39          "type": "plane",
40          "v1": [-2.0, 8.0, 0.0],
41          "v2": [-2.0, 8.0, 5.0],
42          "v3": [2.0, 8.0, 5.0],
43          "v0": [2.0, 8.0, 0.0],
44          "material":{
45              "ks": 0.6,
46              "kd": 0.7,
47              "kr": 0.3,
48              "specularexponent": 10,
49              "diffusecolor": [1.0, 1.0, 1.0]
50          }
51      },
52      {
53          "id": "floor",
54          "type": "plane",
55          "v3": [-2.0, 0.0, 0.0],
56          "v0": [-2.0, 8.0, 0.0],
57          "v1": [2.0, 8.0, 0.0],
58          "v2": [2.0, 0.0, 0.0],
59          "material": {
60              "ks": 0.6,
61              "kd": 0.7,
62              "kr": 0.3,
63              "specularexponent": 10,
64              "diffusecolor": [1.0, 1.0, 1.0],
65              "tPath": "../textures/sicily.jpg",
66              "tWidth": 600,
67              "tHeight":  490
68          }
69      },
```

Figure 4: Input file example (b)

```
    {
        "id": "left wall",
        "type":  "triangle",
        "v0": [-2.0, 8.0, 0.0],
        "v1": [-2.0, 8.0, 5.0],
        "v2": [-2.0, 0.0, 0.0],
        "material": {
            "ks": 0.6,
            "kd": 1.0,
            "specularexponent": 10,
            "diffusecolor": [1.0, 0.2, 0.2]
        }
    },
    {
        "id": "right wall",
        "type":  "triangle",
        "v0": [2.0, 8.0, 0.0],
        "v1": [2.0, 0.0, 0.0],
        "v2": [2.0, 8.0, 5.0],
        "material": {
            "ks": 0.6,
            "kd": 1.0,
            "specularexponent": 10,
            "diffusecolor": [0.2, 0.2, 1.0]
        }
    }
    ]
}
}
```

Figure 5: Input file example (c)

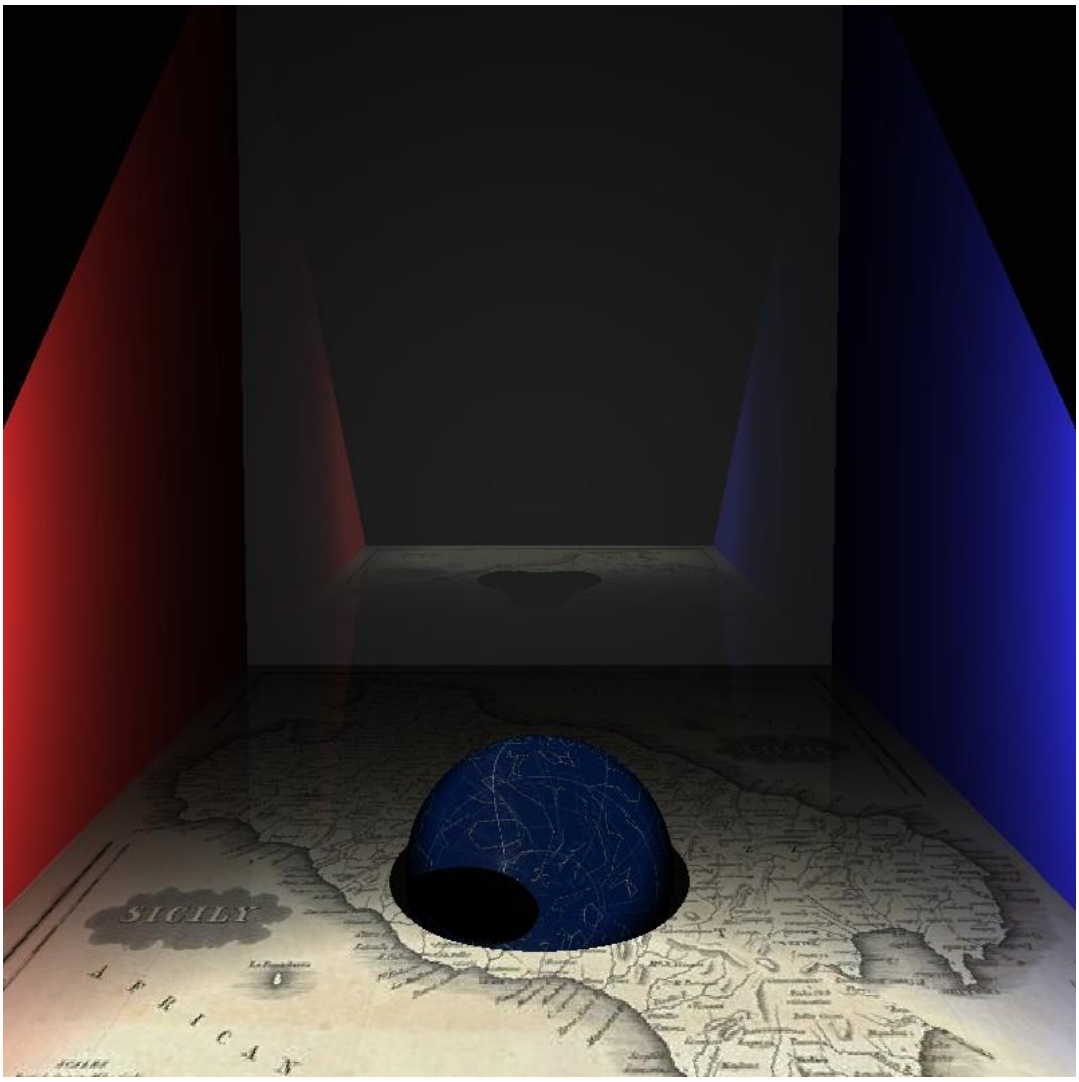


Figure 6: Input file result

5 Implementation Details

5.1 Supplied code

The code structure supplied is already organised 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, just open your terminal and run:

```
1 cd /path/to/RayTracer
2 mkdir build
3 cd build
4 cmake ..
5 #run your code with
6 ./raytracer <arg1> <arg2> ...
7
```

Alternatively, you can use your preferred CMake tool to build your project in the operating system of your choice. In this case, however, before submitting your assignment, you are asked to **make sure that your project can be build and executed as intented in a *unix based* system**.

5.2 UML diagram

Unified Modelling Language (UML) class diagrams provide a useful specification, visualisation, and documentation tool for object-oriented software systems. With UML, 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 7, far left).
- **Inheritance** relationship between a parent class (*Camera*) and child subclasses (*Pinhole* and *ThinLens*) are denoted by a hollow arrow pointing from the children to the parent class (Figure 7, second from the left). Inheritance 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 7, 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 2 and described here, should be present in your implementation.

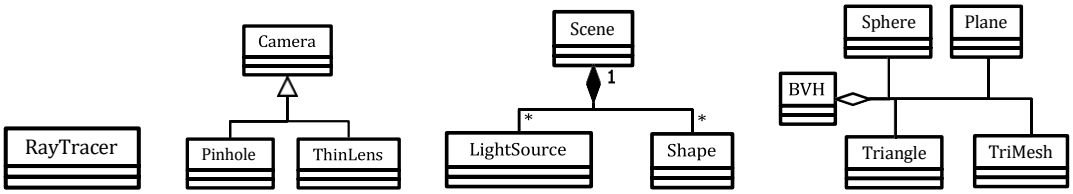


Figure 7: UML relations. Classes are depicted as rectangular nodes, to the left. Inheritance is denoted by a hollow arrowhead pointing from the children classes to the parent class, second to left. The filled-in diamond indicates the composition relation between classes, and the hollow diamond shape indicates the aggregation relation between classes, second to right, and far right. Numbers nearby endpoints dictate multiplicities of the association.

5.3 Inheritance

Inheritance is one of the most useful concepts of the Object-Oriented Programming paradigm. It enables logically linking functionality modules in a hierarchical way, making your code more expressive and portable, while reducing development time. You are advised to view this project as a chance to get comfortable with Inheritance and get experience in it in a highly practical problem setting.

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 2.

5.4 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 matri- ces , 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:

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

5.5 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.

The sampler comparison workflow goes like this:

1. 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}.

2. Then, you render scenes with step-wise increasing sample counts (10-2000 with step size 50) using both uniformly random and jittered sampling.

3. 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 refer- ence image and the test image (rendered with either jittered or random sampling), respectively.

4. Finally, plot the MSE against the increasing number of samples in log-log scale for both sampling strategies and comment on your findings.

6 Some Tips

1. Start by carefully going through the provided code and understanding how the provided classes correspond to the Figure 2 UML diagram.

2. Write the basic code for a scene creation (class *constructors*) and destruction (class *destructors*). Aim to create, print the members, and destroy the objects of the scene provided in example.json (Figures 3, 4 and 5, result visible in Figure 6). Make sure that you are comfortable with how each class fits in the overall raytracer logic, before getting to implement any actual functionality.

3. Start modestly. Aim to create a scene with just a pinhole camera, a pointlight and a sphere.

   (a) First, make sure that the sphere is rendered correctly and appears where you expect it to be in the output image.

   (b) Then, work with the lighting, making sure that the sphere is lighted up in a realistic way.

4. Add a planar quad (*plane*), either under the sphere (like a floor), or behind it (like a wall).

   (a) Make sure that both objects are lighted up and rendered correctly.

   (b) Then, make sure that the sphere's shade is realistically casted in the plane.

5. Get creative! How about the plane having reflective properties? Make sure that your raytracing logic holds true for this more challenging case, as well.

6. Add a second sphere with different properties than the existing one (change colour and size, to begin with). Make sure that your three objects (planar quad and two spheres) interact realistically in terms of lighting (colouring, shading, reflections).

7. Further "bound" the scene. Add another plane, thus having both a floor and a wall facing against the camera. See that lighting, shading and reflections work as you would expect.

8. Go to implement triangle and triangle meshes. Make sure that all types of objects you created can harmonically exist in a scene.

9. Challenge yourself by trying to implement more advanced features, like texture mapping, Bounding Volume Hierarchy (BVH), thin lens camera and area light source. Carefully document your ideas, your attempts and your results. What challenges did you faced and what resources did you consulted trying to overcome them?

    - Don't get discouraged. If you reached this far you are already doing well. Try to challenge yourself and demonstrate your understanding. *An honest attempt will be appreciated, even if the result is not excellent.*