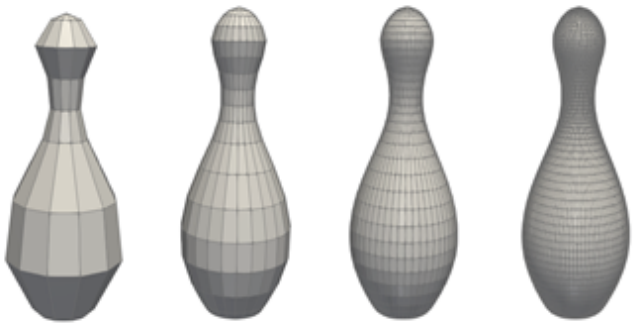
Scientific Computing for Mechanical Engineering
— The Raytracer —

Joris Remmers

March 12, 2025



Version 2024-2025

Instructions

This document describes the final group Raytracer project for the course Scientific Computing for Mechanical Engineering [4ME30]. In Section 1 the background of the program is introduced, as well as the basic concepts of the underlying mathematical-physical model. Section 2 describes in detail all the required developments. These developments are presented in assignment boxes, in which the deliverables are also listed.

The final deadline of this project is **April 21st 2025**. On this date, the following deliverables are due:

Individual Report
Notes on the code development and the results of your own component are included in the Individual Report. You must use the template `raytracer_indiv_report.tex`. Details on the exact contents of this report are given in this assignment document. The report must be uploaded as a pdf file on the Canvas website before the due date.

Group Report
A short user manual, notes on code integration and a number of showcase renders are included in the Group Report. You must use the template `raytracer_group_report.tex`. Details on the exact contents of this report are given in this assignment document. The report must be uploaded as a pdf file on the Canvas website before the due date.

Source code and example input files
All source code must be developed in your group's `Gitlab` repository. After the due date your access to your `Gitlab` repository will become read-only. The latest push to the `Gitlab` repository **main** branch before the deadline will be assessed, unless stated otherwise. The source code of the unit tests need to be stored on gitlab as well. All functions and structures must be documented in the source code in terms of short descriptions and user contracts. The input files that are needed to reproduce the images can be stored in the examples directory in the repository. It is not allowed to include the images in the repository, unless these images are used in the Raytracer calculation (for example textures).

1 Introduction

Raytracing is a numerical technique to make photo realistic, computer generated images (CGIs). It was developed in the 1970s and was used for the first time in feature movies in the 1990s. In the 1993 movie Jurassic Park, most of the dinosaurs were created using ray tracing techniques. Toy Story (1995) was the first movie that was completely computer generated. In most modern day movies, backgrounds, special effects and even characters are generated using raytracing techniques. A good example is the introduction of a human character in Star Wars: Rogue One (an actor that should look the same as 40 years before) that is completely rendered with a computer, see Figure 1.

Figure 1: Examples of Computer Generated Images (CGI) by using Raytracing techniques in feature length movies. Left: Jurrasic Park (1993), Middle: Toy Story (1995) and Right: Rogue One, a Star Wars Story (2016). All images from Google images, 5-2-2020.

1.1 Goal

In this assignment, you will be developing a Raytracer to create a scene in which a sports car is presented in front of the well-known flying pins. To this end, you will receive a framework of the code in C, the geometry of a single pin and the wireframe model of the car (a BMW M6, model 2006).

1.2 Raytracing for Mechanical Engineers?

You may wonder why you should be able to develop a raytracer as a Mechanical Engineer. Well, the answer is simple. Raytracing is a relatively simple and straightforward concept, but contains many numerical techniques that can also be found in the software development in the Mechanical Engineering practice. For example, you can think of the physics of light propagation, homogenised material properties, the high performance computing aspect of a large number of particles (photons) and the fact that you are dealing with meshes and mesh-refinement strategies.

1.3 Organisation of the project

This project is composed of the following 5 components.

1. **Smooth shading** The smoothness of the multifaceted objects can be improved by implementing an interpolation algorithm, such as Gouraud's or Phong's algorithm.


Figure 2: Left: The iconic "Flying Pins" in front of the Campus of Eindhoven Univeristy of Technology. Right: A render of the BMW M6. Images from Google images, 5-2-2020 and 5-1-2022.

2. **Various material models** Currently, the code only contains a model for a matte material. You are asked to implement other material models to represent glossy and even reflecting materials.

3. **Sampling and alternative camera models** In the current code, a simple pin-hole camera is implemented. This is a camera model for which every object is in focus. A more realistic image is created with a camera model in which some objects are out of focus. In addition, new scanning (sampling) techniques improve the quality of the picture.

4. **Shadows and new light models** Light sources that do not leave shadows will obviously never result in a realistic image. So, an algorithm to model shadows must be implemented. In addition, new light models (spot lights, colored lights and surface lights) will result in an every more realistic scene.

5. **Code integration and optimisation** A ray tracing algorithm can quickly become very inefficient. Clever techniques can be used to reduce the computation time drastically. The final project component will be dedicated to the implementation of numerical techniques to improve the speed of the program.

A detailed explanation of the 5 components is given in the next section.

1.4 Background reading

Raytracing is a popular field within computer science and has been described in a large number of books. For this project, the following sources are recommended for further reading:

- The book *Physically Based Rendering: From Theory to Implementation*, by Matt Pharr, Wenzel Jakob and Greg Humphreys [1] is a very extensive source on raytracing techniques and the numerical implementation thereof. It can be downloaded for free from the library of the university.

  There is no need to read all of its 1266 pages, but for some components of this project it gives the proper amount of theoritical background. If so, the correct chapters and pages will be indicated.

- The online document *Raytracing in One Weekend* [2] gives a nice overview of the basics of raytracing, illustrated by the development of a small code in C++. The code lacks structure and abstraction and can only be used for to produce very simple scenes. However, it demonstrates the implementation of some of the algorithms that need to be implemented in this project. Please do not get frustrated by the title of this document. It really takes more than one weekend to implement a decent raytracer code.....

2 Program components and pair programming

The project has a due date April 21st 2025. In this section, the assignments and deliverables for both phases will be presented.

2.1 The starting point

You will receive a working C-code that can be used as the basis for your own code. The code can be compiled in VS Code using a Makefile (Linux). Please read the installation instruction in the file `README`.

After a successfull compilation, the program can be used to generate an image. The details of the scene such as the geometry, light sources and materials are saved in an ascii input file with the extension `.in`. You can run the executable in a terminal or command prompt by giving the entire path to the executable, followed by the input file. In Windows, the execution of the `singlePin.in` example, located in the directory `testcases/singlePin` is done as follows:

```
../../raytracer.exe singlePin.in
```

The program will run for a few seconds (30 seconds at most) and create a file `singlePin.bmp`, which contains the image shown in Figure 3.

After you have installed, compiled and run the executable, it is time to look at the code itself. It consists of the following components:

- An input file reader that reads the very basic ascii input file with the extension `.in`.
- A pinhole camera, which is a simple camera model in which all objects are in focus.
- Two types of shapes: spheres and triangular/quadrilateral surfaces. The intersection algorithm for a ray with these shapes have been implemented, according to Section 3.2 and Section 3.6 from [1].
- A simple sunlight model.
- A Lambert material model, for a number of primitive colors.
- A tool to store the image in bitmap format (`.bmp`).

All project components must be made in this existing C code. Additional software in order to create more complex input files may be written as a separate code made in Matlab or Python. You can think of small pieces of code to scale, translate and rotate the objects to create a larger input file.

2.2 The preparation phase

In order to prepare for the project, you are asked to:

- Study the C code that you will receive via your `Gitlab` account. The code is a basic framework for a raytracer and consists of a number of files, which are partially documented by means of comments (function descriptions and contracts). In addition, a simple input-file `singlePin.in` is given.

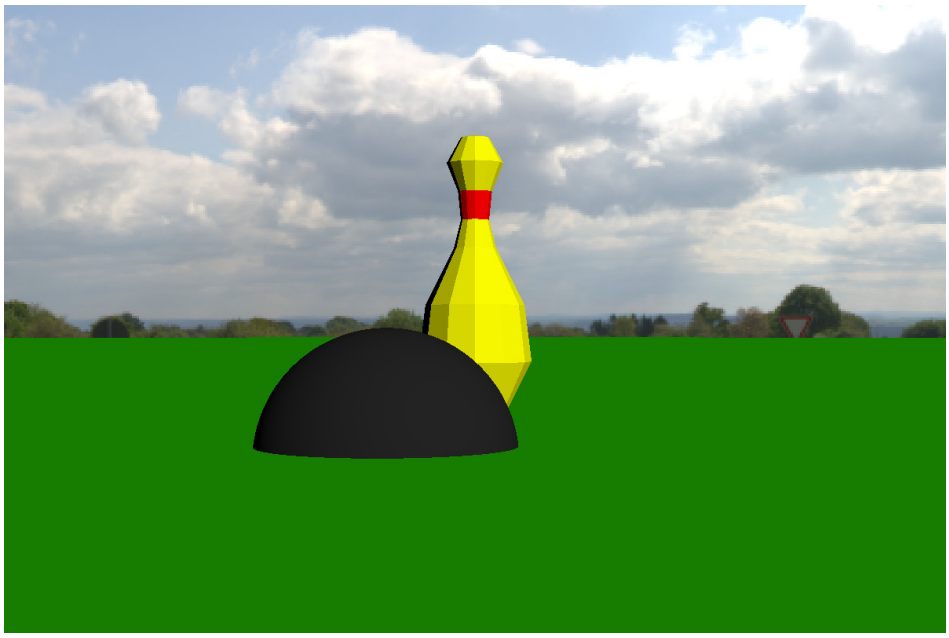


Figure 3: The image that will be generated by running the original raytracer code.

- Provide documentation for the entire original code by means of function descriptions and contracts. For now, only the file `vector.h` is documented. Adding function descriptions and contracts is a perfect way to understand the code and its components in more detail. Avoid writing comments in the actual code implementation.
- Read Section 1.2 of the book by Pharr et al. [1]. This section gives a nice overview of Raytracer components. The estimated reading time is about 30 minutes. You may browse through the rest of this chapter to get some inspiration. Another overview of Raytracing techniques can be found on Wikipedia [5].
- Distribute the 5 components (the different assignments that will be discussed in the next section) among the five group members. The components are developed using the concept of pair programming. Hence, one team member is the responsible *Driver* for a component, and another team member acts as the *Navigator*. In case your group is smaller than 5 persons, you may skip one component, or you may assign multiple components to a single member of the group. Please discuss this with the lecturer.
- The *Driver* is the sole responsible programmer for this component. He/she writes the report and receives the full grade for this component.

Note that there is no deadline or deliverable to finish this phase, but we strongly recommend that you have distributed the work before March 17, 2025.

2.3 The project phase

In the project phase, the 5 components of the code (and the bonus assignment) are developed simultaneously. Please develop each component in a unique branch. A proper branch name is suggested.

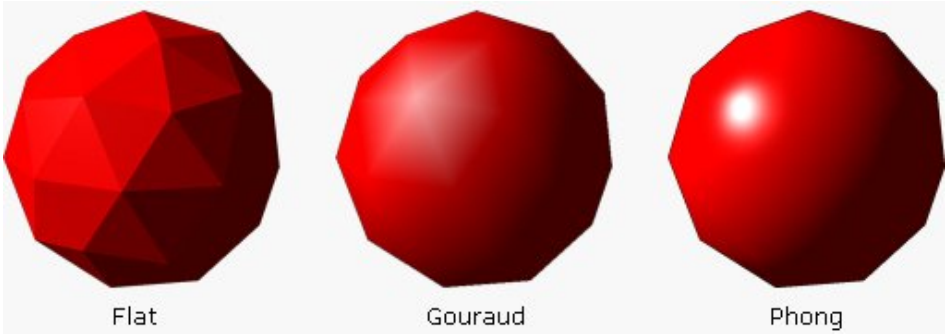


Figure 4: The effect of Gouraud and Phong shading on the smoothness of an object.

2.3.1 Component 1. Smooth shading

In the current code, two type of objects can be used: spheres and multifaceted objects. The spheres are described by a exact analytical function and appear perfectly smooth in your render. The multifaceted objects however consist of piece-wise linear (flat) surfaces. Their boundaries remain visible in your renders as sharp lines, even for relatively dense meshes.

The root cause of this is effect is that the light intensity of the surface is based on the angle of a outward normal vector of the face with respect to the incoming light source. This outward normal vector is constant for the face and changes abruptly at from face to face causing an abrupt in intensity, which results in the hard lines in your images.

The problem can be alleviated by using an algorithm that interpolates the angle of the outward normal of the face using the outward normal of neighboring faces. This enables a smooth transition of the intensity of faces boundaries, even for coarse meshes. Examples of such algorithms are Gouraud [3] and Phong [4].

The procedures consists of two steps. Prior to the actual ray tracing, the average outward normal for each vertex in the model must be calculated, based on the outward normal vectors of the adjacent surfaces. Then these vertex normal vectors can be used to construct the normal vector in a given point on the face using a set of interpolation functions (shape functions) that are very similar to the shape functions used in the Finite Element assignment.

Tasks

- Develop and implement a data structure to store the outward normal per vertex.
- Implement a routine to calculate these outward normal vectors.
- Implement the interpolation function to calculate the local outward normal in a face.

In addition, you as the "mesh expert", you are responsible for the development of additional tools, written in Matlab or Python to transform (move, rotate and scale) the objects in your scenes. This code can be placed in the directory **tools**, but does not need to be documented in the report.

Deliverables

In the **individual report**, you describe the theory behind the algorithm, present the pseudo-code and give implementation details. You show the performance of the algorithm by means

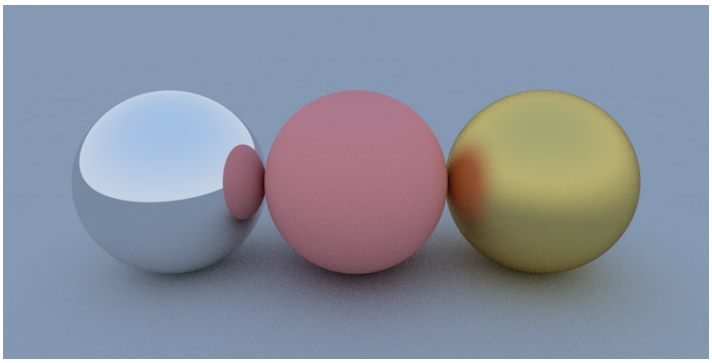


Figure 5: Demonstration of various materials.

of an example (figures) in which you clearly demonstrate the effect of this interpolation step.

You develop your code in the branch `smoothing` and make sure that this branch is merged with `main` branch. You add all the input-files of all the examples in the directory `testcases/smoothing`. A simple, small example file named `demonstration_smoothing.in` is added that demonstrates all the aspects of your shading algorithm. This example input file will be used to test the validity of your code and may not run longer than 60 seconds on a laptop computer. You may change the scene, for example by adding objects or changing camera positions. If the **main** branch of the code does not work, the code from your own `smoothing` branch will be used instead.

2.3.2 Component 2. Material models

Implement more realistic material models, such as a model for a diffuse material, a shiny material and a reflecting material model. Some examples and background information of these models can be found in references [1, 2]. The material models in the current version of the code are so-called Lambert material models and are implemented with a very simple form of abstraction, whch allows you to use 100 different materials of the type Lambert Material. One of the first task in this component is to modify this data structure to store the data of different material models (reflection, glossy material, transparant material, etc.). Next, you implement the actual material models.

Tasks

- Develop and implement a data structure for multiple different material models.
- Implement a material model (e.g. BRDF) in which you can control the opaqueness of the material
- Implement a reflective material model that takes into account the reflection of the background image.

Deliverables

In this **individual report** you explain the theory behind the algorithms, present the pseudo code of your implementation and give implementation details. Briefly discuss the input file

commands (how to control the various options in the input file) and illustrate the performance by means of simple examples (a collection of spheres) in which you clearly demonstrate the various material models, see Figure 5.

You develop your code in the branch `materials` and make sure that this branch is merged with `main` branch. You add all the input-files of all the examples in the directory `testcases/materials`. A simple, small example file named `demonstration_materials.in` is added that demonstrates all the aspects of your material models. This example input file will be used to test the validity of your code and may not run longer than 60 seconds on a laptop computer. You may change the scene, for example by adding objects or changing camera positions. If the **main** branch of the code does not work, the code from your own `materials` branch will be used instead.

2.3.3 Component 3. Sampling and Alternative Camera models

The camera in the original ray tracing code is a so-called pin-hole camera. Besides that, the camera is hard coded, apart from the camera position and the Field of View, the orientation of the camera can not be modified. In this component, you are asked to implement a more realistic camera model, with a real lens model that allows you to set the focal length to create image with a more realistic in-depth focus. An overview of these camera models is given in Ch. 6 of [1].

The implementation of these models requires the implementation of an alternative sampling technique. In the current version, each pixel in the image is determined by a single ray of light, which leads to aliased images. Random sampling, in which multiple rays are used to determine the color of a pixel, gives a much better picture and enables the implementation of more sophisticated camera models.

Tasks

- Implement a camera that can be moved and rotated via parameters in the input file.
- Implement random sampling techniques.
- Implement a realistic camera model that has a specific focal length.

Deliverables

In the **individual report** you explain the theory behind the algorithms, present the pseudo code and explain the implementation details. Explain the input file commands (how to control the various options in the input file) and give examples (by means of figures) in which you clearly demonstrate the various setting of your camera model and the effect of different sampling techniques and conclusions and recommendations, see for example Figure 6.

You develop your code in the branch `camera` and make sure that this branch is merged with `main` branch. You add all the input-files of all the examples in the directory `testcases/camera`. A simple, small example file named `demonstration_camera.in` is added that demonstrates all the aspects of your camera models. This example input file will be used to test the validity of your code and may not run longer than 60 seconds on a laptop computer. You may change

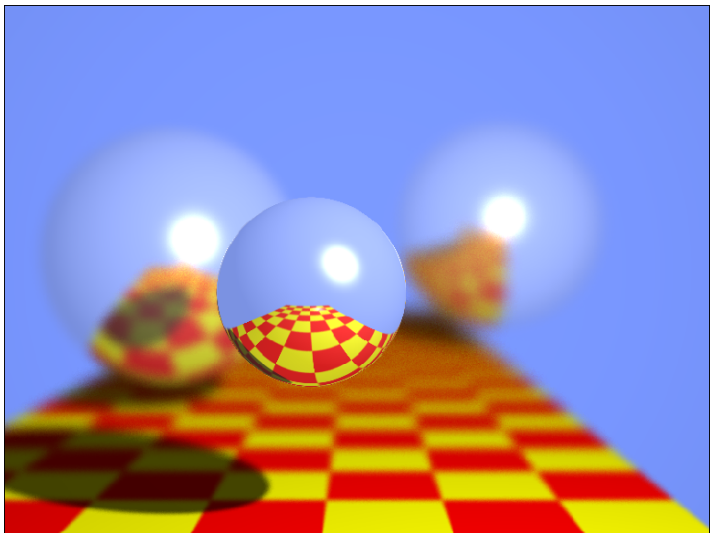


Figure 6: Demonstration of the effect of focus in a camera model.

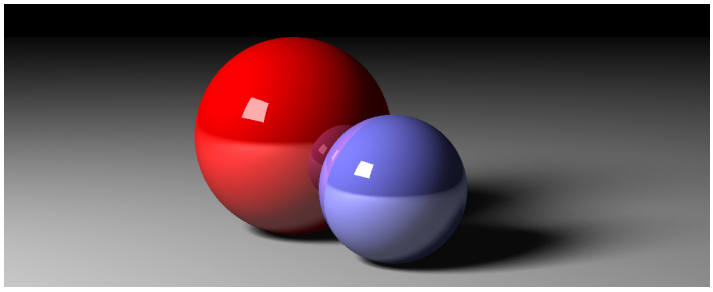


Figure 7: Demonstration of soft shadows due to surface lights.

the scene, for example by adding objects or changing camera positions. If the **main** branch of the code does not work, the code from your own `camera` branch will be used instead.

2.3.4 Component 4. Shadows and new light models

Shadows are an important aspect of photo-realistic rendering. In this component, you will implement shadows and alternative light sources, such as a spotlight, global illumination and area lights. For more information on these kind of models, one can read [2] and Chapter 12 of [1].

Tasks

- Implement an efficient algorithm to model shadows.
- Implement a spotlight.
- Make sure that all light sources can occur multiple times in a scene and can be set in the input file.

Deliverables

In the **individual report** you explain the theory behind the algorithms, a pseudo code of your implementation, the implementation details, the input file commands (how to control the various options in the input file). Give examples (by means of figures) in which you clearly demonstrate the various settings of your shadow and light model.

You develop your code in the branch `lights` and make sure that this branch is merged with `main` branch. You add all the input-files of all the examples in the directory `testcases/lights`. A simple, small example file named `demonstration_lights.in` is added that demonstrates all the aspects of your light models. This example input file will be used to test the validity of your code and may not run longer than 60 seconds on a laptop computer. You may change the scene, for example by adding objects or changing camera positions. If the **main** branch of the code does not work, the code from your own `lights` branch will be used instead.

2.3.5 Component 5: Code integration and optimisation

The more complex the scene is, the longer it takes to render it. Raytracing of triangular surfaces is an expensive operation where you have to determine the intersection of thousands of rays with thousands of triangles. This procedure can be improved significantly by neglecting triangles that are far away from the rays, or hidden by another object. In raytracing this technique is called intersection acceleration and is discussed in more detail in Chapter 4 of the book by Pharr et al. [1]. You can use this chapter as a starting point to implement your own, simple intersection accelerator. For this, you may have to impemented new data structures to store the mesh data. In addition, you can use parallel computing techniques to further accelerate your calculations.

Tasks

- Implement a hierarchical data structure for a fast analysis of the scene.
- Parallelise the code for even better performance using `openmp`.

Deliverables

In the **individual report** you explain the theory behind the algorithms, present the pseudo-code, discuss implementation details, the input file commands (how to control the various options in the input file), the performance for various settings (by means of graphs). You can use one of the BMW wheel examples to demonstrate the performance.

You develop your code in the branch `optimisation` and make sure that this branch is merged with `main` branch. You add all the input-files of all the examples in the directory `testcases/optimisation`. A simple, small example file named `demonstration_optimisation.in` is added that demonstrates all the aspects of your optimisation alogrithm. This example input file will be used to test the validity of your code and may not run longer than 60 seconds on a laptop computer. You may change the scene, for example by adding objects or changing camera positions. If the **main** branch of the code does not work, the code from your own `optimisation` branch will be used instead.

2.4 Group assignment

As a group, you need to carefully integrate all components in a single working code. The final version in the branch `main` will be asessed as the group product.

Deliverables

In the **group report** you explain the composition of the group, the contribution of each member to the total code (without the details of the various components) and the issues the group encountered in merging the code. In addition, you add a single image to highlight all features of the code (i.e. all aspects of the 5 components). Add a short description in which you explain how each component is represented in this image.

You add this input-files of all the examples in the directory `testcases/group`. A simple, small example file named `demonstration_group.in` is added that demonstrates the presence of all components in your code. This example input file will be used to test the validity of your code and may not run longer than 300 seconds on a laptop computer.

2.5 Deliverables

The project has four deliverables in total: (i) the actual source of the raytracer that you have developed with your group, (ii) a set of example input files to demonstrate the performance of the code, (iii) an individual report in which you explain the development of your component, (iv) a group report in which you explain the code integration

Deliverables (i) and (ii): the code and input files

The code in the *main* branch of your group on the April 21st 2025 is the version of the code that will be assessed.

- All examples and testcases must be executed with the same code. Hence, the code needs to be compiled only once and all instructions for the various scenes are given in the input file. Ideally, all examples can be executed as follows:

```
C:\raytracer.exe sample.in
```

- If your component cannot be demonstrated with the main code, we will checkout the code in the corresponding branch. However, some points will be deducted.
- All parts of the given code may be changed, if necessary. However, it is not allowed to use external libraries, other than the standard c libraries, such as `stdio.h` and `math.h`. It is also not allowed to copy paste third party software into your code. In case of doubt, ask the instructor for permission.
- Additional tools to create input files may be implemented in separated codes, which are also added to the git repository.
- Unit tests are added to test the most important functions and data types .
- All input files that demonstrate the functionality of your component, must be added to the directory testcases in the git repository.

Deliverable (iii): the individual report

In this document you report on the development of your own component, i.e.:

- A description of the numerical techniques.
- An overview of their implementation in the code, including pseudo code.
- A reference to a test to verify the implementation.
- If possible, a comparison of the performance with respect to the original code.
- A show case image in which the models are clearly demonstrated.
- A short conclusion and recommendation for this model.

The maximum length for each component is max. 6 pages, including figures. You must use the template `raytracer_indiv_report.tex`. This document must be uploaded to Canvas before April 21st 2025.

Deliverable (iv): the group report

This report contains the following

- the composition of the group
- the contribution of each member to the total code (without the details of the various components)
- the issues the group encountered in merging the code.
- a single image to highlight all features of the code (i.e. all aspects of the 5 components). Add a short description in which you explain how each component is represented.

The maximum length for this report is 3 pages, including figures. You must use the template `raytracer_group_report.tex`. This document must be uploaded to Canvas before April 21st 2025.

2.6 Final grade

The final grade G of this project is determined as follows:

$$G = 3/4 G_{\text{indiv}} + 1/4 G_{\text{group}}$$

where G_{indiv} is the indiviudal grade and G_{group} is the group grade (both on a scale of 10). The individual grade G_{indiv} is based on the assessment of the individual report (iii) and the corresponding source code (i) and input files (ii). The group grade G_{group} is based on the group report (iv) and the corresponding source code (i) and input files (ii).

References

[1] M. Pharr, W. Jakob, G. Humphreys (2016) *Physically Based Rendering: From Theory to Implementation*, 3rd Edition, Morgan Kaufmann. ISBN-13 978-0128006450.

[2] P. Shirly (2020) *Ray Tracing in One Weekend*
`https://raytracing.github.io/books/RayTracingInOneWeekend.html`

[3] Gouraud shading (Wikipedia page)
`https://en.wikipedia.org/wiki/Gouraud_shading`

[4] Phong shading (Wikipedia page)
`https://en.wikipedia.org/wiki/Phong_shading`

[5] Ray tracing (graphics) (Wikipedia page)
`https://en.wikipedia.org/wiki/Raytracing(graphics)`