

# Simulation of the Optical Path

## Computational Engineering Science

### Report

Pham Tien Dung  
Shunta Takushima  
Vu Thanh Mai  
Zhang Le



# Contents

<b>1</b>	<b>Project Description</b>	<b>2</b>
<b>2</b>	<b>Analysis of the Project</b>	<b>3</b>
2.1	Product Use Cases/User Requirements . . . . .	3
2.2	Technical Background . . . . .	4
2.2.1	Optical Components . . . . .	4
2.2.2	Data flow . . . . .	5
2.2.3	Choice of Optimizer . . . . .	5
2.3	System Requirements . . . . .	8
<b>3</b>	<b>Design</b>	<b>9</b>
3.1	Graphical User Interface . . . . .	9
3.2	Software Architecture . . . . .	10
3.2.1	Code Guidelines . . . . .	10
3.2.2	Folder structure . . . . .	10
3.3	Implementation . . . . .	11
3.3.1	Running the program from the source code . . . . .	16
<b>4</b>	<b>Room for Improvement: Post-processing Optimization</b>	<b>17</b>
<b>5</b>	<b>Project Management</b>	<b>19</b>
5.1	Time Management and Task Distribution . . . . .	19
5.2	Takeaway from This Project . . . . .	20

# Chapter 1

## Project Description

This document is a detailed report on *Simulation of Optical Path*, a software development project conducted at RWTH Aachen University.

With the help of fluorescence injected inside the subject of interest, which in our case is a mouse, the mouse will emit photons of different wavelengths. These photons will then go through a series of different optical components, including mirrors, filters, lenses, and in the end, an optical pixel-based detector. On this detector, we will have an image of the organs of the mouse.

In this project, we have two main goals. The first goal is to simulate the optical path of photons emitted from our subject of interest going through the optical components. The second goal is to optimize the quality of the image obtained on the detector by adjusting the characteristics of the optical components, including their position, angle, and so on.

Our software is written in C++. We use Qt to generate the GUI.

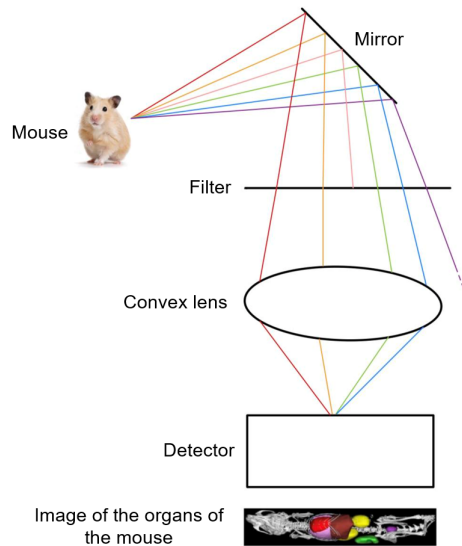


Figure 1.1: Overview of the optical system

# Chapter 2

## Analysis of the Project

### 2.1 Product Use Cases/User Requirements

- Use Case Diagram

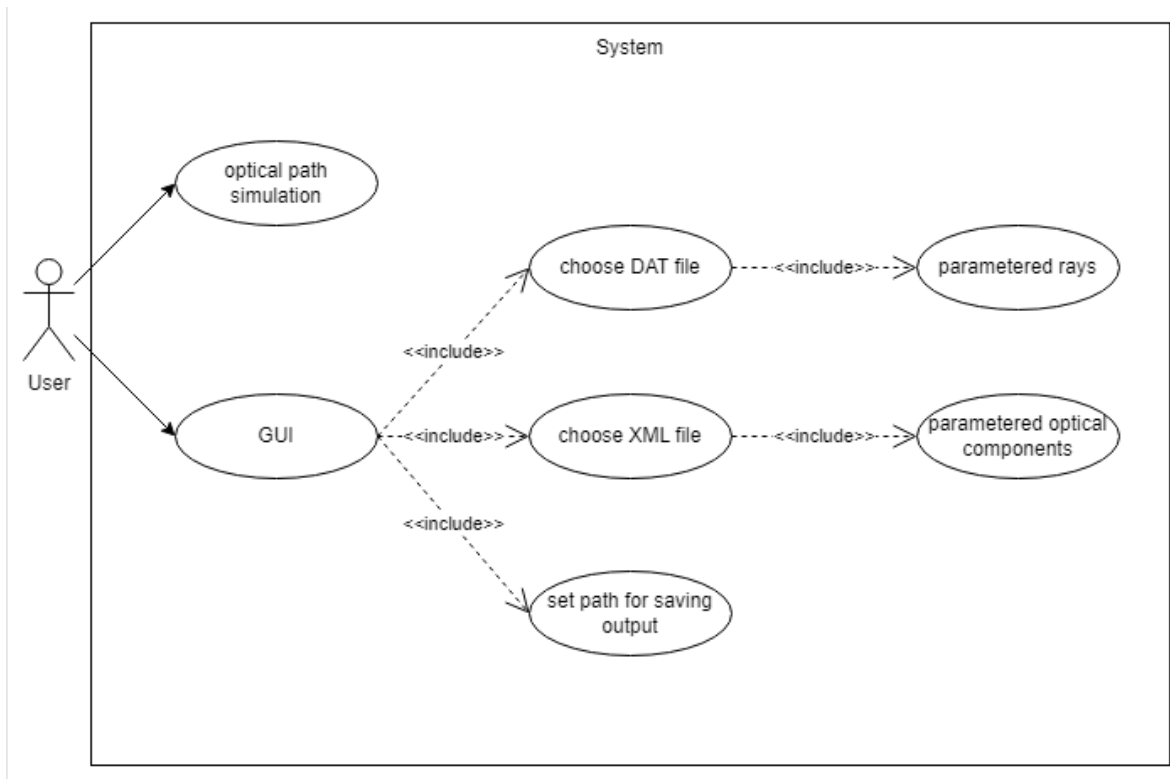


Figure 2.1: Use Case Diagram

The design of our software is to the fig. 2.1 above.

## 2.2 Technical Background

### 2.2.1 Optical Components

In this part, we will not be explaining the functionalities of each optical component separately, but rather how does each optical component contribute to the optical machine as a whole.

- **Mirror**

Similar to real-life applications of optical systems, the optical components in our project will stand on the same axis. The mirror helps redirect the light into this axis based on the law of reflection.

- **Lens**

The lens focuses or disperses light. While users have the freedom to simulate configurations with multiple lenses, we have streamlined the optimization process by allowing only a single lens to be adjusted. This decision is grounded in practicality: in real-world applications, lenses typically have a fixed focal length, and adjustments are made by moving the lens within a predefined range to optimize image quality, not by altering its focal length.

Multiple lens systems can be represented by a single effective lens, and thus implementing multiple lenses is similar to implementing only one lens with the focal length  $f$  calculated using this mathematical formula:

$$\frac{1}{f} = \frac{1}{f_1} + \frac{1}{f_2} - \frac{d}{f_1 \cdot f_2} \quad (2.1)$$

with  $f_1$  and  $f_2$  being the focal lengths for the two lenses respectively and  $d$  being the distance between the two lenses. When we perform optimization algorithms, we will assume that there is only one lens with the focal length calculated using the aforementioned formula for the sake of simplicity.

- **Filter**

We use a band-pass filter to pass photons from a specific wavelength range and filter out photons out of this range. Different organs of the mouse will emit photons with different wavelengths, and the role of the filter is to only capture important photons (e.g. photons emitted from the lungs).

The filter's bandwidth and radius can be set by the user. We also have to note that the bigger the filter is, the higher its cost will be. Therefore, it is important to correctly position the lenses to focus light on the filter so that we can keep its radius small.

- **Detector**

The detector is a 2-dimensional pixel grid. It will assign photons to the correct pixel on the image of the organs of the mouse.

### 2.2.2 Data flow

Input data are given into the program per XML files. They will contain the name of the physical components, their positions and further specific characteristics, such as focal length for lenses.

```

3  <OpticalComponents>
4    <Component name="Mirror">
5      <Surface type="Circle" diameter="4.0" />
6      <Position x="0.0" y="0.0" z="0.0" />
7      <Normal dx="0.0" dy="1.0" dz="1.0" />
8    </Component>
9    <Component name="Lens">
10     <Surface type="Circle" diameter="5.0" />
11     <Position x="0.0" y="0.0" z="2.9" />
12     <Normal dx="0.0" dy="0.0" dz="1.0" />
13     <Focallength value="2.0" />
14   </Component>
15   <Component name="Filter">
16     <Surface type="Circle" diameter="2.0" />
17     <Position x="0.0" y="0.0" z="5.0" />
18     <Normal dx="0.0" dy="0.0" dz="1.0" />
19     <WavelengthRange min="450.0" max="700.0" />
20   </Component>
21   <Component name="Detector">
22     <Surface type="Rectangle" width="1.5" height="1.5" />
23     <Position x="0.0" y="0.0" z="6.0" />
24     <Normal dx="0.0" dy="0.0" dz="1.0" />
25     <HeightDirection hx="0.0" hy="1.0" hz="0.0" />
26     <Pixels width="512" height="512" />
27   </Component>
28 </OpticalComponents>

```

Figure 2.2: Example of an input XML file for optical component properties

### 2.2.3 Choice of Optimizer

Our search for a suitable optimizer for the optical system is full of trial and error.

At first, we decided to go with the common gradient-based optimizer ***Gradient Descent***. We then encounter this problem: what if the function being used for optimization is not convex? Gradient Descent's assumption of the optimized function being convex is too strong for a requirement, which makes it unsuitable for real-life applications.

Being aware of the drawback of Gradient Descent, we decided to implement a stochastic-based optimizer called ***Particle Swarm Optimization***, or in short, PSO. PSO also takes the same approach as Gradient Descent when in each iteration, it tries to move all variables to locally minimize the function, but in this case, using randomization and trial-and-error. However, the mathematical background behind Particle Swarm Optimization is too complex to the extent that to be able to implement it within the time constraint, we have to simplify the algorithm to the point where the accuracy is not acceptable. After many unsuccessful attempts at adjusting and refining the algorithm, we decided to move on to find a new optimizer.

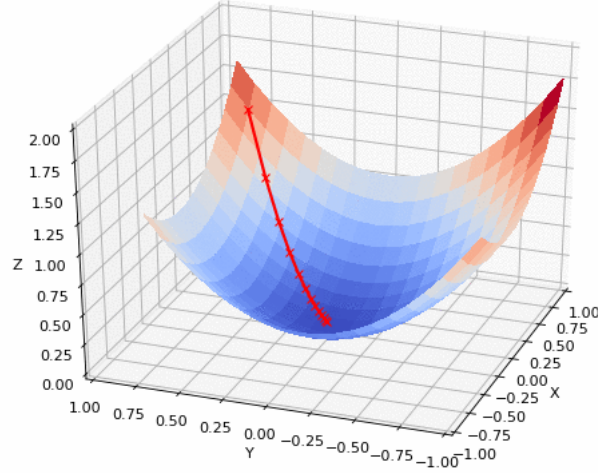


Figure 2.3: An example of Gradient Descent applied for a convex function

On our third attempt, we decided to go in a whole different route. Instead of thinking about optimization as a mathematical problem, we see optimization in a more real-life-application-based approach: how to improve the quality of a fluorescence image? We found out that for a fluorescence image which appears to have a lot of scattered points, **sharpness** is the most important quality. To calculate the sharpness of our image, we use the ***Variance Measure*** method. This method computes the sum of the squared differences between each pixel and the mean image intensity:

$$\text{Sharpness} = (i_1 - i_m)^2 + (i_2 - i_m)^2 + (i_3 - i_m)^2 + \dots \quad (2.2)$$

with  $i_n$  being the intensity of the  $n^{\text{th}}$  pixel and  $i_m$  being the mean image intensity. High variance aligns with clear, contrasting features, while low variance aligns with blurred, homogeneous regions.

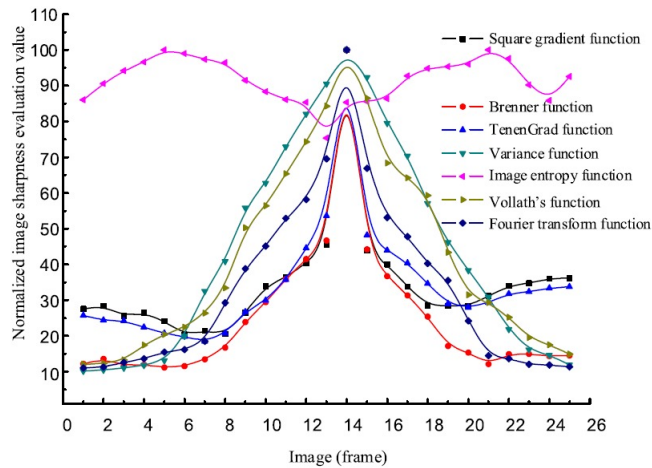


Figure 6. Sharpness evaluation function curve.

Figure 2.4: Sharpness evaluation function curve

So why is Variance Measure suitable for quantifying the sharpness of a fluorescence image? Based on Formula 2.2, we can see that regions of higher sharpness should contain mostly pixels of either high or low intensity, while regions of lower sharpness tends to be constituted of more pixels of intensity closer to the mean (neither high nor low). This perfectly matches the nature of sharpness in fluorescence imaging:

1. *Sharp, in-focus areas* are characterized by high contrast due to the nature of fluorescence. The fluorescent markers, by design, emit light when excited, yielding bright spots against a darker backdrop. This results in regions predominantly composed of high- or low-intensity pixels, leading to higher variance in these zones.
2. *Out-of-focus areas* display a diffusion of fluorescent light, which diminishes the contrast between luminous and dim areas. The intensity values in these regions tend to cluster around the mean, lacking the extremes that signify sharp regions. Consequently, the intensity value range is compressed, manifesting as a reduced variance.

After deep research into the practical application of optical machines, we learn that **lens** has the largest influence on the sharpness of the picture. The better the lens can focus light on the detector, the better the sharpness of the picture will be. As a result, to maximize the sharpness of the picture, we decided to keep the mirrors, filters and detector at constant positions and only move the lens in a predefined range. With this approach towards optimization, we later obtain a picture with way superior quality in comparison with Gradient Descent and Particle Swarm Optimization.

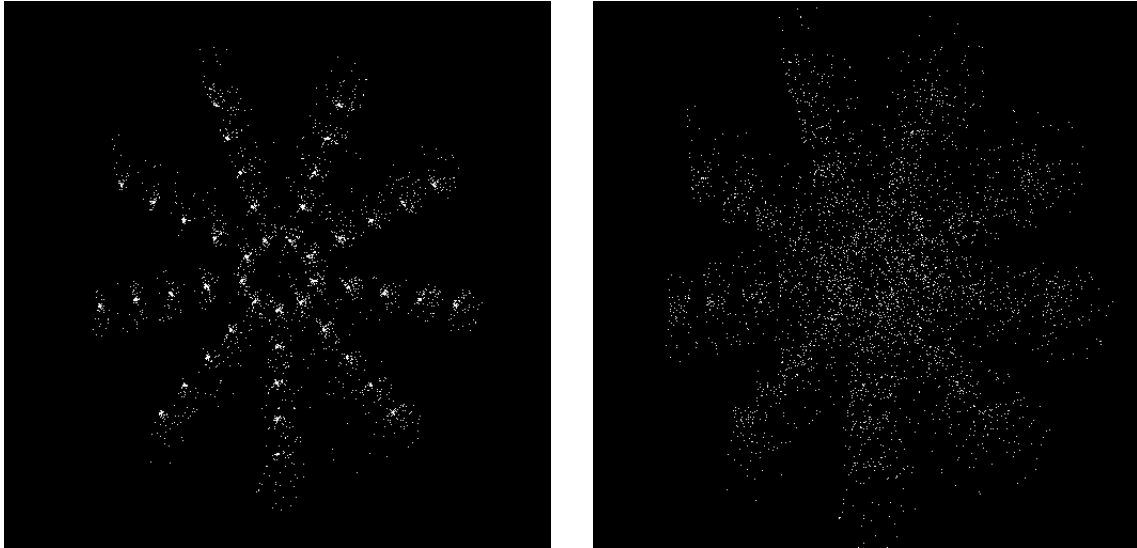


Figure 2.5: Simulation with 45 light sources, 100 rays from each light source with and without Sharpness Maximization with Variance Measure



## 2.3 System Requirements

Table 2.1 presents a comprehensive list of both functional and non-functional requirements that have been meticulously defined for our project. The configuration and attributes of optical components are efficiently managed through XML files for seamless reading and storage. Our simulation codes are written in C++ and parallelized using OpenMP to ensure high-performance computation. For an intuitive user experience, we have integrated Qt to create a user-friendly Graphical User Interface. Additionally, Doxygen is employed to generate a well-structured and detailed documentation. This ensemble of technologies and practices signifies our commitment to delivering a well-executed and professional simulation project.

Functional requirements	Nonfunctional requirements
Reading data input that describes the properties of optical components	XML file
Reading data input that describes the distribution of rays	DAT file
Simulating optical paths	C++, OpenMP
Creating a final virtual image	Gnuplot
Optimizing the quality of the virtual image	Maximizing the picture's sharpness using Variance Measure and optimally positioning the lens
Creating a simple graphical user interface (GUI)	Qt
Documentation	Doxygen

Table 2.1: System Requirements

# Chapter 3

## Design

### 3.1 Graphical User Interface

We aim to design a user-friendly Graphical User Interface with Qt with the following traits:

- Simple and organized design
- Display a 3D plot of the simulation using Gnuplot
- Display a 2D image of the picture on the detector

To achieve this, we created a GUI without any window transitions. All buttons are included in one relatively small window to minimize mouse cursor movement.

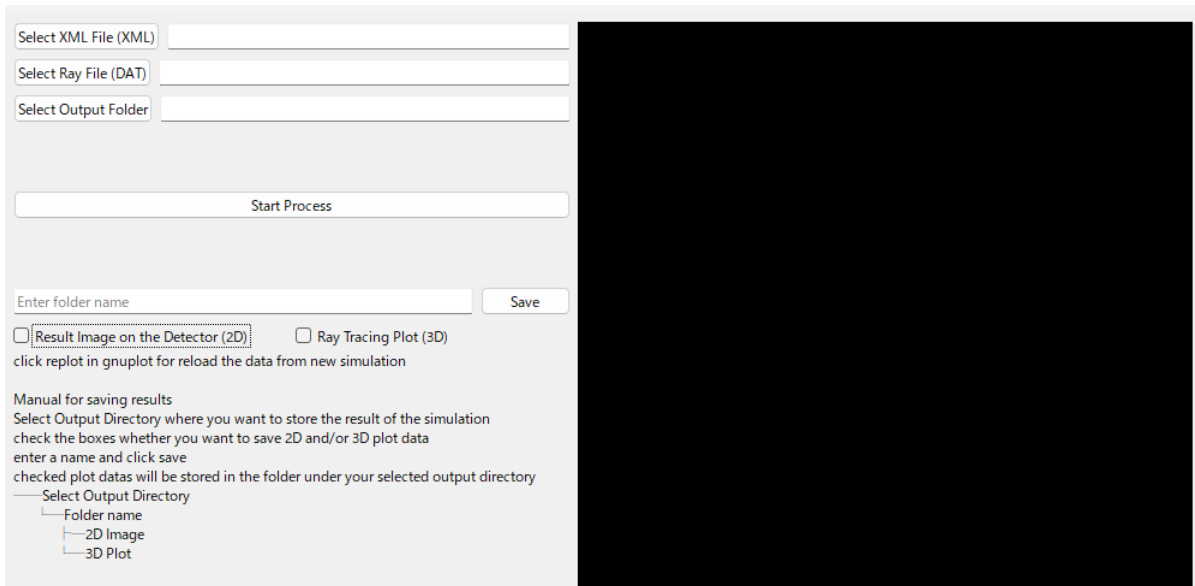


Figure 3.1: Window of GUI

A 2D image of the picture on the detector and a 3D Ray Tracing Plot of the optical path are going to be generated after the simulation. Users can rename and save the two aforementioned resulting images.

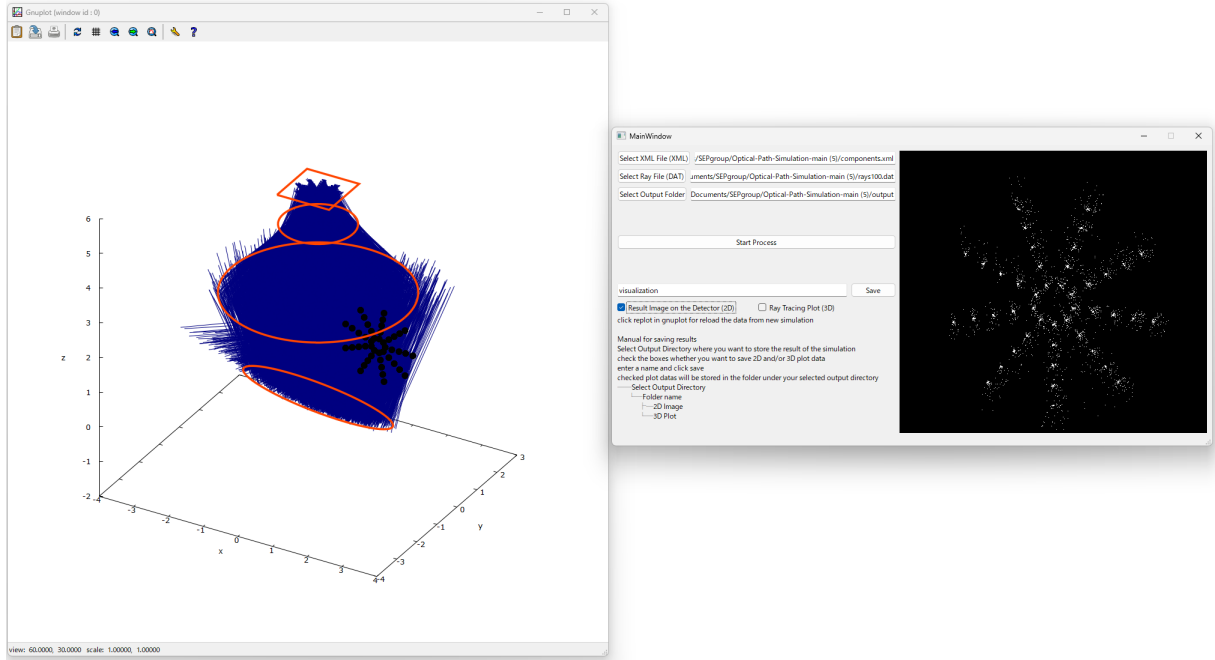


Figure 3.2: Generation of a 2D and 3D image after the simulation

## 3.2 Software Architecture

### 3.2.1 Code Guidelines

In the beginning of the project, we all agreed on these two code guidelines for seamless cooperation with each other:

- Naming conventions for classes, variables and functions.
- Paying close attention to independent loop structure, private and shared variables as well as avoiding data races so that parallelization with OpenMP can later be used to enhance the performance of the program.

### 3.2.2 Folder structure

All directories of our project are described in Figure 3.2 below:

- **thirdParty:** including **Eigen** library for efficient linear algebra, **tinyxml2** for reading XML input files and **stb\_image\_write** for writing **png** output images
- **simulator:** containing all files for optical path simulation and image optimization
- **simulator/opticalComponents:** containing files for optical components
- **simulator/opticalSurfaces:** contains files for different optical surfaces (right now we have implemented planar circular and planar rectangular surfaces)
- **simulator/optimizer:** containing files for optimizing the quality of the image on the detector

- **visualization:** containing files for visualization
- **data:** containing initial positions of components and photons and their parsers (a.k.a readers)
- **docs:** containing documentation of this project
- **gui:** containing files for a Graphical User Interface (GUI) that provides a window for users to interact with

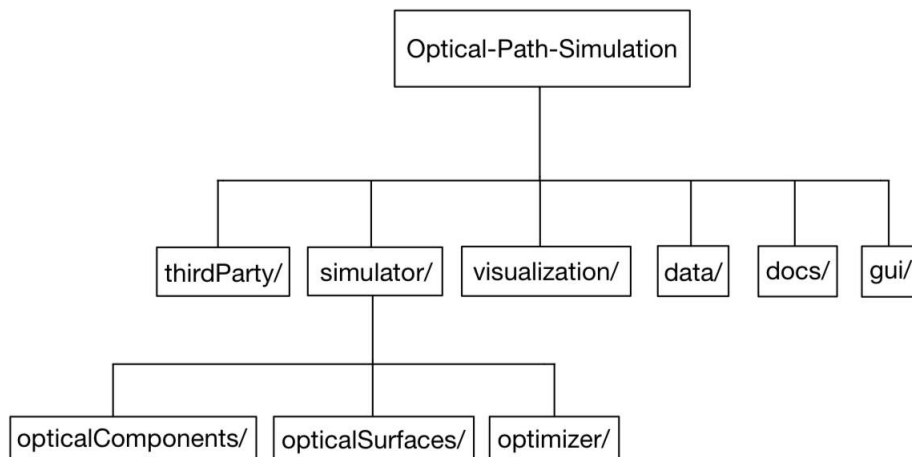


Figure 3.3: Folder structure

### 3.3 Implementation

In this section, we will be explaining how do we implement the simulation of photon path and optical components in our code.

First and foremost, it is important to note that for all of our classes and functions, we are working with the **Eigen library** as well as **templates** for the sake of flexibility and efficiency. Type aliasing can be seen in Figure 3.4 below, which allows cleaner syntax and more readable code.

```
1  #include "../thirdParty/eigen-3.4.0/Eigen/Dense"
2
3  template<typename T>
4  using vec3 = Eigen::Matrix<T, 3, 1>;
```

Figure 3.4: Type aliasing

Now, let us begin with the simulation of the photon path. In our program, we do not view photons as single points, but rather as **rays**, which are narrow beams of light travelling in a straight line from their place of origin. We find this definition with rays

easier for the conception of light propagation in optical path simulation. From now on, in this section, we will be replacing the name *photon* with *ray*.

For the simulation of rays, we have two files: **ray.h** and **rayTracing.h**.

1. **ray.h**: In this file, we define the **Ray** class, which encapsulates the properties of a ray of light, including its position, direction, wavelength, and intensity. The direction vector is maintained as a normalized vector to ensure an accurate representation of the ray's path.
2. **rayTracing.h**: We implement the simulation of ray propagation in this file. To begin with, we will be going through all rays in the XML file in a for loop. For each ray, we will check if the ray intersects with the next optical component. If an intersection happens, we will further propagate the ray in the optical system by calling the **handleLight** function, which describes how that particular optical component interacts with the ray. If the ray does not intersect with the next optical component, it will continue going in its current path. All intersection points of rays and optical components are stored in a file named **raysFile**. The simulation of rays is illustrated in Figure 3.5 below.

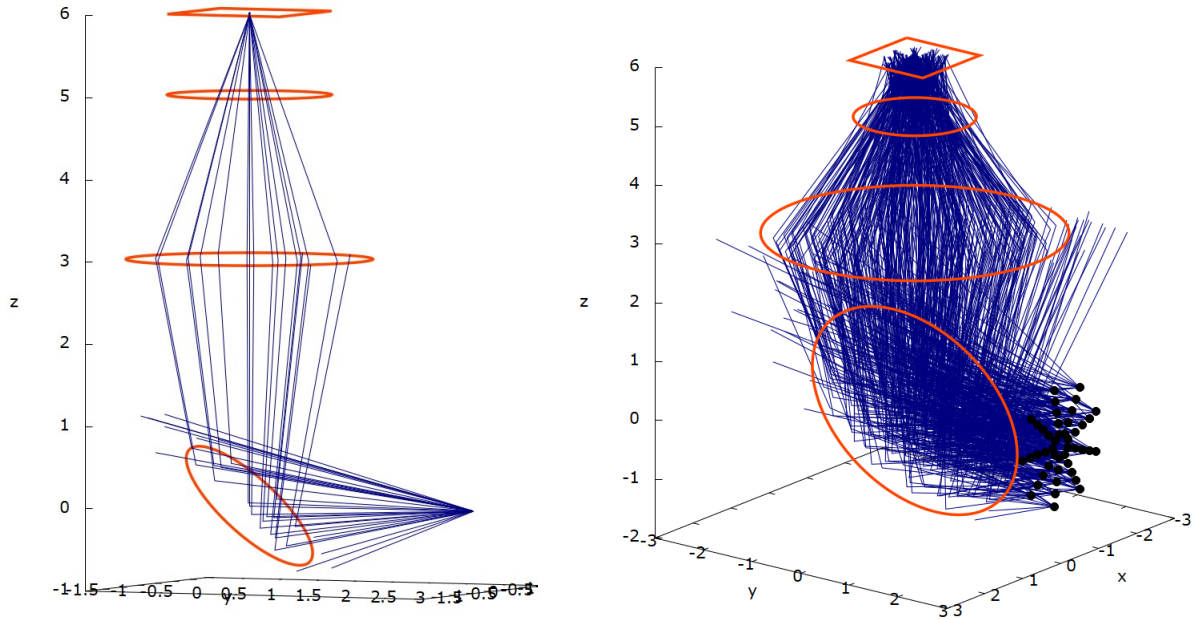


Figure 3.5: Simulation of 1 and 45 light source(s) with 10 rays coming out of each light source

Now we will continue with the implementation of optical components in our code.

Since the beginning of the project, we have been fully aware that to be able to implement a project of this magnitude of complexity with a lot of different optical components, the utilization of abstract classes and virtual functions is crucial. Our first step towards the

implementation of optical components is to find out what functionalities do all these components have in common and group them in an abstract class. The first idea that came into our mind was that every component has a *surface*, and this surface varies in shape and interactions with rays from component to component. This leads to the creation of the abstract class **OpticalSurface**:

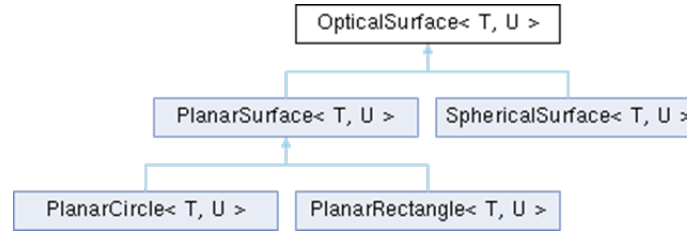


Figure 3.6: Class hierarchy 1

In the class `OpticalSurface`, we have two protected variables: **position** and **normal**, which are shared by all optical components. We then implement getter and setter functions to interact with the position and the normal. After that, we declare two pure virtual functions:

- **getIntersectionPoint**: returns the intersection point of the ray and the optical component.
- **intersects**: returns a pair of a boolean variable and a 3-dimensional vector describing whether a ray intersects with the optical component and the intersection point.

```

10 template<typename T, typename U>
11 class OpticalSurface {
12 protected:
13     vec3<T> position; // position or center of the optical surface
14     vec3<T> normal; // normal or principal axis of the optical surface
15
16 public:
17     OpticalSurface(const vec3<T>& position, const vec3<T>& normal) : position(position), normal(normal.normalized()) {}
18     virtual ~OpticalSurface() = default;
19
20     vec3<T> getPosition() { return position; }
21     const vec3<T>& getNormal() const { return normal; }
22     void setPosition(const vec3<T>& newPosition) { position = newPosition; }
23
24     virtual vec3<T> getIntersectionPoint(const Ray<T, U>& ray) const = 0; // of a ray with the optical surface
25     virtual std::pair<bool, vec3<T>> intersects(const Ray<T, U>& ray) const = 0; // if intersection and the intersection point
26     virtual void generatePoints(std::ofstream& outFile) const = 0; // points on the surface for visualization
27 };
  
```

Figure 3.7: Implementation of OpticalSurface

Two classes are derived from the abstract class `OpticalSurface`: **PlanarSurface** and **SphericalSurface**. The class `PlanarSurface` imitates surfaces of planar shape, while the class `SphericalSurface` simulates surfaces of spherical shape,

1. For the class `PlanarSurface`, we only implement *getIntersectionPoint*, while leaving *intersects* undefined because the method for checking whether a ray intersects with a surface or not is different depending on whether the surface is circular or rectangular. As a result, 2 concrete classes derive from `PlanarSurface`: *PlanarCircle* (for mirrors, lenses and filters) and *PlanarRectangle* (for the detector).

2. For the class `SphericalSurface`, we implement both *getIntersectionPoint* and *intersects*. `SphericalSurface` is meant for the implementation of more complex surfaces (e.g. concave mirrors or thick lenses), where parameters of the surface curvature and the material used and law of refraction are needed for accurate simulation.

All optical components derive from the same abstract class **OpticalComponent**. The class hierarchy is illustrated in Figure 3.6 below:

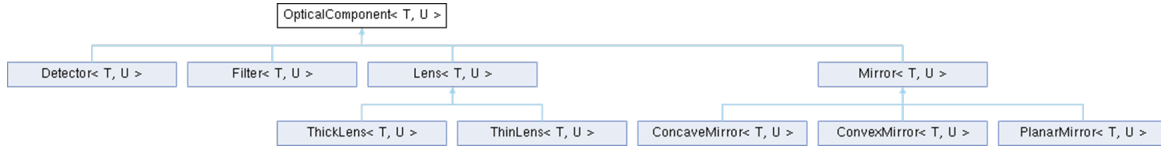


Figure 3.8: Class hierarchy 2

In the class `OpticalComponent`, we have a protected pointer named **surface** of class `OpticalSurface`. We later utilize this pointer so that all optical components fully inherit the functionalities of `OpticalSurface`: interactions with *position* and *normal*, as well as the two functions *getIntersectionPoint* and *intersects*. In `OpticalComponent`, we also declare a pure virtual function named **handleLight**, which describes how does the optical component interact with rays. This function is later implemented based on the physical background of the optical component that derives from `OpticalComponent`.

```

9  enum class OpticalComponentType {
10      Mirror,
11      Lens,
12      Filter,
13      Detector
14  };
15
16  template<typename T, typename U>
17  class OpticalComponent {
18  protected:
19      std::unique_ptr<OpticalSurface<T, U>> surface;
20      OpticalComponentType type;
21
22  public:
23      OpticalComponent(std::unique_ptr<OpticalSurface<T, U>> surface, OpticalComponentType type)
24          : surface(std::move(surface)), type(type) {}
25      virtual ~OpticalComponent() = default;
26
27      const vec3<T>& getPosition() const { return surface->getPosition(); }
28      const vec3<T>& getNormal() const { return surface->getNormal(); }
29
30      OpticalSurface<T, U>* getSurfacePtr() { return surface.get(); }
31      OpticalComponentType getType() const { return type; }
32
33      virtual void handleLight(Ray<T, U>& ray, const vec3<T>& intersectionPoint) = 0;
34  };

```

Figure 3.9: Implementation of `OpticalComponent`

We still make the classes **Lens** and **Mirror** abstract with the pure virtual function *handleLight*, so that we can implement lenses and mirrors of different shapes and functionalities: thick lenses, thin lenses, planar mirrors and spherical mirrors, which include concave mirrors and convex mirrors. The *handleLight* function in the class **Filter** works just as intended, only allowing rays from a specific range of wavelength to pass through.

In the class **Detector**, the `handleLight` function calls another function named *mapToPixelGrid*, which maps the ray that intersects with the detector to a corresponding pixel on the grid. We also include the function *adjustSurfaceToFitPixelGrid*, which rescales the detector to match the ratio of the pixel grid.

```

19 template<typename T, typename U>
20 class Detector : public OpticalComponent<T, U> {
21 private:
22     int pixelWidth, pixelHeight; // dimensions of the pixel grid
23     Eigen::Matrix<U, Eigen::Dynamic, Eigen::Dynamic> pixelGrid; // store the accumulated light intensity
24
25     void adjustSurfaceToFitPixelGrid(); // adjust the detector's surface size to match the aspect ratio of the pixel grid
26     std::pair<int, int> mapToPixelGrid(const vec3<T>& intersectionPoint) const; // maps the intersection point to a pixel
27
28 public:
29     Detector(std::unique_ptr<PlanarSurface<T, U>> surface, int pixelWidth = 512, int pixelHeight = 512)
30         : OpticalComponent<T, U>(std::move(surface), OpticalComponentType::Detector),
31           pixelWidth(pixelWidth),
32           pixelHeight(pixelHeight),
33           pixelGrid(Eigen::Matrix<U, Eigen::Dynamic, Eigen::Dynamic>::Zero(pixelHeight, pixelWidth))
34     {
35         adjustSurfaceToFitPixelGrid();
36     }
37
38
39
40     const Eigen::Matrix<U, Eigen::Dynamic, Eigen::Dynamic>& getPixelGrid() const { return pixelGrid; }
41
42     void handleLight(Ray<T, U>& ray, const vec3<T>& intersectionPoint) override; // handle incoming light
43     void resetPixelGrid() { pixelGrid.setZero(pixelHeight, pixelWidth); }
44 };

```

Figure 3.10: Implementation of Detector

Thanks to the effective implementation of abstract classes and virtual functions, as well as the implementation of **Eigen** library for matrix and vector operations instead of traditional `std::vector` data type, each of our optical component classes only contains 3 to 4 functions despite the complexity of simulating optical paths. We also make the program completely **type generic**, which means that the program is now easily utilizable in different research fields since it can adapt to any precision requirement, which is particularly beneficial for high-fidelity scientific applications where precision is paramount. Parallelization with **OpenMP** is also implemented to enhance the performance of our program further. Last but not least, we have made the **Makefile** to automate the build process so that the user can directly get to the result from the GUI.

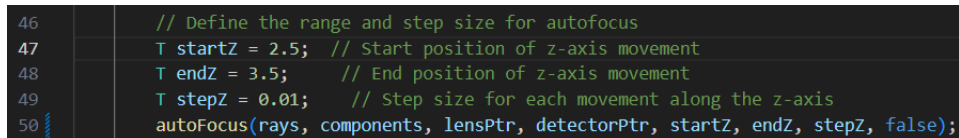
In our code architecture, we've established a hierarchy of abstraction that caters to both simplicity and complexity in optical simulations. The `PlanarSurface` class serves as the foundation for simulating basic optical elements, while the `SphericalSurface` class extends this functionality to model sophisticated components like convex and concave mirrors. For more nuanced simulations that account for material properties and the laws of refraction, we've introduced the `ThickLens` class, which incorporates the radius of curvature in its calculations. This versatile design paves the way for simulating a plethora of optical systems ranging from simple to more complex ones. The scalable nature of our project's architecture ensures that it can be readily extended to cover different optical configurations and applications (e.g. in Canon cameras, telescopes, microscopes, and more).



### 3.3.1 Running the program from the source code

In this section, we will briefly explain how to run our program from the source code without the Graphical User Interface. This process consists of four steps:

1. Download the required Eigen library from STCE RWTH Aachen and add it to the thirdParty folder (since these files are huge and Github does not allow to upload large files): <https://www.stce.rwth-aachen.de/teaching/cppnum-library>
2. Choose whether to run the program with or without autoFocus (name of the optimizer) in main.cpp by setting the input into autoFocus as true or false



```
46 // Define the range and step size for autofocus
47 T startZ = 2.5; // Start position of z-axis movement
48 T endZ = 3.5;   // End position of z-axis movement
49 T stepZ = 0.01; // Step size for each movement along the z-axis
50 autofocus(rays, components, lensPtr, detectorPtr, startZ, endZ, stepZ, false);
```

Figure 3.11: Running the program without autoFocus (the optimizer)

3. make run to run the whole program
4. make clean to delete the generated files

# Chapter 4

## Room for Improvement: Post-processing Optimization

In this chapter, we will briefly discuss one feature that we have not been able to implement in our program due to time constraints: **Post-processing Optimization**.

We do not want to approach this project in a mathematics- and programming-based manner. We tried to conduct research on how optical machines are implemented in real life and see if we can bring any of these features into our program. The deeper we dug into the research of fluorescence imaging, the more we realised that due to the nature of the low quality of fluorescence images, Post-processing Optimization is crucial to successfully extract useful information from the picture on the detector. Post-processing Optimization means that after we have correctly positioned the lens to optimize the quality of the image, we then apply further algorithms on the image to make its interesting components, the so-called **Regions of Interest**, more visible.

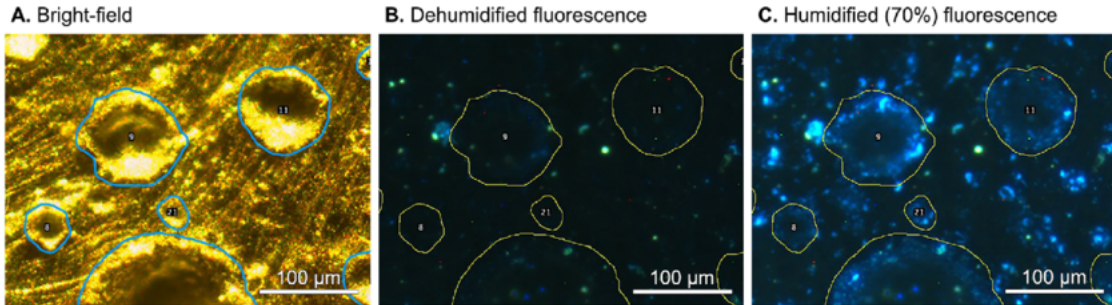


Figure 4.1: [Examples of Regions of Interest in fluorescence imaging](#)

Regions of Interest might include:

- Pixels with high fluorescence intensity (high brightness)
- Regions with high dynamic range (containing both pixels of low brightness and high brightness)

The following methods can be integrated into our program to highlight the Regions of Interest in the image on the detector:

1. **Contrast Enhancement:** By implementing Contrast Enhancement, we will manipulate the pixel values of an image to improve the visual perception of details by increasing the distinction between intensity levels. This might involve:
  - **Histogram Equalization:** Histogram Equalization enhances the global contrast of an image by redistributing pixel intensities to cover the entire dynamic range.
  - **Contrast Stretching:** Contrast stretching amplifies the differences in intensity by uniformly scaling the pixel values. This method is particularly effective when the original image has a limited intensity range, and stretching helps to utilize the full dynamic range.
2. **Smothering the image with Linear Filtering:** Linear Filters replace each pixel with a linear combination of other pixels. This method is particularly useful when we want to average out the noisy variations in a noisy image, resulting in a resulting picture that is considerably smoother. It is also important to note that the larger the filter becomes (e.g. replacing each pixel with a linear combination of its 24 neighbouring pixels in a 5x5 filter in comparison with a 3x3 filter), the more details we are going to lose in our image along with noise, which makes the resulting picture more blurry. As a result, if noise reduction is our primary goal, it is more advisable to choose the smallest filter that gives an acceptable result.

# Chapter 5

## Project Management

### 5.1 Time Management and Task Distribution

Since this was our first project to work as a group on a huge software creation with deadlines and presentations, we had many problems with time management.

In the beginning of the project, we agreed to have an online meeting every week aside from the exam phase, which did not work as assumed. We began to work later than planned and had many unexpected struggles as we dug deeper into research and software implementation. We could easily have avoided these unnecessary issues if we had been consistent with our weekly meetups so that we would not have to hurry to finish the project before the deadline.

The following timetable can be roughly accounted for how we approached the project in chronological order:

<b>Milestone 1 (27/06 - 11/07):</b>	building the backbone structure for the project, researching different data structures and algorithms, defining subtasks and planning work accordingly.
<b>Milestone 2 (22/07 - 05/09):</b>	trials and errors with the implementation of optical components and photon path simulation, including making files to link everything together and integration testing.
<b>Milestone 3 (15/09 - 10/11):</b>	further refining the simulator, trials and errors different choices of optimizer, integration of simulator and optimizer.
<b>Milestone 4 (20/11 - deadline):</b>	code review, debugging, final testing and validation, preparing user documentation, project delivery and report.

Table 5.1: Time Management

We are extremely grateful for the advice that we received from our supervisor, Dr. Gremse. Even though we could not arrange as many meetings with Dr Gremse as we

wanted to, the help from an actual researcher gave us a clear path towards how we have to approach the project as a whole. Stepping into the implementation of software with an actual purpose was much harder than we thought, and without his help, we would have struggled more with having a clear overview of the project.

We all agreed on task distribution since the beginning. Vu Thanh Mai and Pham Tien Dung will be responsible for the implementation of the simulator and the optimizer, while Shunta Takushima and Zhang Le will be implementing the Graphical User Interface and the report. During the project, we all helped each other a lot whenever struggles arose with the task of a team member.

Pham Tien Dung	Optimizer, Report
Shunta Takushima	Report, Qt
Vu Thanh Mai	Simulator, Optimizer, XML reader
Zhang Le	Qt

Table 5.2: Task Distribution

## 5.2 Takeaway from This Project

This 5-month project was an invaluable opportunity for us to get a better understanding of software development and implementation. We learnt the importance of having to conduct proper research to deeply understand what we were working with before diving into implementation with coding. Having to implement the entire project in C++ also helped us understand this programming language better. C++ is indeed an amazing programming language since it provides abstract classes, virtual functions and very efficient memory management, which greatly reduces the complexity of software implementation.

Apart from the technical aspects, we also learnt to work as a team. In the end, no one in our group was an expert in any particular field, and it came to the time that we had to help and explain to the other team members what we were doing that helped us the most with understanding the project better.

# List of Figures

1.1	Overview of the optical system . . . . .	2
2.1	Use Case Diagram . . . . .	3
2.2	Example of an input XML file for optical component properties . . . . .	5
2.3	An example of Gradient Descent applied for a convex function . . . . .	6
2.4	Sharpness evaluation function curve . . . . .	6
2.5	Simulation with 45 light sources, 100 rays from each light source with and without Sharpness Maximization with Variance Measure . . . . .	7
3.1	Window of GUI . . . . .	9
3.2	Generation of a 2D and 3D image after the simulation . . . . .	10
3.3	Folder structure . . . . .	11
3.4	Type aliasing . . . . .	11
3.5	Simulation of 1 and 45 light source(s) with 10 rays coming out of each light source . . . . .	12
3.6	Class hierarchy 1 . . . . .	13
3.7	Implementation of OpticalSurface . . . . .	13
3.8	Class hierarchy 2 . . . . .	14
3.9	Implementation of OpticalComponent . . . . .	14
3.10	Implementation of Detector . . . . .	15
3.11	Running the program without autoFocus (the optimizer) . . . . .	16
4.1	Examples of Regions of Interest in fluorescence imaging . . . . .	17

# List of Tables

2.1	System Requirements . . . . .	8
5.1	Time Management . . . . .	19
5.2	Task Distribution . . . . .	20