FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT • DEPARTMENT INFORMATIK

**Lehrstuhl für Informatik 10 (Systemsimulation)**

# Optimization of Mirrorshapes in Optically Pumped Solar Lasers Using Ray Tracing Simulation Techniques

Matthias König

Master's Thesis

# Optimization of Mirrorshapes in Optically Pumped Solar Lasers Using Ray Tracing Simulation Techniques

Matthias König

Master's Thesis

| | |
|---|---|
| Aufgabensteller: | Prof. Dr. C. Pflaum |
| Betreuer: | 1.11.2021 – 2.5.2022 |
| Bearbeitungszeitraum: | |

**Abstract**

This work showcases the application of ray tracing techniques for the calculation of absorption profiles in optically pumped solar lasers. It aims at using a lightweight and fast physically based raytracer combined with a biobjective mesh adaptive direct search algorithm to optimize total power absorption and to minimize variance across the crystal. An exemplatory setup of a side pumped Nd:Yag solar laser was simulated, optimized and the resulting beam quality evaluated.

**Erklärung:**

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Systemsimulation (Informatik 10), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Master's Thesis einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 30. März 2022 ......................................

# Contents

## 1 Introduction

# 2 Methodology

# 3 Lasers

## 3.1 Solar Lasers

# 4 Raytracing Framework

With the advent of cheap processors and increasingly powerful consumer hardware, ray tracing has become more popular in recent years. For the purpose of global illumination in video games and image processing, more advanced techniques have been continuously developed and improved. In optical design ray tracing is used to analyse the imaging quality of optical systems or as in this work other illumination properties can be simulated. The need for fast refresh rates in video games and the requirement of modelling more complex physical phenomena in optical design have led to tracing and sampling techniques that reduce the computational expense dramatically with minimal loss of accuracy. Focused on the specific problems of laser design, these improvements make it possible to get physically accurate results in an acceptable amount of computational time in an iterative context.

As in optical design systems are mostly rotationally symmetrical, the framework is meant to be used in a two dimensional setup and calculated quantities, e.g. absorbed power in a medium converted to three dimensional values after a simulation step. This significantly reduces the amount of rays needed to avoid undersampling effects and to produce stable results across multiple simulation runs. Intersection tests also require less computation and objects in the scene require less fundamental shapes to test a ray against. The resulting performance gains makes it possible to run the simulation thousands of times in an iterative process to optimize some parameters in the optical setup even on consumer grade hardware. The objects in a scene are preproccessed to group fundamental shapes into leaves of a quadtree to reduce the amount of shapes a ray has to be tested against even further. To achieve the satisfied accuracy and to reduce noise the appropiate sampling strategies have to be used for a given problem. The most important techniques are provided including uniform sampling, stratified and importance sampling.

The framework was designed to provide a simple yet powerful interface for the user and was implemented in C++17. It provides the necessary data structures and algorithms for a fast raytracing solution. The sampling techniques are implemented in specialized classes of abstract interfaces. They can also be used by the user to implement custom techniques. The framework extensively relies on lambda functions to be provided by the user and thus naturally is customizable, although some preset functions are also provided. Because the calculations in the framework are so similar to applications in graphics software the OpenGL Mathematics header only library GLM [1] was used as an underlying maths library. GLM is based on the OpenGL Shading Language (GLSL) and so in a potential later step the framework could be ported to work on graphics cards providing that the data structures are changed to be accessable from a GPU. As in the specific problem in this work the tracing of each ray has side effects on the scene and on itself, i.e. the absorbed power of each ray has to be accumulated, it was decided to focus more on single core performance first and leave the parallel execution and execution on GPUs for a later point. Furthermore IO utilities for simulations are provided for Comma Separated Values (CSV) files and structured output for the commonly used Visualization Toolkit (VTK) [2].

In the following chapters the applied ray tracing techniques explained in detail and the structure of the developed framework is presented with code samples.

## 4.1 Raytracing Basics

Rays are represented as a parametric line from a ray origin $o$ in direction $d$. The parameter $t$ goes is in the interval $[0, \infty)$ and represents the closeness of the ray to the origin. The mathematical representation therefore is given as

$$\mathbf{r}(t) = \mathbf{o} + t\mathbf{d} \tag{1}$$

After the ray is generated it is tested against intersections with the scene. Here the smallest $t > 0$ of all the intersections with objects has to be found. The question if a ray intersects an object can usually only be answered for simple fundamental shapes, e.g. lines, circles, axis aligned bounding boxes (AABBs) in 2D or planes, triangles, spheres, etc. in 3D. Therefore objects are normally comprised of a collection of fundamental shapes and an intersection occurs if one of the fundamental shapes is intersected. Naturally, an object can be intersected multiple times by the same ray and so the results have to be searched for the smallest $t$. Each fundamental shape should be represented in a parametrised form so the intersection test can be represented as a system of equations. The two fundamental shapes used in this work are 2D lines and axis aligned bounding boxes (AABBs).

Lines are represented by two points $\mathbf{a}$ and $\mathbf{b}$. So the intersection problem can be written as a ray-ray intersection as follows:

Find $\alpha \in [0, 1]$ and $t \in [0, \infty)]$ s.t.

$$\mathbf{a} + \alpha(\mathbf{b} - \mathbf{a}) = \mathbf{o} + t\mathbf{d} \tag{2}$$

If such a combination of $\alpha$ and $t$ exists, we have an intersection. As we are in 2D there are two equations for two unknowns and the system always has a solution. The solution can then be checked, s.t. the values are in the right intervals. A small mathematical trick is to define a 2D cross product which is basically just the $z$ component of a 3D cross product if the two input vectors $\mathbf{p}$ and $\mathbf{q}$ were parallel to the $xy$ plane:

$$\mathbf{p} \times \mathbf{q} = p_x \cdot q_y - p_y \cdot q_x \in \mathbb{R} \tag{3}$$

Observe that same as the 3D cross product, the 2D version becomes 0 when you cross a vector with itself. If one now crosses Eq. (2) with $\mathbf{d}$ on both sides the intersection equation becomes:

$$\mathbf{a} \times \mathbf{d} + \alpha(\mathbf{b} - \mathbf{a}) \times \mathbf{d} = \mathbf{o} \times \mathbf{d} \tag{4}$$

So $t$ has been eliminated from the equation and we can solve Eq. (4) for $\alpha$:

$$\alpha = \frac{(\mathbf{a} - \mathbf{o}) \times \mathbf{d}}{\mathbf{d} \times (\mathbf{b} - \mathbf{a})} \tag{5}$$

If $\alpha$ satisfies the condition, we continue analogously for $t$ by crossing Eq. (2) with $\mathbf{b}-\mathbf{a}$. The resulting $t$ is then checked against the condition and a normal at the intersection point is calculated. The intersected rays can be seen in in Figure 1.

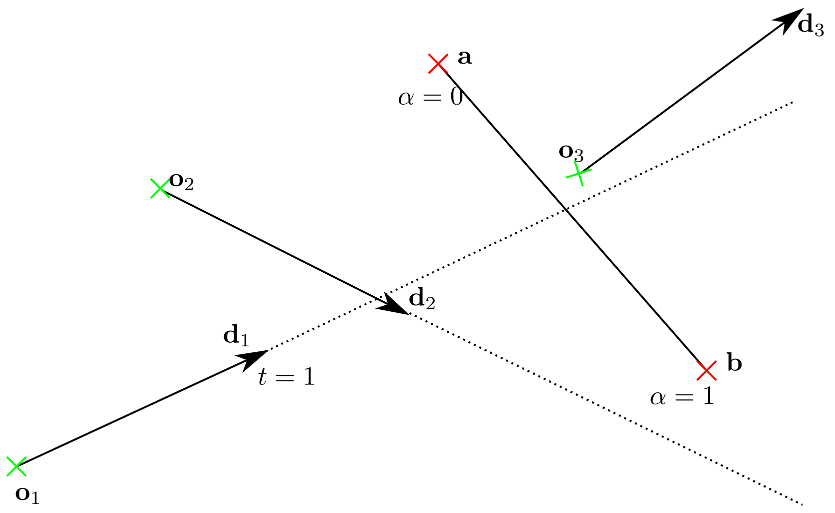

Figure 1: Ray-line intersection of two rays. The line is specified by the points $\mathbf{a}$ and $\mathbf{b}$ and the rays are defined by the origins $\mathbf{o}_i$ and directions $\mathbf{d}_i$. Ray $(\mathbf{o}_1, \mathbf{d}_1)$ satisfies the conditions $t \geq 0$ and $0 \leq \alpha \leq 1$ and therefore causes an intersection, ray $(\mathbf{o}_2, \mathbf{d}_2)$ dissatisfies the $\alpha$ condition and ray $(\mathbf{o}_3, \mathbf{d}_3)$ does not satisfy the $t$ condition.

Another important shape to intersect are AABBs. They are rectangles aligned with the axis of the coordinate system so they require minimal memory space and intersection tests are as simple as possible. They most often used to surround complex objects or parts of it to reduce the amount of intersection tests. First the AABB of the object is tested and only if there is an intersection the actual fundamental shapes inside the AABB are tested. A 2D AABB is defined by two points $\mathbf{b}_{min}$ and $\mathbf{b}_{max}$ which represent the lower left and upper right corner of the rectangle. The intersection test is done by comparing the values of $t$ at each of the axis aligned lines defining the box. The $t$ values for the $x$ axis aligned lines can be calculated as shown in Algorithm 1.

**Algorithm 1:** Intersection test for a AABB $(\mathbf{b}_{min}, \mathbf{b}_{max})$ with ray $(\mathbf{o}, \mathbf{d})$

```
$t_{x1} = \frac{b_{min_x} - o_x}{d_x}$;
$t_{x2} = \frac{b_{max_x} - o_x}{d_x}$;
$t_{min} = \min(t_{x1}, t_{x2})$;
$t_{max} = \max(t_{x1}, t_{x2})$;
$t_{y1} = \frac{b_{min_y} - o_y}{d_y}$;
$t_{y2} = \frac{b_{max_y} - o_y}{d_y}$;
$t_{min} = \max(t_{min}, \min(t_{y1}, t_{y2}))$;
$t_{max} = \min(t_{max}, \max(t_{x1}, t_{x2}))$;
if $t_{min} \geq 0$ and $t_{min} \leq t_{max}$ then
    AABB was hit!
end
```

If the conditions $t_{min} \leq t_{max}$ and $t_{min} \geq 0$ hold there is an intersection. This process is better understood visually and is illustrated in Figure 2. If normals are needed they can be easily calculated since there are only four possibilities depending on which side of the box is intersected first.

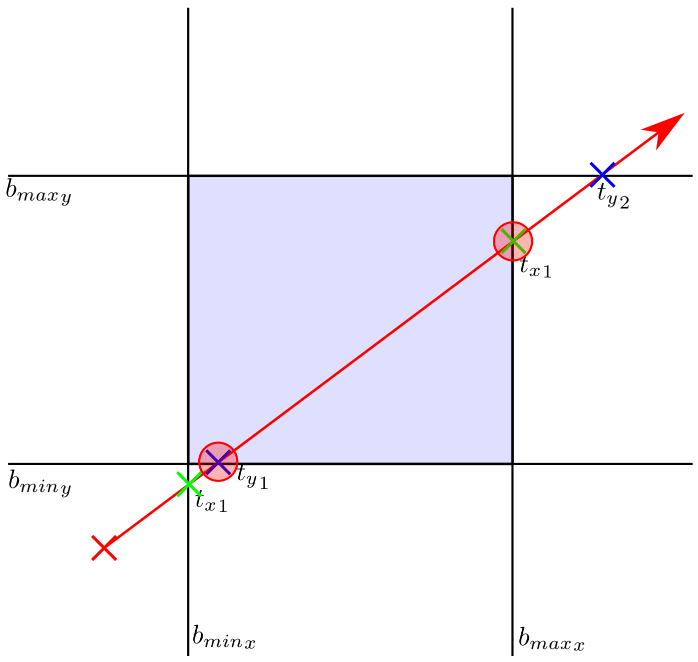

Figure 2: Ray-AABB intersection, where first the intersection points with the $x$ axis (marked in green) and $y$ axis (marked in blue) are calculated. Each of the values $t_{i1}, t_{i2}$ are then split into the minimum and the maximum of the two. Both maximums are then compared and the minimum is chosen as the final $t_{max}$. Analogously the minimums are compared and the maximum is chosen as $t_{min}$ (marked with red circles). Thus there is an intersection with an entry point $\mathbf{o} + t_{min}\mathbf{d}$ and an exit point $\mathbf{o} + t_{max}\mathbf{d}$ and the normals can be calculated depending on which sides the points reside.

The AABB intersection becomes really handy once one wants to use them for ray tracing acceleration techniques, as they can easily be constructed to surround a cloud of points and then be used as a spacial subdivider in a tree structure. This is described in detail in the next chapter. Other shapes like circles or ellipses are intersected in a similar way but as they are not used in the example below the intersection process is not explained here.

Explain reflection

Explain refraction with dispersion effects

## 4.2 Raytracing Acceleration

As the raytracer is later intended to be used in an iterative Optimization algorithm, it is of vital importance that unnecessary computational cost is avoided. For a raytracer this can be achieved in a number of ways. The first and simplest way is to simply reorder the objects in a scene by a heuristic that describes the likelihood of on object to be the first object hit by the majority of the rays. Of course, this only works well if rays are shot into a scene from a dominant direction. Another way would be to subdivide the entire 2D scene with quadtrees and try to fill each branch of the tree with an equal amount of objects or shapes.

Similarly one can also subdivide an object itself and sort the fundamental shapes comprising that object into a quadtree. This method was chosen in this work as there are a limited amount of objects in the scene with the objects possibly being quite complex. Once the fundamental shapes of an object are known, they can be sorted into a quadtree of AABBs of a chosen depth. The outermost AABB is the root of the tree with four children, each encompassing the shapes inside their quarter of space as tightly as possible. This is recursively done until the desired depth is hit. The intersection test of an object then can be done by hitting the root AABB of the tree and then stepping through its children via breadth first search. Each AABB child the ray hits, is persued further and the ones the ray doesn't intersect are ignored. If a leaf has been hit all the shapes inside are then tested for intersection. Finally the $t$ values of all the intersections are compared and the minimum and the maximum chosen as entry and exit points. The advantage of this is that

AABB intersection tests are done really fast and a large number of fundamental shape intersection tests are avoided. An illustration of subdivision of a mirror comprised of line segments is given in Figure 3 and a the intersecting algorithm for a single object is givin in Algorithm 2.

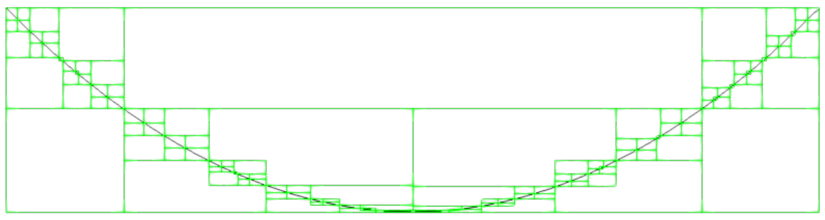

Figure 3: A parabolic mirror comprised of line segments subdivided by a quadtree of AABBs with depth 4. Note that the AABBs are encompassing their contained line segments as tightly as possible.

**Algorithm 2:** Intersection test for a single object subdivided by a quadtree

```
IntersectionResult objectResult;
objectResult.tEnter = MaxFloatingPoint;
objectResult.tLeave = MinFloatingPoint;
Queue treeQueue;
treeQueue.push(object.root);
while !treeQueue.empty() do
    tree = treeQueue.front();
    IntersectionResult aabbResult = tree.aabb.intersect(ray);
    if aabbResult.hit then
        for shapes in tree.shapes do
            IntersectionResult shapeResult = shape.intersect(ray);
            if shapeResult.hit then
                set objectResult appropiately;
            end
        end
        treeQueue.push(tree.children);
    end
    treeQueue.pop();
end
```

## 4.3 Sampling Techniques

Uniform Sampling

Stratified Sampling

Importance Sampling

## 4.4 Framework Structure

### 4.4.1 Rays

### 4.4.2 Shapes

### 4.4.3 Objects

### 4.4.4 Scene

### 4.4.5 Tracing Algorithm

### 4.4.6 Sampler

### 4.4.7 IO Utilities

## 4.5 Setup Specific Objects

### 4.5.1 Lens

### 4.5.2 Mirror

### 4.5.3 Crystal

# 5 Optimization

## 5.1 Functional Analysis

## 5.2 Mesh Adaptive Direct Search (MADS)

## 5.3 Biobjective MADS

## 5.4 Nomad Library

## 5.5 Integration into Framework

# 6 Examplatory Setup

## 6.1 Setup

## 6.2 ASLD Software

## 6.3 Beam Analysis

## References

[1] Opengl mathematics (glm). `https://glm.g-truc.net/0.9.4/api/index.html`. [Online; accessed 28-March-2022].

[2] Will Schroeder, Ken Martin, and Bill Lorensen. *The Visualization Toolkit (4th ed.)*. Kitware, 2006.