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:	
Bearbeitungszeitraum:	1.11.2021 – 2.5.2022

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 11. April 2022 .....

Contents

1 Introduction

In the light of the recent developements in global energy policy, renewable energy has become one of the most important problems humanity has to solve. Ever new ways of exploiting the sun's vast amount of energy are becoming relevant if nations across the world want to achieve net zero carbon emissions. One of those novel methods is the generation of hydrogen from water using solar energy. It can be achieved by common electrolysis or by reacting alkali metals with water. Researchers have been specifically looking at reacting magnesium (Mg) with water (H_2O) to produce hydrogen (H_2) and magnesium oxide (MgO) [7]. This reaction is exotherm and therefore causes a large amount of heat and produces hydrogen gas which could be stored in hydrogen fuel cells. Now if one can reduce the magnesium oxide to pure magnesium using the suns energy one would have a solution to store solar energy using hydrogen. To drive the reduction of magneisum oxide a laser can be used but a considerable amount if energy is needed. In order to reduce losses that are induced by using a conventional solar panel that drives a diode to pump the laser, it could be benefitial to pump the laser crystal directly using sunlight. This is exactly what Shigeaki Uchida and his team in Japan have been researching [13] [14].

Another area of application that is becoming increasingly relevant is the usage of solar lasers in space exploration. Since there is no access to grid power in space and nuclear power is coupled with significant costs solar power is the most used source of power in space. The low efficiency of solar panels and the reduced number of parts of solar lasers make them an interesting prospect for usage in space. As mass is a high cost factor in space exploration the reduced weight and lower number of potential points of failure solar lasers could become a more relevant option in the future. The tasks of a solar laser in space could range from deep space communication, remote power transmission or tracking of objects.

Solar lasers require the collection of sunlight and focusing onto a gain medium to surpass the lasing threshold thereof. As it is the most simple and cost effective method, usually a primary collector is used together with a secondary mirror in a two stage collector to focus the sunlight onto the gain medium [14]. The primary collector can be another mirror or a fresnel lens as a cheaper alternative. The beam power and quality is significantly impacted by the amount of power absorbed by the gain medium and the uniformity of the absorption profile. The natural divergence of sunlight and dispersion effects in the fresnel lens make it important that the secondary mirror is shaped in an optimal way. Both absorbed power and the uniformity of absorption need to be optimized. For the optical design of the collection system it is benefitial that the system is simulated accurately beforehand. For both the simulation and optimization part of the design process a free and open source framework was developed in this work in the hope that parts or the entirety of code may prove useful to engineers designing solar lasers. The goals of the framework are to offer a simple yet powerful interface for C++ applications. It provides a fast 2D raytracer for the calculation of the absorption profile in the gain medium which is then used by a mesh adaptive direct search algorithm in a biobjective manner (BIMADS) to increase both absorbed power and uniformity of the absorption profile. This is done via the open source library NOMAD version 3 [6], which implements the MADS [3] algorithm and a biobjective variant of it. It offers the efficient derivative free optimization of a black-box function with constraints.

The application of the framework is then demonstrated via an examplatory setup of a Nd:YAG solar laser using a two stage concentrator consisting of a fresnel lense and a secondary mirror. In principle, any parameter of the setup can be optimized but for this example in particular the mirrorshape is the interesting property. The result of the BIMADS algorithm is a pareto front of optimal points determined by the algortihm. Depending on whether a more even distribution of power is desired or the total amount of power absorbed is relevant to the application, some points of the pareto front are then chosen and simulated with the software ASLD [12] to evaluate the resulting beam properties.

2 Lasers

Laser is an acronym which stands for Light Amplification by Stimulated Emission of Radiation. Lasers amplify coherent radiation at the infrared, visible, or ultraviolet part of the electromagnetic spectrum [11]. The principle was originally used in so called masers the amplification of Microwave radiation or even radio frequencies. The advantages of using a laser system as a lightsource are that they produce a directional beam of coherent light which can be focused to a narrow spot [5]. Additionally the emitted light is usually of a very narrow spectrum and thus reduces dispersion effects when shooting the beam through different media. Lasers are therefore used in a wide variety of applications which range from manufacturing processes to measuring systems to optical communication.

A laser usually consists of a gain medium that is capable of amplifying light that passes through by stimulated emission, a pumplight to excite the atoms of the gain medium to higher quantum states and an optical feedback mechanism - often called cavity or oscillator - which usually consists of two mirrors that bounce the light back and forth through the gain medium [11]. One of those mirrors is only partially reflective and is transparent so that a portion of the amplified light can escape. Usually in more complex setups cooling is applied to the gain medium and some other optical elements like lenses or polarization filters may be present to ensure a better output beam quality. An ideal output beam is both temporally and spatially coherent, meaning that the emitted light is a perfect sine wave with constant amplitude und frequency and has a definite amplitude and phase pattern across any transverse plane inside the laser [11].

2.1 Stimulated Emission

There are three ways in which atoms exchange energy with a radiation field [5] identified by Albert Einstein. There is absorption where an electron is excited by a photon to a higher quantum energy state. Hereby the photon must have the exact amount of energy (wavelength) that the difference between energy state is. Then there is spontaneous emission where the excited electron jumps back to a lower energy state, emitting a photon in a random direction with a random phase shift but again with the same amount of energy as the difference between states of the electron. This can occur spontaneously at any time as the name suggests. The main priciple why the gain medium amplifies light is the priciple of stimulated emission. Electrons in the gain medium are stimlated by photons to a higher energy state. If now a again photon with the same amount of energy and a certain direction hits the atom the electron jumps back to the lower state again emitting a photon of the same energy but crucially and contrary to spontaneous emission in the exact same direction and the exact amount of phase shift as the incoming photon. Therefore amplifying the light by essentially "duplicating" the incoming photon.

Now if one wants a coherent output beam one needs to make sure that the photons are travelling only in one direction and with constant phase. This is the job of the resonator. It uses the photons from spontaneous emission which at some point will have the correct direction and bounce them between the mirrors. Photons with other directions will be lost from the sides entirely or will get absorbed again by the medium. Due to this process a majority of photons will be travelling in the desired direction after some time.

2.2 Gain Medium and Population Inversion

In order to be able to amplify the emitted light, more atoms in the gain medium have to be in an already excited higher state than in the lower state. Otherwise "duplicated" photons will be reabsorbed by atoms in lower state and will not be able to stimulate another emmission or make it out of the laser cavity. Hence the population of atoms needs to be inverted [5]. For this to happen an external source of energy needs to be supplied. This is called pumping and is usually achieved by a pump flash light or another laser. As it is equally likely that a photon causes stimulated emission or absorption there can not be only two states but at least three states are needed in optically pumped lasers [9]. The electrons are then excited into the highest state by the pump light from which they can decay into the middle state ready for stimulated emission. It is crucial for level three lasers that the pump light cannot push the electrons in the middle state back to ground

state. This way it is possible to have more atoms in the middle state than in ground state and therefore population inversion is achieved. For this to happen the pump light intensity in level three lasers needs to be sufficiently high enough for the photons to be "ignored" by the electrons in the second state. The threshold for the pump power to achieve population inversion is called the *lasing threshold* and due to the energy levels in the atoms the choice of gain medium usually implies the choice of pump light or vice versa.

Materials that offer this property can be in gas, liquid or solid form. Solid form lasers are usually some sort of ion doped crystals or semiconductor diodes [5]. As an example for a three level gain medium ruby ($Cr^{3+} : Al_2O_3$) can be used. In practice mostly four level gain media are used as they offer a far lower lasing threshold for the pump power [9]. These are usually neodymium doped media like the most popular choice neodymium doped yttrium aluminum garnets (Nd:YAG).

3 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) [10].

In the following chapters the applied ray tracing techniques explained in detail. Firstly the basics of raytracing, i.e. intersection tests of fundamental shapes and objects and reflection and refraction effects are shown. Then an applied method of subdivision for the performance optimization of the

raytracer is explained. Lastly some methods of sampling are shown before the structure of the developed framework is presented with code samples. Here the usage of classes is demonstrated and it is shown how specific objects are defined. In particular the objects which are relevant for the simulation of laser cavities and which are used in the example setup are shown.

3.1 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 α 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 α:

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

If α 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.

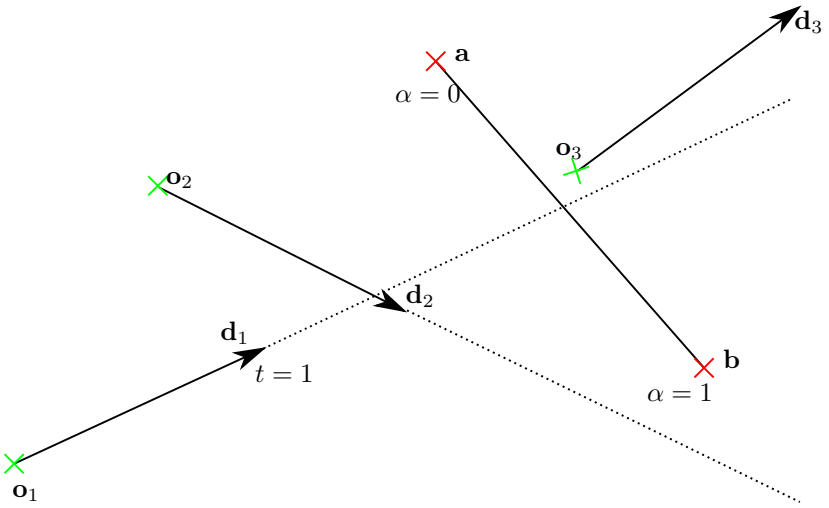

Figure 1: Ray-line intersection of two rays. The line is specified by the points $\mathbf{a}$ and $\mathbf{b}$ and the rays are defined by the origins $\mathbf{o}_i$ and directions $\mathbf{d}_i$. Ray $(\mathbf{o}_1, \mathbf{d}_1)$ satisfies the conditions $t \geq 0$ and $0 \leq \alpha \leq 1$ and therefore causes an intersection, ray $(\mathbf{o}_2, \mathbf{d}_2)$ dissatisfies the α 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.

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.

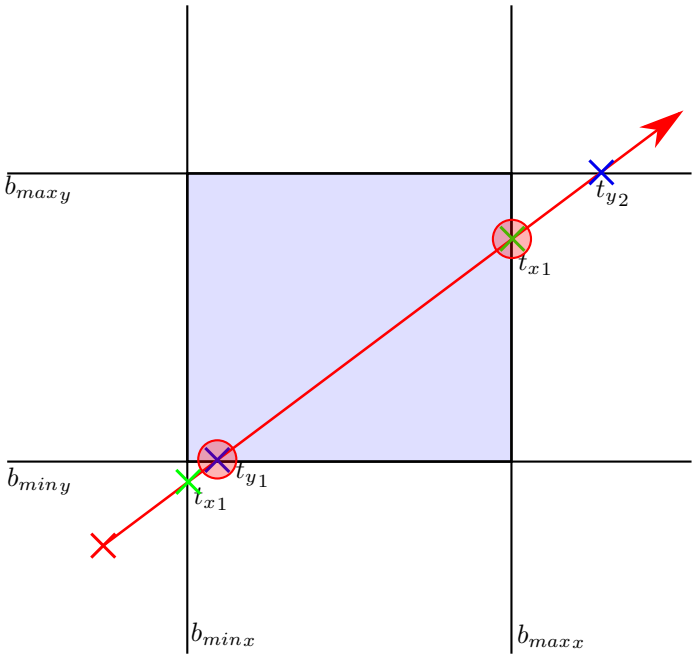

Figure 2: Ray-AABB intersection, where first the intersection points with the x axis (marked in green) and y axis (marked in blue) are calculated. Each of the values t_{i1}, t_{i2} are then split into the minimum and the maximum of the two. Both maximums are then compared and the minimum is chosen as 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.

Once an intersection takes place, the ray will be either reflected, terminated or refraction occurs depending on the desired material of the object. Total reflection is only dependent on the incident angle θ_i to the normal of the surface at the hitpoint. Then the reflection angle θ_r is given by Eq. (6).

$$\theta_r = -\theta_i \tag{6}$$

A new ray is then generated at the hitpoint pointing in the direction given by θ_r. Due to limited floating point precision, it is required that the origin of the new ray is shifted by a small ϵ towards the reflection direction in order to make sure the ray is originated at the correct side of the material. Since the reflection is total the entire amount of power of the incident ray is transferred to the reflected ray.

When hitting a material that is transmissible for light the ray will be refracted at the boundary between the two media. The effects of matter on a light beam are described by Snell's law and the Fresnel equations. The ray is split into a reflected and a transmitted ray. The direction of the transmitted ray is governed by Snell's law in Eq. (7) which depends on the indices of refraction of the two media n_i and n_t.

$$n_i \sin(\theta_i) = n_t \sin(\theta_t) \tag{7}$$

Naturally, the reflected ray is still reflected as given in Eq. (6). The transmitted and reflected power can be calculated with the transmission- and reflection rates given by Fresnel's equations. These are dependent on the orientation of the polarization of the incident ray (perpendicular or parallel) to the surface.

$$R_\perp = \frac{\sin^2(\theta_1 - \theta_2)}{\sin^2(\theta_1 + \theta_2)} \quad R_\parallel = \frac{\tan^2(\theta_1 - \theta_2)}{\tan^2(\theta_1 + \theta_2)} \quad T_\perp = 1 - R_\perp \quad T_\parallel = 1 - R_\parallel \tag{8}$$

For unpolarized light the total rates are just given by the average.

$$R_{total} = \frac{R_\perp + R_\parallel}{2} \quad T_{total} = \frac{T_\perp + T_\parallel}{2} \tag{9}$$

An additional effect that can be significant espacially for broadband applications is the dependency of the index of refraction of a medium on the wavelength of the light passing through and the dispersion of light resulting from this. This is modelled by the Sellmeier equation, which is based on empirical measurements. The relationship between the refractive index n and the wavelength λ in micrometers is described by a series of Sellmeier coefficients B_i and C_i that have been determined by experiment. The Sellmeier equation is thus given by Eq. (10).

$$n^2(\lambda) = 1 + \sum_i \frac{B_i \lambda^2}{\lambda^2 - C_i} \tag{10}$$

3.2 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.

3.3 Sampling Techniques

The accuracy and performance of ray tracing simulations are heavily dependent upon using the correct sampling techniques. One could sample values on a uniform grid or equally spaced intervals. The problem with this is that there is no randomness or irregularity causing structured aliasing errors in most applications. Random sampling however always relies on some sort of random number generation. These are usually pseudo random numbers generated according to some distribution with the generation engine initialized with a seed. Usually when one samples according to some scheme only values in $[0, 1]$ are allowed. The returned sample is then later scaled to the desired range depending on the usecase. It is also to be noted that calls to a sampler must be ensured to be as effiecient as possible as a large number of calls will be made during the simulation.

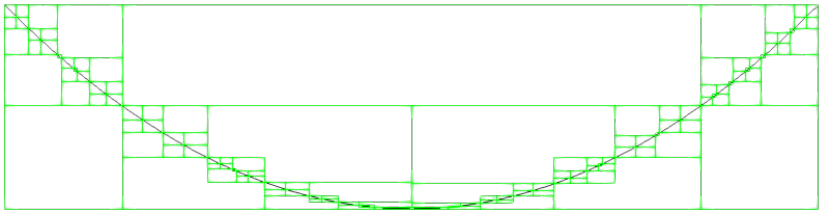

Figure 3: A parabolic mirror comprised of line segments subdivided by a quadtree of AABBs with depth 4. Note that the AABBs are encompassing their contained line segments as tightly as possible.

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
```

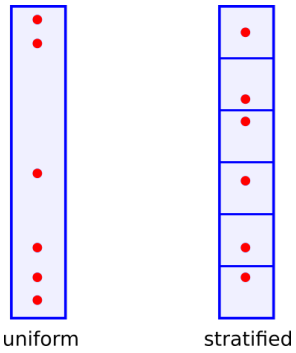

Figure 4: Uniform sampling of a 1D interval (left) vs. stratified sampling (right). Observe the large gap between samples on the left whereas the samples on the right are spaced out more equally.

The simplest sampling scheme is uniform sampling. It returns values uniformly and can be implemented right on top of the random number engine of the used system. The advantage of uniform sampling is that it produces close to random samples without the need for additional logic and therefore performance losses. The disadvantage is the irregular density of samples within the interval. There can be areas with a lot of samples and large gaps between. So in scenarios where there needs to be a more regular distribution of samples uniform sampling is not optimal.

For this reason another sampling technique called stratified uniform sampling exists. Here the domain is split into N equally spaced intervals and the uniform sampling occurs within each interval. This ensures that there is some amount of regularity while still keeping the randomness of uniform sampling. An application of stratified sampling would be the definition of a light source in the simulation. The direction or origin of the rays the light source emitts can be sampled according to stratified sampling to ensure a smooth illumination of the scene. The difference between uniform and stratified uniform sampling can be observed in Figure 4.

A more advanced technique is to sample values where they are contributing the most. Suppose one wants to approximate some function f over a domain $[0, 1]$. The criterium that has to be met in order to calculate some quantity Q is given by any arbitrary integral in the domain. With uniform sampling of $x \in [x_1, x_2]$ by the sequence (x_i) and $i = 1...n$ the integral is approximated by Eq. (11).

$$Q = \int_{x_1}^{x_2} f(x)dx \approx \frac{1}{n} \sum_{i=1...n} f(x_i) \tag{11}$$

Now it is also possible to sample according to some other distribution. One just has to know the probability density function (pdf) p to know how likely it is that a sample x_i is generated. Probability density functions are nonnegative across the domain and their integral over the domain is always 1. This corresponds to the probability that a sampled value is in the interval $[x_1, x_2]$ which is of course the case when all possible values are within that interval. The more accurately the pdf p matches f the more $[x_1, x_2]$ is sampled at the points where the function f has a large contribution to the integral. This can significantly reduce the amount of samples needed to get an accurate approximation for the integral Q. The integral is then approximated by Eq. (12).

$$Q = \int_{x_1}^{x_2} f(x)dx \approx \frac{1}{n} \sum_{i=1...n} \frac{f(x_i)}{p(x_i)} \tag{12}$$

Observe that now the values for f are weighted by the likelihood p. If it is likely that a sample x_i is generated - larger p - then the contribution is worth less and vice versa. Of course this requires some preprocessing in order to generate the distribution P from a pdf p. Usually one wants to

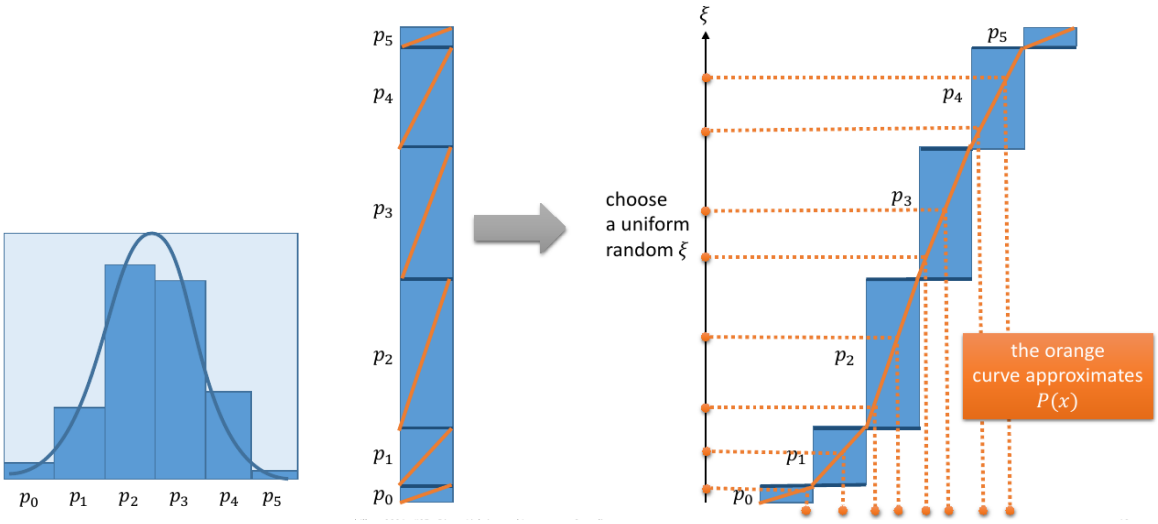

Figure 5: Visualization of inversion method to sample according the pdf on the left. The intevals p_i are summed up and stacked. Then a $\xi \in [0, 1]$ is uniformly sampled and the corresponding interval is searched. The returned value is interpolated linearly according to the value of $/xi$ within the rectangle.

generate p from f but there are also cases where one might choose another function to deduce the pdf and subsequently the distribution.

One method of building such a distribution is to first integrate f in the domain $[x_1, x_2]$ and normalize the function so the integral is guaranteed to be 1. Then to sample according to the resulting pdf one discretizes the pdf to a finite amount of equidistant intervals $p_i = [x_{1i}, x_{2i}]$. For an interval p_j the values $p(x_{2_i})$ for all $i < j$ are then summed up progressively assigned to that interval. These resulting rectangles are then stacked so that each rectangle represents an interval in x and the corresponding values of the sum of the pdf. Then a sample $\xi \in [0, 1]$ is drawn from a uniform sampler and the corresponding interval in which the value ξ lies in is searched in the list of intervals using binary search. Once the interval is found we interpolate linearly between the lower boundary x_{1i} and x_{2i} depending on ξ. Then the resulting $x \in [x_{1i}, x_{2i}]$ is returned along the value of the pdf $p(x)$ as a sample. This process approximates the distribution P and is called the inversion method. It is better understood when visualized as in Figure 5.

3.4 Framework Structure

The goal of the raytracing framework is to offer a free and lightweight alternative to commercial raytracers in order to stay efficient for the later usage in a black-box optimization context. For the optimization, it is crucial that the raytracer can be executed thousands of times even with a higher number of rays without significant overhead. It is to be noted that the raytracer has been specifically designed in order to be used in the optimization of geometrical parameters in rotationally symmetrical systems. Therefore the decision was made to work in a 2D context rather than a 3D one. The additional computational cost of going from a 2D to a 3D context for raytracing is significant and since the framework is to be used as the optimization part of the design process the results can then be taken and further analysed in a 3D raytracer. Suppose a 2D raytracer needs 10000 rays to reduce the noise to a satisfactory level. Then it can be assumed that a 3D raytracer needs about $10000 \cdot 10000$ rays to produce the same result. The only difference is that transversal rays cannot be modelled in 2D but transveral rays which may occur by divergence of sunlight or scattering effects in inpure media are statistically distributed equally across the full rotation of a rotationally symmetrical system and the impact on the result is therefore negligable. Additionally, the described objects in the scene require a significantly higher amount of fundamental shapes, which compounds the performance loss of needing more rays even further. Furthermore the ray

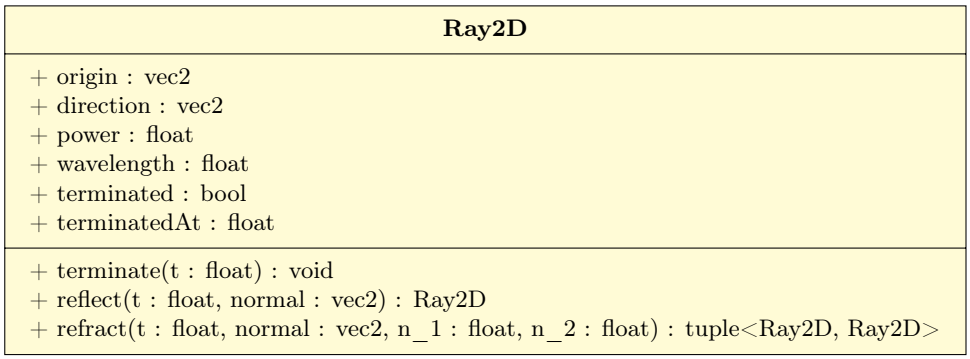

Figure 6: UML class diagram of the Ray2D class

intersect equations shown above would become more complex and would require more arithmetic and cases. So for the relatively small benefit of being able to model transversal rays it is infeasable in a rotationally symmetrical system to simulate in a 3D environment. Of course, if the system is not rotationally symmetrical then there is no other choice than to trace the scene in 3D, but this is not the norm in laser optical systems. In the following chapter the most important parts of the raytracer framework are presented. The datastructures and algortihms are shown in an UML class diagram and an example application of the datastructure or algorithm is given. It is to be noted that not all fields, methods or datatypes are shown in the UML diagrams and the code should therefore be seen as pseudocode instead of specific C++ code. When the classes have already been explained and the fields are not necessary to understand the relation between them they are omitted from the diagram to keep the diagrams more managable. The *float* datatype represents some floating point datatype and not necessarily the C++ datatype. The *vec** vector datatypes are either the GLM [1] datatypes for vectors - if the dimension is 4 or below - or a custom implementation to mimic the vector arithmetic of GLM for higher dimensions. Pointer datatypes in the UML diagrams are entirely reliant on the STL smart pointers in C++. Thus the user usually constructs instances of the objects or shapes via smart pointers which implement reference counting in order to manage heap allocated resources. Therefore there is no need for the user to worry about memory management.

3.4.1 Rays

Rays are the most basic datastructure in the simulation. They are described by an origin, a direction, a field for the amount of power, wavelength and a bool field if a ray has already been terminated. The simplest action is to just terminate the ray at some t in which case it will no longer be considered in the tracing algorithm. Rays can alse be reflected at a point t with a certain normal, where the original ray is terminated and a new ray according to Eq. (6) with a small ϵ shift for te origin in the reflected direction is returned. Another action is to refract the ray between the boundary of two media with refractive indices n_1 and n_2. Here a tuple of new rays is returned and the original ray is terminated. The first element is the reflected ray and the second element is the transmitted ray into the medium. The direction of the transmitted ray is calculated using Eq. (7) and the power of the original ray is split between the reflected and transmitted part according to the Fresnel laws Eq. (8) for unpolarized light Eq. (9). The UML class diagram for the Ray2D class and other relevant classes is shown in Figure 6.

As the user rarely has to interact directly with ray class and the implementation and usage of the methods are straight forward an example is not necessary here.

3.4.2 Shapes

The Shape2D class represents all fundamental shapes in two dimensions It is the baseclass for all specific 2D shapes like lines, AABBs or circles etc. The specific intersection tests with rays are implemented in those derived classes. Each shape additionally has an AABB that encompasses the

represented shape. This AABB is then used to construct the quadtree for the object the shape is part of. When a shape is intersected an interection result is returned, which contains the enter and leave t value of the intersection and a boolean that represents whether an intersection has even occurred. Furthermore the normals at the enter and leave point are contained in the intersection result. Each shape also has a line representation in order to later output the scene in the VTK format for visualization. A UML class diagram of the Shape2D class is shown in Figure 7.

The shapes that are derived from the Shape2D class are the basic unit for the intersection tests used in the tracing algorithm. The implementation of those tests should therefore be computationally efficient and should avoid unnecessary branches. Ideally they should be branchless implementations where only the final decision if an intersection has occurred requires a branch. An example for a branchless intersection test has been shown in Section 3.1 above, specifically the ray-AABB and ray-line intersections. For the ray-AABB intersection two versions of the test should exist. One version that only checks for the intersection and one version, which also calculates the normals. The check-only variant should then be used in Algorithm 2 as no normals are needed to traverse the quadtree and the AABB normal calculation is quite branch heavy. This is what the *intersectCheck* method of the BoundingBox2D class is for. Furthermore the BoundingBox2D class provides multiple ways of constructing the represented AABB. Either directly by giving b_{min} and b_{max} or by providing a point cloud or by merging multiple AABBs together. This is then later used in the construction of the quadtree of the object the shape is in.

3.4.3 Objects

Objects are represented by the Object2D class and are essentially a collection of Shape2D instances. Similarly to the Shape2D class the Object2D is just the base class for the specific objects implemeted by the user or predefined by the framework. To initialize an object a list of shapes has to be provided. The constructor of the Object2D class then builds the quadtree of shapes from that list as described in Section 3.2. The intersection test for the object is then done as shown in Algorithm 2. Furthermore the object base class provides a virtual method for a user defined action to be executed once a ray hits the object. A few presets are already implemented namely pass, absorb and reflect. Pass just ignores the object entirely, absorb terminates the ray at the entry point of the intersection and reflect reflects the ray at the hitpoint. The user defined action provides the capability for the framework to be extensible and customizable. It provides the ray, the intersection result and a reference to a list of newly created rays for the user. For example, the ray could hit the surface and spawn multiple new rays instead of just being reflected modelling some sort of scattering effect. This isn't implemented by the framework out of the box but the user can easliy implement this functionality in a single function. The Object2D class therefore is intended to provide the bare bones functionality for efficient tracing of an object, but the actual physical modelling of the interaction of a light ray with the object can be entirely customized by the user.

Objects that are arbitrarily complex can thus be efficiently implemented with minimal programming. Examples of how the usage of the Object2D class is intended can be seen in the paragraphs below. Here a thin lense approximation, a reflective mirror, and a grid representing a laser crystal are implemented and can be used by the user. A UML class diagram of the Object2D class and associated classes is given in Figure 8.

Medium2D All object that consist of some material that is transmissible by light and have a non negligable thickness should implement the Medium2D interface. It holds information about the material via the materials Sellmeier coefficients and implements the all the phenomena relevant to the correct tracing of light on the border between two materials of different refractive index. These are Snell's law in Eq. (7) for the bending of the ray on material boundaries, the Fresnel laws in Eq. (8) for the correct amount of intensity to be transmitted or reflected and the Sellmeier equation of the lens material in Eq. (10) to accurately describe dispersion effects of the material.

Like all objects in the simulation it inherits from the Object2D class and implements the *action* method in to include the above described effects. Furthermore the Medium2D class provides the abstract *actionEnter* and *actionTransmit* methods which are executed once the ray first hits the

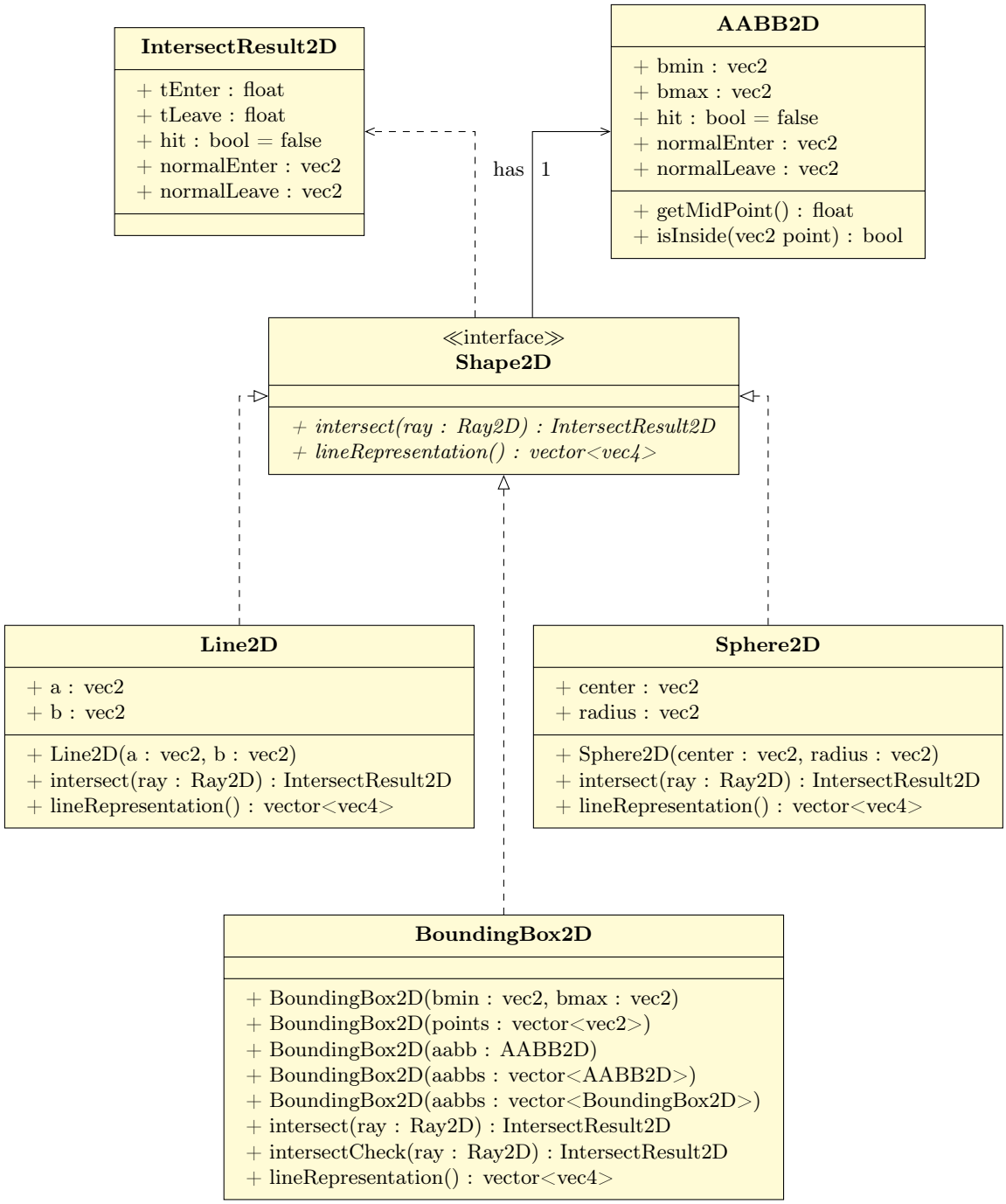

Figure 7: UML class diagram of the Shape2D and associated classes

medium and once the transmitted part of the ray was traced within the medium respectively. They can then be implemented by the user in a user defined object implementing the Medium2D interface. These actions can even modify the affected light ray in order to model absorption or thermal lensing effects within the medium. If the ray is terminated by the user in the *actionTransmit* function, no further tracing is done and no additional rays are created. An example of this modification to the ray itself by the user is the Grid2D class. Here the power of the ray is modified by each cell the ray intersects in order to model the absorption of light within a medium.

The Medium2D inteface should serve as the base class for all user defined objects where the tracing of light needs to be physically accurate. This means that no approximations are made and the required computation per ray transmitting through these objects is quite high. Additionally, one ray that hits the object then in turn creates three rays only one of which - the ray within the medium - is always guaranteed to be terminated. The other two generated rays, namely the reflected part and the ray exiting the medium on the other side are two completely new rays entering the scene. These rays then need to be traced, increasing the overhead of tracing a medium accurately even more. Therefore it is advised to use the Medium2D interface with great care and sparingly in order to keep the computational cost to a minimum. Only the most relevant objects to the simulation should be traced with full physical accuracy. Suppose one wants to model a mirror using the Medium2D class. Technically this is possible, but practically it is infeasable, since the only thing the user is most likely interested in is the reflected part of the incoming ray. The same is true for lenses or optical elements where either the dispersion effects are negligable or the angle of incidence is such that almost all light is transmitted anyway. In this case the user is most likely only interested in the transmitted part of the ray and not the reflected or internal part. To prevent this unnecessary computational cost, one usually wants to approximate or neglect certain parts of the tracing in media. This is what the ThinLens2D and Mirror2D classes shown below demonstrate.

ThinLens2D The accurate physical simulation of a lens follows the same laws as the ray tracing in all media. This is modelled by the Medium2D interface, which implement all relevant effects in a transmissible material. Although it is phyysically accurate to trace the light rays with the Medium2D class, significant amounts of computation is needed to calculate and describe the light paths. Furthermore, the geometry of the lens needs to be modelled accurately. This is simple for conventional glass lenses where the geometry can be described only by the radii of the surfaces and the thickness of the lens. But for fresnel lenses this process can be quite complex and time consuming, both in the modelling phase and the computational cost of tracing many fundamental shapes. Thus it is rarely feasable to trace lenses without any amount of simplification. The question of choosing the appropiate amount of approximation depends on the goal of the simulation and the use case of the optical system and cannot be generally answered. It can be required to trace a specific lens with all the effects described above if the goal of the simulation is to design that specific lens for example. For imaging systems like camera objectives, the lenses are usually aligned in their optical axis, rays are predominantly entering the system at shallow angles and dispersion effects can be largely neglected. Other tasks my require rays to hit the lens at odd angles so the correct geometry of refraction is important, but dispersion effects and transmitted power are not that important or can be approximated by another way. Therefore the user ultimately needs to decide which amount of approximation is acceptable for the task at hand.

Suppose the user simulates a camera objective. Like in most optical systems a larger number of lenses are aligned in their optical axis and rays are only hitting the lens with a small angle with respect to those axis. Here the only thing that matters is the correct refraction of rays according to the rules of geometric ray tracing in a thin lens. Parallel rays always intersect the focal point and rays that travel through the middle of the lens will not be refracted at all. Those lenses are therefore often modelled by the ABCD ray transfer matrix [8]. The lens is seen as a sort of blackbox with a certain thickness and the transfer matrix describes how a ray is transmitted through the lens. The only information needed is the thickness of the lens and the focal point of the lens. If multiple lenses are in a row and described by ABCD transfer matrices, the transfer matrix of the whole system can then be calculated from the individual ones. The ABCD transfer matrix only provides correct results once the angles of the ray with respect to the optical axis of the lens are shallow. Then the $\sin(\theta)$ in Snell's law Eq. (7) is just approximated by θ for small angles (paraxial approximation).

Since lenses in the raytracer in this work can be rotated in any way, a more general approach is to solve a trigonometrical formula for each ray hitting the lens. In this case also rays with steep angles are refracted correctly. For the purposes of simplification the lens is usually made infinitely thin. This way of tracing a thin lens is described in [4]. It provides a good compromise between an approximation and using fully accurate refraction on a exactly modelled piece of glass, where normals and angles need to be calculated and the laws of Snell, Fresnel and Sellmeier need to be applied. Naturally, dispersion effects can not be modelled by this approach, but for non-imaging systems it is often sufficient to trace lenses without dispersion effects, especially because glass has a very low dependency of the refractive index on wavelength according to its Sellmeier coefficients. If one still wants to model dispersion effects with this approach, one method could be to provide an artificial thickness function or just upper bound depending on the applications requirement for physical accuracy. Then the Sellmeier coefficients for the desired material of the lens in combination with the sampled thickness at the hitpoint can be used to disperse the ray correctly. An example for this would be a fresnel lens made from PMMA. The sawtooth pattern of the ridges could then be given as a function to be sampled at the hitpoint.

The ThinLens2D class represents the approximation of a thin lens according to [4]. It takes the radius and desired focal length and constructs an object whose only fundamental shape is a Line2D shape representing the thin lens. Once a ray hits the line the ray is terminated and a new ray pointing in the direction of refraction is generated at the hitpoint.

OPTIONAL: implement abcd lenses and maybe fresnel lens, dispersion!!!

Mirror2D Mirrors like lenses are commonly used in optical systems. Most of the time one wants optimize the shape of a mirror system in order to fullfill some optical property. As all objects in the framework, mirrors are also considered to be rotationally symmetrical. The mirrors implemented in the Mirror2D class consist of line segments of the class Line2D, the amount of which can be given as an argument to the constructor. Additionally the shape of the mirror can be given as a parametrized 2D curve. The function should take a single parameter $t \in [0, 1]$ and return 2D points along the curve. The parameters of the shape function can then be changed by the simulation in an optimization algorithm and the mirror reconstructed with the updated values using the *rebuild* method. Naturally the action of the Mirror2D object is to reflect the ray according to Eq. (6).

Grid2D The Grid2D class offers to simulate quantities distributed within a medium. Rays are traced according to the laws in Eq. (7), Eq. (8) and Eq. (10). When the ray first hits the medium a hit action can be given if the hitpoint contains relevant information to the user. Then the distance within each cell the ray is travelling through is eveluated. It is to be noted that it is assumed the medium is homogenous and has constant refractive index. Thus the rays travel in a straight line within the medium. Additionally to the hit action a cell action can be given as a parameter to the constructor. The cell action function is executed once the ray hits a cell. The distance travelled through the cell is evaluated and given to the cell action function, which then in turn can access the cell values in order to change them. The tracing within the grid is done using a fast 2D voxel algorithm shown in [2]. Here the cells are accessed in order the ray traverses them, which is important should the cell action change the ray parameters in any way.

Absorbing media are modelled using the Grid2D class and supplying a cell action that evaluates the lost power via Lambert's law of absorption. It describes the exponential decay of the remaining power of a ray of light in an absorbing medium depending on the coefficient of absorption of the medium and the distance travelled. The remaining power of a ray of light is then given by Eq. (13).

$$P_{rem} = P_{ray} \cdot e^{-\alpha(\lambda)d} \tag{13}$$

Here the absorption coefficient is α and the distance travelled by the ray is d. In general α depends on the vacuum wavelength of the ray λ. For this reason an absorption spectrum of the medium can be supplied and evaluated at each cell. Since the ray's wavelength stays constant this evaluation can be optimized by evaluating $\alpha(\lambda)$ only once the ray hits the medium and keeping track of this value for successor cells. The absorbed power is then added to the cells value. Thus after a full simulation of the scene the absorbption profile is known. Additionally the total irradiation

is evaluated when the ray hits the medium, if the user requires this information. Supplying the apsorption spectrum can be done by using the various utilities the framework provides. These are described in Section 3.4.6.

optimize α evaluation

split crystal class to medium, grid and crystal class

3.4.4 Scene

The Scene2D class represents an entire simulation setup with a collection of initial rays and objects. Objects can be added to the scene via the *add* method. It also provides functionality for generating the initial rays via different light source setups and implements the tracing algorithm. The tracing algorithm simulates rays until a certain depth is reached and then outputs all rays that have been created during the simulation. The depth is defined by the number of executed actions by objects along the path of the ray. So suppose two opposing mirrors $mirror_1$ and $mirror_2$ bounce a ray between them. A depth of two is reached once the ray hits the $mirror_1$ object and gets reflected by the objects defined action. The second action is executed by $mirror_2$ when the ray is reflected again on its surface. Because a depth of two has been reached the simulation stops and outputs the created rays for each depth: At depth zero the initially created ray, at depth one the reflected ray which bounced from $mirror_1$ to $mirror_2$ and at depth two the ray which originates at $mirror_2$'s surface and goes to infinity. This way the user can decide which depth is sufficient to simulate the desired phenomena. If one wants to observe the absorption of energy in a thin piece of ionized glass with a weak absorption coefficient it might be required to choose a higher depth in order to bounce the ray through the glass multiple times until the absorbed power becomes insignificant and a steady state is reached. Theoretically the user can implement a preprocessing step where the simulation is executed multiple times with increasing depth until a steady state is reached or no more rays are created and choose that depth parameter for the optimization. There are valid reasons though to choose a smaller value than is actually required to reach steady state. If the scene is quite complex and a significant number of rays are needed, it can be necessary to limit the depth in order to reach a result in an acceptable amount of time. The same is true if the absorption for the example above is dominated by the absorption from the first pass of the ray, i.e. the medium has a strong absorption coefficient.

The tracing algorithm starts with the initial rays at depth zero. The rays for the next higher depth are generated by tracing the rays of the next lower depth against the scene. Here rays that have been terminated in the previous depth are ignored and no longer persued. If a ray has not been terminated it will be tested against all objects in the scene. The intersection results are then stored in an ordered datastructure, sorted in an ascending order by the t value of the object intersection result. The actions defined by the intersected objects are then executed in that order. If preceeding object terminates the ray or all actions have been executed the iteration is stopped and the created rays are stored for the current depth. Created rays can also be immediatly terminated by the action of the creating object. An example for this would be the refraction of the ray where the transmitted ray is terminated at the boundary of the object and the exiting ray is still traced by the next higher depth. The tracing algorithm of the Scene2D class can be seen in Algorithm 3.

3.4.5 Sampler

The sampling classes implement the sampling techniques discussed in Section 3.3. The abstract interface for sampling is the Sampler class, which is can sample any given datatype. It describes how all other sampling classes are to be used. After construction the sampler needs to be initialized to have an idea about how many samples the user wants in total. Then the user can retrieve samples via the *next* method. The Sampler class also provides an index fields that keeps track of the current index of the sample, should a derived class need that information. One layer below the Sampler interface are interfaces which describe the type of distribution the implemented sampler relies upon. Here the random engines of the STL are usually constructed and initialized with a unique seed. Most specialized sampling techniques for specific datatypes then inherit from those interfaces or use some sort of combination of different random engines. A UML class diagram of the sampler environment is given in Figure 9. Example outputs of predefined sampling techniques are shown in Figure 10.

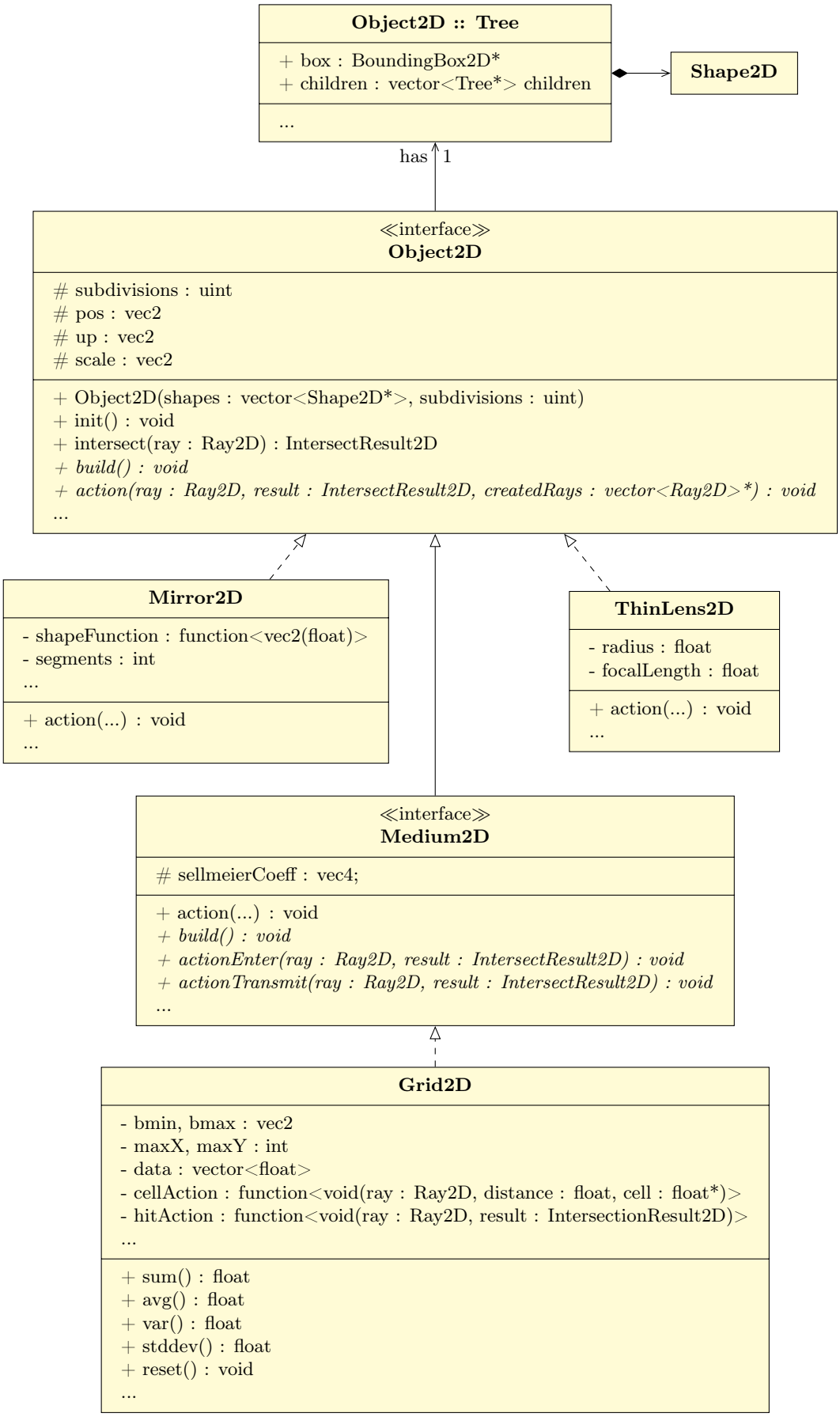

Figure 8: UML class diagram of the Object2D and associated classes

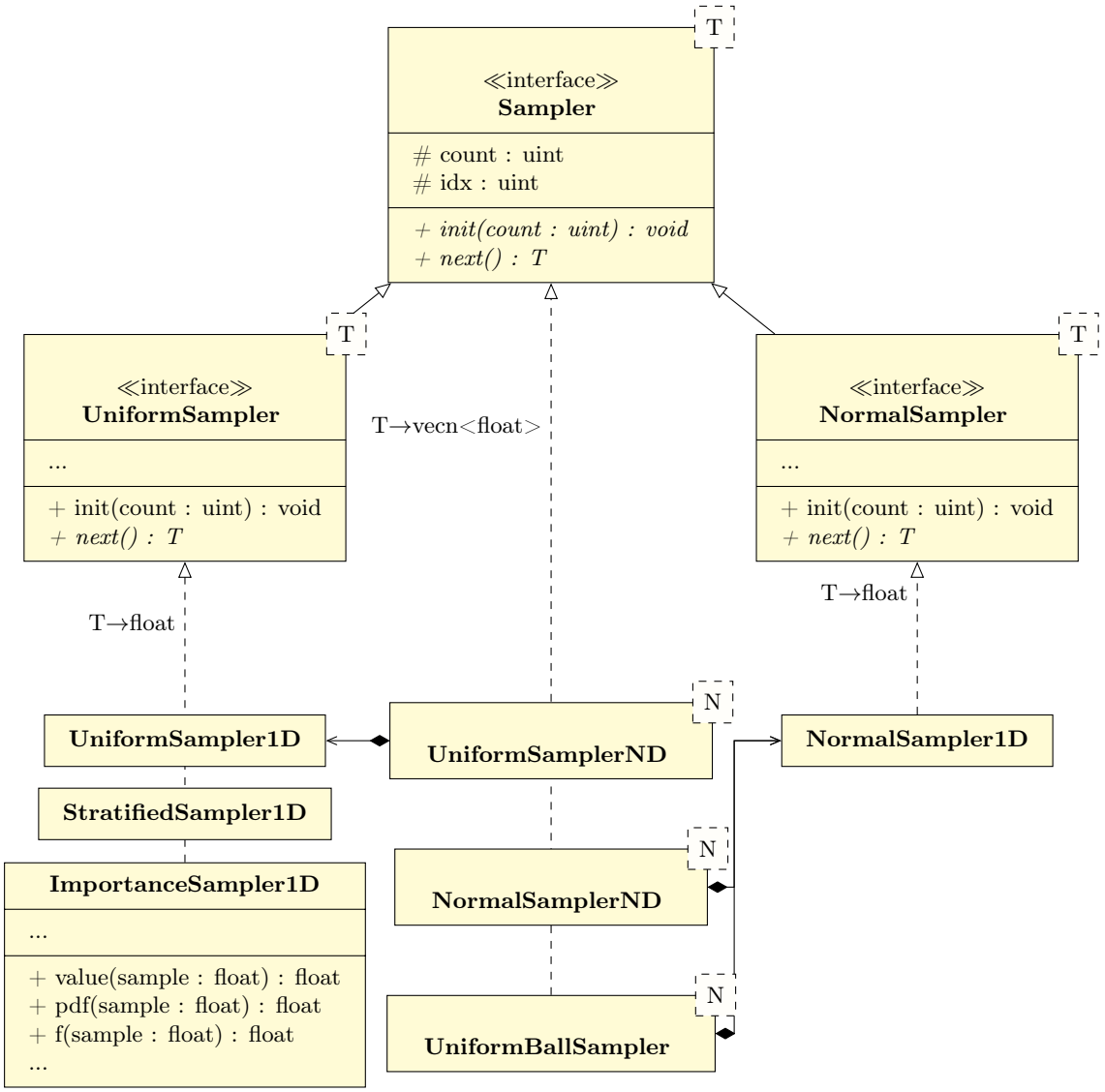

Figure 9: UML class diagram of the Sampler and associated classes

Algorithm 3: Tracing algorithm of the Scene2D class

```
vector<vector«Ray2D» rays(depth);
rays[0] = startrays;
for d in 1 ... depth do
    vector<Ray2D> createdRays;
    for ray in rays[d-1] do
        if ray.terminated then
            continue;
        end
        orderedMap<float, tuple<Object2D*, IntersectResult2D» intersections;
        for object in objects do
            IntersectResult2D result = object->intersect(ray);
            if result.hit then
                intersections.insert(result.tEnter, object, result);
            end
        end
        for (t,item) in intersections do
            item.object->action(ray, item.result, createdRays);
            if ray.terminated then
                break;
            end
        end
    end
    rays[d] = createdRays;
end
```

3.4.6 Utilities

4 Optimization

4.1 Functional Analysis

4.2 Mesh Adaptive Direct Search (MADS)

4.3 Biobjective MADS

4.4 Nomad Library

4.5 Integration into Framework

5 Examplatory Setup

5.1 Setup

5.2 ASLD Software

5.3 Beam Analysis

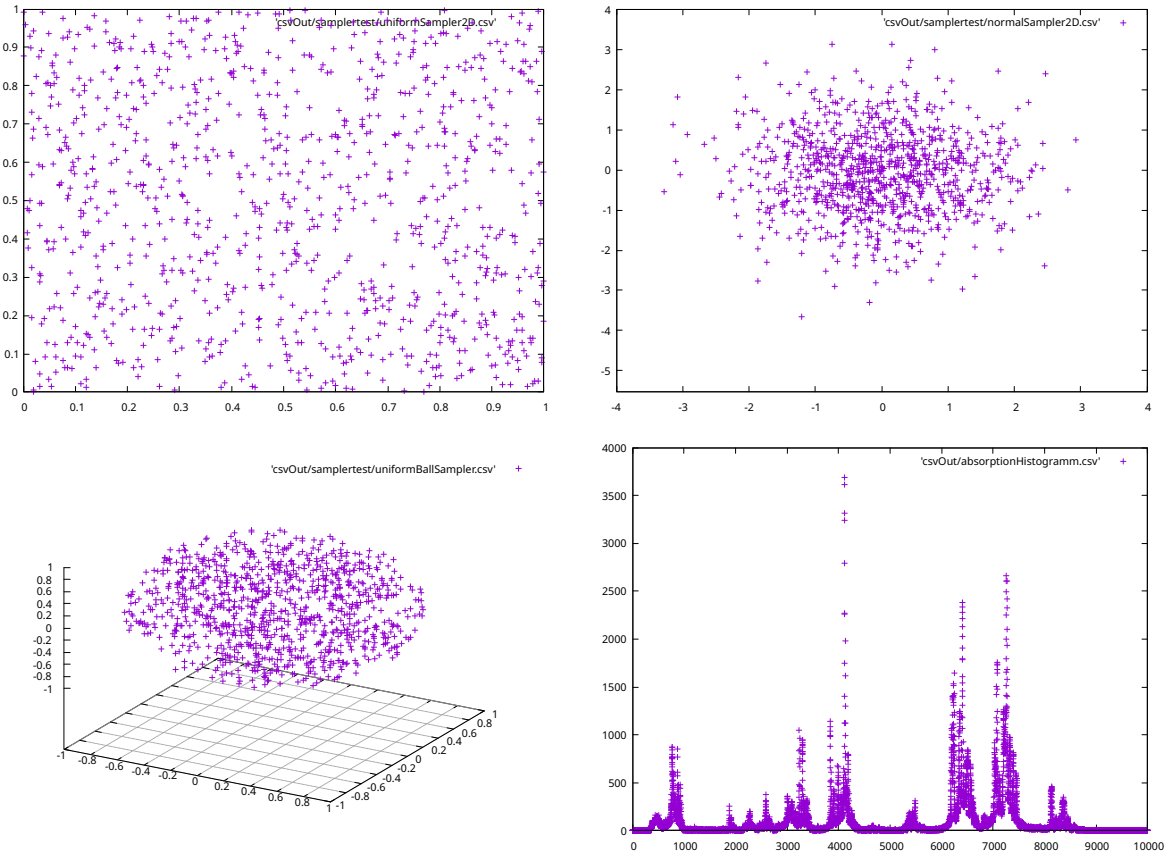

Figure 10: Different examples of sampler output. Two dimensional samplers using the UniformSamplerND (left) and NormalSamplerND (right) classes. A three dimensional uniform ball using the UniformBallSampler class (bottom left) and a histogramm of an importance sampled distribution showing the absorption coefficient for an Nd:YAG crystal (bottom right). Higher value means that the respective index has been sampled more often.

References

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

[2] John Amanatides, Andrew Woo, et al. A fast voxel traversal algorithm for ray tracing. In *Eurographics*, volume 87, pages 3–10, 1987.

[3] C. Audet and J.E. Dennis, Jr. Mesh adaptive direct search algorithms for constrained optimization. *SIAM Journal on Optimization*, 17(1):188–217, 2006.

[4] GJCL Bruls. Exact formulas for a thin-lens system with an arbitrary number of lenses. *Optik*, 126(6):659–662, 2015.

[5] Laura Corner. Introduction to laser physics. https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwjuxZK2tvP2AhXO8LsIHU-kCugQFnoECAYQAQ&url=https%3A%2F%2Findico.cern.ch%2Fevent%2F759579%2Fcontributions%2F3184719%2Fattachments%2F1810600%2F3003248%2FL_Corner_Introduction_to_Laser_Physics.pdf&usg=AOvVaw0n0-uNGRjVKHzUSAAaXGkl, 2019. [Online; accessed 1-April-2022].

[6] S. Le Digabel. Algorithm 909: NOMAD: Nonlinear optimization with the MADS algorithm. *ACM Transactions on Mathematical Software*, 37(4):1–15, 2011.

[7] Dawei Liang and Joana Almeida. Highly efficient solar-pumped nd: Yag laser. *Optics express*, 19(27):26399–26405, 2011.

[8] RP Photonics. Abcd matrix. https://www.rp-photonics.com/abcd_matrix.html. [Online; accessed 7-April-2022].

[9] RP Photonics. Four-level and three-level laser gain media. https://glm.g-truc.net/0.9.4/api/index.htmlhttps://www.rp-photonics.com/four_level_and_three_level_laser_gain_media.html. [Online; accessed 1-April-2022].

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

[11] A. E Siegman. *Lasers*. Mill Valley, Calif.: University Science Books, 1986.

[12] Multiphysics Laser Simulation Software. Asld. http://www.asldweb.com/index.html.

[13] Wikipedia. Solar-pumped laser. https://en.wikipedia.org/wiki/Solar-pumped_laser. [Online; accessed 2-April-2022].

[14] Takashi Yabe, Kunio Yoshida, and Shigeaki Uchida. Demonstrated fossil-fuel-free energy cycle using magnesium and laser. In *International Congress on Applications of Lasers & Electro-Optics*, volume 2007, page M1103. Laser Institute of America, 2007.