Diese Arbeit wurde vorgelegt am
Lehr- und Forschungsgebiet Theorie der hybriden Systeme

Effiziente Simulation von kleinen Spiegel in Solarturmkraftwerken

Efficient Simulation of Small Mirrors in Central Receiver Systems

Bachelorarbeit
Informatik

October 2025

Vorgelegt von Presented by	Andrii Podriez Matrikelnummer: 421002 andrii.podriez1@rwth-aachen.de
Erstprüfer First examiner	Prof. Dr. Dr. h.c. Erika Ábrahám Lehr- und Forschungsgebiet: Theorie der hybriden Systeme RWTH Aachen University
Zweitprüfer Second examiner	Prof. Dr. rer. nat. Jürgen Giesl Lehr- und Forschungsgebiet: Programmiersprachen und Verifikation RWTH Aachen University
Betreuer Supervisor	Dr. rer. nat. Pascal Richter Lehr- und Forschungsgebiet: Theorie der hybriden Systeme RWTH Aachen University

Contents

1 Introduction

Global energy demand grew by 2.2% in 2024, while energy-related CO_2 emissions reached a new high. Meanwhile, renewable energy reaches record deployment and generation, but current pace remains insufficient to put emissions on a sustained downward path [12, 10]. Thus, systems that can provide clean, dispatchable power are increasingly valuable among various renewable energy sources. Central receiver systems (CRS) use a field of movable mirrors called heliostats to concentrate solar radiation (flux) onto a receiver mounted on a central tower. The absorbed flux heats a working fluid that can be buffered in thermal storage, electricity is then generated when needed, enabling demand-following operation. Figure 1.1 shows the PS10 central receiver system located in Spain.

Figure 1.1: PS10 solar power plant, central receiver system located in Andalusia, Spain. The heliostat field focuses sunlight onto a tower-mounted receiver, where heat is collected for storage and electricity generation.[1]

However, CRS plants are large, capital-intensive projects that take years to build. They require substantial land, a tall tower and receiver, hundreds or even thousands of heliostats, thermal storage, and must operate within tight thermal and mechanical limits. Small design choices in field layout, receiver geometry, aiming strategy, and storage sizing cascade into construction cost, operability, and lifetime performance. Because full-scale prototyping is infeasible and field experiments are slow, disruptive, and expensive, design and operation must be optimized largely before construction. The SunFlower simulation tool addresses these needs by simulating CRS power plants using various ray tracing methods. These simulations provide the means of constructing and optimizing CRS layouts as well as estimating the annual power output.

[1] Photo by afloresm, Flickr, https://www.flickr.com/photos/afloresm/1448540890/in/photostream/

1.1 Motivation

In SunFlower, the core optical elements of the CRS: heliostat field and the receiver tower are part of the simulation scene with terrain elevations taken into account. However, unlike in other similar tools, there is no way to introduce other environmental obstructions. Although CRS are usually built on a plain field, there are other structures besides heliostats and the tower: maintenance platforms, stairways, control rooms, storage tanks, pipe racks, other buildings, fences and possibly even vegetation or trees that may grow on the site. Whether designing a new CRS or optimizing the existing system, we want to ensure that these elements do not intercept sunlight that would otherwise reach mirrors or the receiver. Concentrated solar beams could potentially damage any constructions besides the receiver, as surfaces may be overheated, creating safety risks. Furthermore, obstructions can cause energy losses, as solar radiation that was blocked by other objects could otherwise reach the receiver and contribute to the output power.

To address these issues, obstacles need to be embedded directly in the simulation scene. Since we evaluate the optics of the CRS by tracing large numbers of rays, simulating sunlight that is reflected by heliostats and concentrated on the surface of the receiver, the representation of other obstructions must be both expressive and computationally cheap. High-detail meshes or arbitrary CAD solids are ill-suited for this job: they are heavy to load, taking a lot of memory and it is expensive to calculate intersections between them and the rays. Instead, simple analytical primitives with closed-form intersection tests provide efficiency in both tracing calculations and memory usage. Furthermore, approximating the structure's spatial footprint with geometric primitives is a simpler task as opposed to modeling the full model geometry. Geometrical objects like finite cylinders and planar polygons can effectively cover a wide range of structures (poles, walls, fences, buildings, trees). Complex objects could be approximated by a small set of these primitives with configurable fidelity.

There are some minor outstanding issues within SunFlower, as implemented parametrization of heliostats' reflective surfaces is inconsistent in some of the services. In particular, 3D-Printing and Tonatiuh script generation tools are not functioning currently, as they are incompatible with new parameter scheme for the small mirrors, also known as facets. These services underpin the visualization and external cross checks, so restoring compatibility is necessary not only for this work, but for SunFlower in general. Furthermore, parameter scheme for heliostats requires large lists specifying every single small mirror on the surface. Providing automatic facet-generation service would enable a compact specification from which the full, positioned list of facets is generated automatically, reducing manual input and errors.

1.2 Related work

There are other ray tracing tools for CRS simulation. Tonatiuh [3] is an open-source Monte Carlo ray tracer for concentrating solar systems that supports power-tower geometries and scriptable placement of analytic primitives (cones, cylinders, planes) that enables simulation of on-site obstructions. SolTrace [15] is an NREL Monte Carlo ray tracer for concentrating solar optics that supports tower configurations and lets users represent on-site obstructions via analytic surfaces and polygonal apertures, as well as imported geometry.

SunFlower was originally introduced by Richter et. al. [13]. Comprehensive overviews of ray tracing methods, checks for shading and blocking by other heliostats and the tower structure is given by Hövelmann [9]. Furthermore, overview of different sun shapes and mean efficiencies calculated within the model is given by Hövelmann et. al. [7, 8]. Schütte [14] extends SunFlower simulation tool with a new Tilt-and-Roll heliostat type and emphasizes fast, accurate shading and blocking checks via triangular discretization with precomputed candidate lists and AABB-tree traversal, validated using a two-heliostat flux-map test. Aldenhoff [2] further extends the simulation tool by implementing CUDA-based GPU acceleration of ray tracing procedures.

1.3 Contribution

In this work we extend SunFlower with an obstacle model that enables rendering of environmental obstructions during simulations, evaluating their impact on the CRS. Within this model, two obstacle types are introduced: finite cylinders and planar polygons. We further validate the correctness of new model and its impact on the runtime. Additionally, we resolve some outstanding issues within the simulation tool. We finish integration of facet parameter schema for 3D-Printing and Tonatiuh export services. Furthermore, we implement an additional service that provides automatic generation of per-facet parameters.

1.4 Outline of this work

First, a brief overview of optical model and raytracing methods of SunFlower is given in Section 2. Then in Section 3 we introduce the obstacle model with two types of obstacles introduced within it: finite cylinders and polygons, with their implementations covered in Sections 3.1 and 3.2 respectively. We discuss the further integration of the model into the optical model in Section 3.3. Afterwards, in Section 4 we briefly cover some of the facet-related issues that were resolved within this work. Finally, we validate and analyze the obstacle model in Section 5 and summarize our findings in Section 6.

2 Simulation environment

In SunFlower the simulated CRS scene is defined using a global Cartesian system in which x-axis points towards the East, y-axis towards the North and z-axis towards the sky. In addition, meters are used as base units in this system. We further describe the site of CRS by specifying coordinates that correspond to the boundaries of the field. Afterwards we can provide elevation profile of the terrain as a grid of height values and specify positions of heliostats and the receiver tower. This configuration forms the foundational basis of every simulation. Given this setup we can now specify parameters of the heliostats and the receiver, as well as characteristics of the incoming sunlight, in order to finally calculate the approximate amount of solar radiation reflected from heliostats on the receiver, subsequently computing the output power.

2.1 Optical Model

The distribution of the sunlight over the simulated power plant is described using different Sun shape profiles: Gaussian, Buie, Pillbox and custom, user-defined distribution. These distributions model the spatial spread of the Sun rays that enter the simulation scene. A concise overview of these distributions is given in [8].

Heliostats are sun-tracking mirrors that continuously adjust the surface normal to bisect the directions to the Sun and the receiver, reflecting solar radiation towards the receiver. The reflective surface is composed of small mirrors called facets. Facets can have either rectangular or other polygonal shapes and their positions on heliostat are defined using a local coordinate system.

Receiver is mounted atop the tower and absorbs sunlight reflected by heliostats, converting solar radiation into thermal energy. Different receiver surfaces can be modeled, among them flat tilted, cylindric cavity and cylindric external types. An example of tower with a modeled external cylindric receiver surface is illustrated in Figure 2.1.

Based on the specified positions and configurations of heliostats and the receiver tower within the boundaries of a particular site, we can compute what portion of solar radiation is absorbed by the receiver and converted into power. This can be performed by conducting a simulation either for a single point in time with a fixed sun position or for the duration of an entire year. In both cases, we rely on a chosen ray tracing method to simulate propagation and reflection of sun rays. Analytic, Monte Carlo, convolution strategies are available on both CPU and GPU. A general overview of these methods is given in [9], while GPU-based implementations and performance optimizations are presented in [2]. In the following, we summarize these ray tracing techniques and their respective approaches.

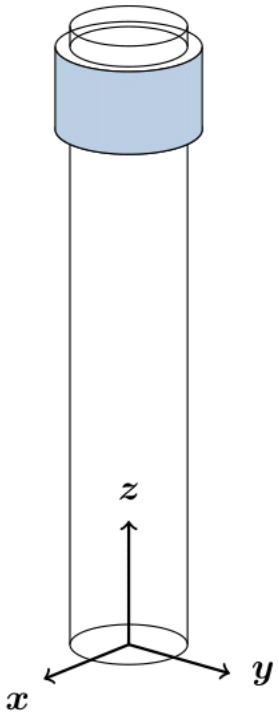


Figure 2.1: Example of a solar tower with cylindrical receiver, taken from [9].

2.2 Ray Tracers

In each ray tracing technique we first subdivide each facet into small cells. For every cell we define an ideally reflected ray and process it according to the chosen ray tracer.

Analytic We deterministically trace the ideal reflected ray towards the Sun and the receiver based on the law of reflection. Computing the reflections directly makes this method computationally efficient, but simultaneously, without introduction of any natural deviations, it does not accurately reproduce real-world behavior.

Monte Carlo Extends the analytic approach by introducing random sampling of rays. This way we can now account for real-world effects using statistical error models, among them slope deviations, tracking inaccuracies and the finite angular extent of the Sun, as rays are not perfectly parallel in real life. An extended method, **Multi-Monte Carlo** ray tracer, further improves statistical reliability by tracing multiple perturbed rays for each cell and then averaging their contribution. Consequently, accurate results are delivered in accordance with the law of large numbers, becoming more accurate with larger amounts of rays, but at the cost of increased processing time.

Convolution This method models a bivariate Gaussian distribution around the ideal reflected ray in order to describe how flux from the entire cell area is distributed over

the receiver. This approach allows us to model real-world deviations, but without relying on a large number of rays for accuracy, therefore achieving realism comparable to the Monte Carlo approach, but with deterministic results and lower computational cost. While this makes the convolution technique highly efficient and robust for optical simulations, the Monte Carlo method remains essential in cases where complex geometries or non-Gaussian error models exceed analytical assumptions of the convolution method. Additionally, an extended variant, the **integrated convolution** ray tracer, further refines this approach, averaging the reflected flux over the finite cell area.

2.3 Shading and Blocking

Calculated reflected solar radiation is affected by different factors, called **mean efficiencies**. These effects cause losses of the incoming solar radiation along the way. By multiplying the incoming power factor with the calculated mean efficiencies we can compute the factor of resulting output power. Solar power per unit area is labeled as I_{DNI} (Direct Normal Irradiance) and measured in $\text{W} \,/\, \text{m}^2$. The ideal reflected solar power P_{in}, without environmental and physical effects given area of a single facet cell A_{cell} can be computed as

$$P_{\text{in}} = I_{\text{DNI}} \cdot A_{\text{cell}}$$

Then, we can calculate individual mean efficiencies and multiply P_{in} with them, in order to compute output power after losses P_{out}. A concise overview of this approach as well as its evaluation is given in [7]. We can now compute the solar power concentrated on the receiver P_{out} as described in the Equation 1.

$$P_{\text{out}} = P_{\text{in}} \cdot \eta_{\text{cos}} \cdot \eta_{\text{ref}} \cdot \eta_{\text{aa}} \cdot \eta_{\text{sp}} \cdot \eta_{\text{ts}} \cdot \eta_{\text{shading}} \cdot \eta_{\text{blocking}} \tag{1}$$

Following is a summary of the mean efficiency factors:

- **Cosine efficiency** (η_{cos}): based on the cosine effect [16], accounts for reduction of reflected solar radiation, due to the tilt of the heliostat's reflective surface relative to the incoming Sun rays.
- **Reflectivity efficiency** (η_{ref}): accounts for the fact that mirror surfaces are not perfectly reflective, typically due to properties of the material and surface degradation.
- **Atmospheric attenuation efficiency** (η_{aa}): represents the fraction of energy lost by absorption and scattering of the reflected rays in the atmosphere, depending on the distance between heliostat and receiver.
- **Spillage efficiency** (η_{sp}): represents the fraction of reflected rays that actually reach the surface of the receiver, as rays may be reflected incorrectly due to optical errors.

- **Tower shading efficiency** (η_{ts}): accounts for losses caused by the tower structure shading the heliostat mirrors.
- **Shading efficiency** (η_{shading}): similarly to tower shading, it is possible that the heliostat mirrors may shade each other.
- **Blocking efficiency** (η_{blocking}): heliostat mirrors can also block the flux reflected towards the receiver.

Shading efficiencies are computed by checking the traced Sun ray for intersections. In any of the ray tracing methods a ray is cast towards the Sun from a cell of each facet and is traced for intersections with the tower structure or other heliostats. If this ray is indeed intersected, we add area of such cells A_{cell} together to A_{shaded}. Then, given area of every facet cell on every heliostat in the field A_{total} we compute the shading mean efficiency

$$\eta_{\text{shading}} = \frac{A_{\text{shaded}}}{A_{\text{total}}} \tag{2}$$

Similarly, for each cell ideal reflected (in analytic or convolution ray tracer) or perturbed (in Monte Carlo approaches) rays, aimed at the receiver, are traced for intersections with other mirrors or with the tower. In case they indeed intersect other heliostats, we add their respective cell areas towards A_{blocked} and then compute blocking mean efficiency

$$\eta_{\text{blocking}} = \frac{A_{\text{blocked}}}{A_{\text{total}}} \tag{3}$$

These procedures are already implemented for every CPU and GPU ray tracer with the exception of convolution and integrated convolution on GPU [2]. A detailed overview of tracing heliostat shading and blocking cases is given and evaluated in [9, 14]. Our goal is now to expand these shading and blocking checks to include intersections with other obstructions located on the field. Through accumulating cell area blocked or shaded by these obstacles we can expand mean efficiencies from Equation 1, introducing losses caused by shading and blocking of heliostats caused by structures other than another heliostat or the tower structure.

3 Obstacles

Given either an incoming Sun ray or a ray reflected from facet cell defined in Equation 4

$$r(t) = p_r + t \cdot \vec{v_r}, \quad t \geq 0 \tag{4}$$

we introduce obstacle model, with a single obstacle specified as a tuple

$$O = (p_0, P, \text{isShading}, \text{isBlocking})$$

where

p_0 is a reference point to the obstacle's position within the global Cartesian system

P describes a set of geometric parameters, defining the shape, dimensions and orientation of the obstacle within the global Cartesian system, placed at p_0 within the site of the simulated power plant.

isShading $\{r(t)\} \longrightarrow \{\texttt{false}, \texttt{true}\}$ is a tracing procedure defined as

$$\text{isShading}(r(t)) = \begin{cases} \texttt{true}, & \text{if } \exists\, t \geq 0 \text{ such that } r(t) \text{ intersects } O, \\ \texttt{false}, & \text{otherwise.} \end{cases}$$

isBlocking $(r(t), (p_t \in \mathbb{R}^3)) \longrightarrow \{\texttt{false}, \texttt{true}\}$ functions similarly to isShading, but additionally ensures that ray intersects O before it reaches target point p_t

$$\text{isBlocking}(r(t), p_t) = \begin{cases} \texttt{true}, & \text{if } \exists\, t \text{ with } 0 \leq t < t_{\text{target}} \text{ such that } r(t) \text{ intersects } O, \\ \texttt{false}, & \text{otherwise.} \end{cases}$$

This structure allows us to locate an obstacle within the simulaton scene together with heliostats and the receiver tower. We can then utilize isShading and isBlocking procedures of every obstacle when performing shading and blocking checks. Abstract nature of this model allows its extension with any geometrical object, given it can be specified and placed within the global Cartesian system and procedures isShading, isBlocking are implemented accordingly. In the following we discuss the implementation of two obstacles introduced within this work, namely polygonal and cylindrical obstacles.

3.1 Polygonal Obstacle

Polygonal obstacle represents a planar polygon whose vertices are defined within its own local two-dimensional coordinate system. This polygon can then be embedded into the global Cartesian system on a specified plane. Parameters P of polygonal obstacle are specified as followed:

$V = \{v_1, v_2, \ldots, v_k\}, \vec{v_i} \in \mathbb{R}^2$ Ordered list of k two-dimensional vertices that define the polygon on a local 2D coordinate system by forming edges from consecutive vertices i.e. edges $(v_i, v_{i+1})v_i, v_{i+1} \in V, i \in \{1, \ldots, k\}$ and the last closing edge (v_k, v_1)

$\vec{v_n} \in \mathbb{R}^3$ Vector that defines rotation of the polygon in 3D space by, along with p_0, defining plane of the local coordinate system. Consequently, changing values of this vector defines rotation of the polygon in space. Any non-zero vector can be chosen. It is then normalized i. e. $\vec{v_n} = \frac{\vec{v_n}}{\|\vec{v_n}\|}$

Additionally, in case of the polygonal obstacle, its position p_0 within the global Cartesian system coincides with the origin $(0, 0)$ of the obstacle's local coordinate system. The plane that hosts the local coordinate frame and consequently the vertices located on it is thus defined as

$$\Pi : (x - p_0) \cdot \vec{v_n} = 0 \tag{5}$$

The local coordinate system of the obstacle is derrived from an orthonormal basis construction, which yields vectors $\vec{v_x}, \vec{v_y} \in \mathbb{R}^3$. This step is crucial, as with these axes projection of 3D points onto local 2D frame and vice versa becomes possible. While any pair of orthogonal vectors lying on Π could serve the role of $\vec{v_x}, \vec{v_y}$, it is important to define $\vec{v_x}$ and $\vec{v_y}$ in a deterministic manner, as for every possible pair of p_0 and v_n it must be clear, where exactly will a polygon defined in V be located in global Cartesian system.

We construct orthonormal basis by first defining $\vec{v_x}$ as a vector both orthogonal to $\vec{v_n}$ and parallel to the XY plane.

$$\vec{v_x} = (-\vec{v_n}.y, \vec{v_n}.x, 0)^T$$

Then we compute $\vec{v_y}$, by performing a cross-product

$$\vec{v_y} = \vec{v_n} \times \vec{v_x}$$

Afterwards we enforce a consistent orientation, by flipping both $\vec{v_x}$ and $\vec{v_y}$ in case of $\vec{v_y}.z < 0$. This ensures a right-handed orthonormal basis with v_y axis pointing upwards along global $Z-$axis and v_x being parallel to the ground. However, this transformation will not work in a case of $\vec{v_n}$ being orthogonal to the XY plane i. e. polygon lies on the XY plane. In case of $\vec{v_n} = (0, 0, 1)$ we simply set $\vec{v_x}$ to $(1, 0, 0)^T$ and $\vec{v_y}$ to $(0, 1, 0)^T$. Similarly, for $\vec{v_n} = (0, 0, -1)$ we can build orthonormal basis by choosing $\vec{v_x} = (1, 0, 0)^T$ and $\vec{v_y}$ to $(0, -1, 0)^T$. Finally, we normalize resulting $\vec{v_x}$ and $\vec{v_y}$ to unit length to preserve scale when transforming between both local and global Cartesian systems. In this way we construct orthonormal basis in a consistent and intuitive way. Figure 3.1a depicts a trapezoid in 2D with vertices $V = \{(0, 0), (1, 3), (3, 3), (4, 0)\}$. By specifying the plane on which we place this polygon in 3D with $p_0 = (5, 0, 2)$ and $\vec{v_n} = (0, 3, 2)^T$, we can now compute v_x and v_y and project each vertex onto the global Cartesian system

$$\forall (x, y) \in V : v_{3D} = p_0 + \vec{v_x} \cdot x + \vec{v_y} \cdot y$$

The resulting projected polygonal obstacle is depicted on Figure 3.1b. Although this construction depicts the exact process of embedding the two-dimensional polygon, it is included mainly as an explanatory model of how polygonal obstacle is defined given the input parameters. During the simulation we do **not** project each vertex onto 3D. As shown later it suffices to project the relevant ray-plane intersection point onto the obstacle's local frame and to map polygon's centroid back to 3D. However, this contruction is still utilized during visualization as will be shown later, when discussing 3D-Printing service in Section 4.1.

The polygonal obstacle is now fully defined and can be located within the simulated scene. In the following we introduce the implementation of isShading and isBlocking procedures, based on a constructed polygonal obstacle.

3.1.1 Ray-Plane Intersection

We begin a tracing procedure with a ray-plane intersection step that determines whether the ray intersects the plane Π of the obstacle. At the start, we compute the intersection point between $r(t)$ and Π in 3D and proceed to project it onto obstacle's local coordinate system. Implementation of ray-plane intersection follows a standard approach described by Akenine-Möller et al. [1, pp. 966–970]. We substitute our ray equation (4) into the plane equation (5)

$$t = \frac{(p_0 - p_r) \cdot \vec{v_n}}{v_n \cdot \vec{v_r}}$$

Two special cases were considered in the implementation of the algorithm. First, we check for $\vec{v_n} \cdot \vec{v_r} = 0$, as dot product being equal to 0 means that plane's normal and ray's direction are orthogonal, therefore ray is parallel to the plane and there can not be any intersection point. Second, we check for t being positive as stated in the ray equation (4), as negative t would mean that the intersection point lies in the opposite direction of $\vec{v_r}$, invalidating resulting intersection. Finally, t can be substituted into the ray equation (4) to calculate the intersection point p_i.

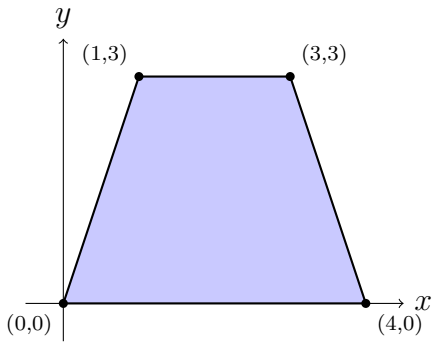
$$p_i = p_r + t \cdot \vec{v_r}$$

Finally, we can project p_i onto local coordinate system in order to determine whether the polygonal surface was intersected. As local frame was defined previously by axes $\vec{v_x}$, $\vec{v_y}$ and by origin point p_0, we are now able to project p_i onto the local frame, computing $p_i' \in \mathbb{R}^2$

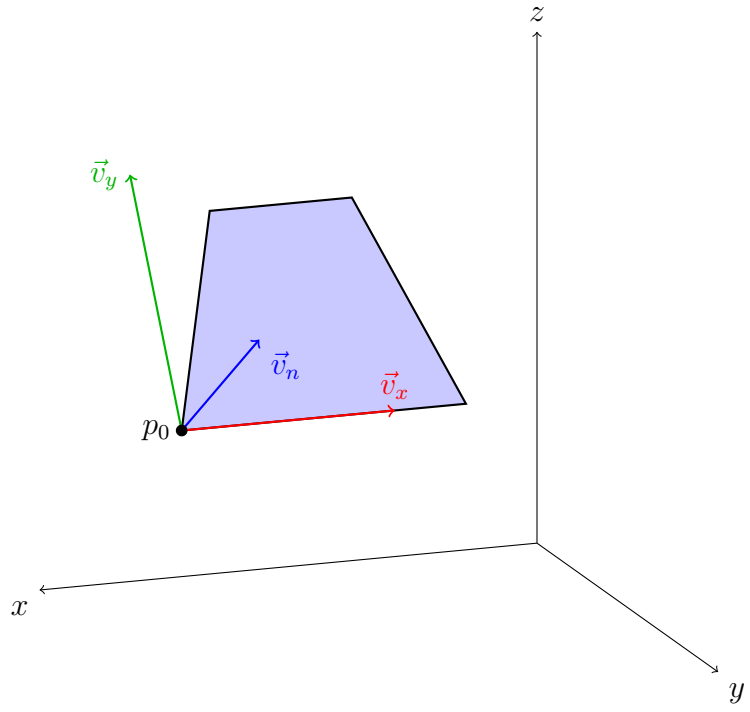
$$p_i' = ((p_i - p_0) \cdot \vec{v_x}, (p_i - p_0) \cdot \vec{v_y})$$

3.1.2 Point-In-Polygon Test

In order to determine whether p_i' lies within the boundaries of a polygon described by vertices in V, we use the crossings (even-odd) test: cast a ray from p_i' in the $+\vec{v_x}$



(a) 2D polygon with vertices $V = \{(0,0),(1,3),(3,3),(4,0)\}$.



(b) Polygonal obstacle with local coordinate axes $\vec{v}_x$ and $\vec{v}_y$.

Figure 3.1: Polygon obstacle shown in (a) 2D and (b) 3D with its local coordinate frame.

direction and iterate the edges, toggling an `inside` flag each time the ray crosses an edge whose endpoints lie on opposite sides of the horizontal line through p'_i. Algorithm 1 follows a compact approach described in [1, pp. 966–970].

To treat points on the edges as points within the polygon, we have extended the original approach by adding an explicit edge test before toggling `inside` flag: for each edge (e_1, e_2) with $e_1, e_2 \in V$, we compute

$$m = (p'_{ix} - e_{1x}) \cdot (e_{2y} - e_{1y}) - (p'_{iy} - e_{1y}) \cdot (e_{2x} - e_{1x})$$

if $m = 0$ and p'_i lies between e_1 and e_2 along both coordinate axes $\vec{v_x}, \vec{v_y}$, this means that p'_i lies exactly on the polygon's edge (e_1, e_2). In this case we stop the procedure and mark ray as intersected by the obstacle, returning `true`. Otherwise, we proceed with the crossings test.

In summary, once a ray-plane intersection is found and projected onto obstacle's local coordinate system, we can now determine if it is located within the boundaries of the polygon, deciding whether ray misses or hits the obstacle.

3.1.3 Potential Hit Check

Although the resulting polygon ray tracing algorithm can correctly determine whether there was a hit, performing it for each ray during a simulation can be inefficient, as in practice, while we can have some heliostats being shaded or blocked by the polygonal obstacle located in close proximity to them, many other heliostats in the field might be located too far away from it, with rays destined to fail the point-in-polygon check by a large margin. To filter out those unnecessary computations, a simpler check is performed first. This is achieved by building a minimal bounding sphere around the obstacle. We place center of this sphere at the centroid of our polygon and setting sphere's radius to a distance from the centroid to a furthest vertex. This way we can simply check if a shortest distance between a ray and sphere's center is greater than its radius. If so, we can skip the ray-plane intersection and the point-in-polygon tests, since the ray does not cross the bounding sphere.

We compute the 2D centroid of the polygon $(C_x, C_y) \in \mathbb{R}^2$ given its vertices $V = \{(x_i, y_i)\}_{i=0}^{N-1}$ using the formulas from Bourke [4]. We iterate edges, computing the area of the polygon

$$A = \tfrac{1}{2} \cdot \sum_i (x_i y_{i+1} - x_{i+1} y_i)$$

and then obtain coordinates of the centroid (C_x, C_y)

$$C_x = \frac{1}{6A} \sum_i (x_i + x_{i+1}) \cdot (x_i y_{i+1} - x_{i+1} y_i)$$

Algorithm 1 Point-in-polygon test

```
1: function IsPointInsidePolygon(V, p'_i)
2:     inside ← false
3:     n ← |V|
4:     e_0 ← V[n − 1]
5:     y_0 ← (e_0.y ≥ p'_i.y)
6:     for i ← 0 to n − 1 do
7:         e_1 ← V[i]
8:         e_2 ← V[(i + 1) mod n]
9:         m ← (p'_i.x − e_1.x)(e_2.y − e_1.y) − (p'_i.y − e_1.y)(e_2.x − e_1.x)
10:        inX ← min(e_1.x, e_2.x) ≤ p'_i.x ∧ max(e_1.x, e_2.x) ≥ p'_i.x
11:        inY ← min(e_1.y, e_2.y) ≤ p'_i.y ∧ max(e_1.y, e_2.y) ≥ p'_i.y
12:        if inX ∧ inY ∧ (m = 0) then
13:            return true
14:        end if
15:        y_1 ← (e_1.y ≥ p'_i.y)
16:        if y_0 ≠ y_1 then
17:            if ((e_1.y − p'_i.y)(e_0.x − e_1.x) ≥ (e_1.x − p'_i.x)(e_0.y − e_1.y)) = y_1 then
18:                inside ← ¬inside
19:            end if
20:        end if
21:        y_0 ← y_1;
22:        e_0 ← e_1
23:    end for
24:    return inside
25: end function
```

$$C_y = \frac{1}{6A} \sum_i (y_i + y_{i+1}) \cdot (x_i y_{i+1} - x_{i+1} y_i)$$

To handle inconsistent input cases we use a small-area guard and default to $(0, 0)$ when $|A|$ is close to 0. We then project computed centroid from local coordinates into the global Cartesian system using local axes $\vec{v_x}, \vec{v_y}$.

$$c_{3D} = p_0 + C_x \cdot \vec{v_x} + C_y \cdot \vec{v_y}$$

Finally, we obtain a bounding sphere by centering it at the polygon centroid and setting its radius to the distance from centroid to the furthest vertex:

$$R_s = \max_i \sqrt{(x - C_x)^2 + (y - C_y)^2} \quad (x, y) \in \{(x_i, y_i) \in V, i \in \{1, \ldots, |V|\}\} \cup \{(v_{|V|}, v_1)\}$$

Since the projection $c_{3D} = p_0 + C_x \vec{v}_x + C_y \vec{v}_y$ uses an orthonormal in-plane basis, Euclidean distances are preserved, so the same R_s serves as the radius of the three-dimensional bounding sphere centered at c_{3D}. We can now calculate the minimal distance from c_{3D} to $r(t)$

$$l = \frac{\|(c_{3D} - p_r) \times \vec{v_r}\|}{\|\vec{v_r}\|}$$

During the tracing procedures isShading and isBlocking we can now apply a following pre-filter procedure: in case $l > R_s$ ray does not intersect the bounding sphere, therefore we can already mark the ray as missing, by returning `false`. Otherwise, in case $l \leq R_s$, we must resume with the rest of procedure, calculating the intersection point and performing a point-in polygon test (in case ray is not parallel to the obstacle).

3.1.4 Polygonal Obstacle Shading and Blocking

Intersection check isShading proceeds in four steps:

1. We begin by performing a bounding sphere check, comparing l to the R_s
2. If bounding check has failed and $r(t)$ potentially intersects the obstacle, we calculate the intersection between Π and $r(t)$.
3. If ray is not parallel to Π and intersection point does not lie behind the ray we project p_i onto the obstacle's local coordinates, computing p_i'
4. Finally, we perform the point-in-polygon test, executing Algorithm 1

The first three stages of the algorithm operate in constant time $\mathcal{O}(1)$. Since the overall computational complexity of Algorithm 1 is $\mathcal{O}(|V|)$, tied to the number of vertices, the overall complexity of isShading is also $\mathcal{O}(|V|)$ for a polygonal obstacle. The complete procedure is summarized in Algortihm 2.

Intersection check isBlocking mainly follows the same pipeline described in the Algorithm 2, as it should also trace intersection of the polygonal obstacle by a ray $r(t)$. However, we introduce an additional step to the isBlocking procedure, in order to address a case in which target of the ray p_t does not lie ahead of the obstacle along the ray, otherwise it does not matter whether the obstacle is intersected by $r(t)$ or not, since blocking only happens in case of obstacle being located between the source of the ray (facet cell) and the intended target p_t (aimed receiver point). Therefore, we compute t_{target}, such that $r(t_{target}) = p_t$ and then we ensure that obstacle is not located behind the target, by comparing t_{target} against t that was computed during the ray-plane intersection procedure. The final isBlocking procedure of polygonal obstacle is depicted in Algortihm 3.

Algorithm 2 Shading test for a polygonal obstacle

function isShading($r(t)$)
  if $l > R_s$ **then**
    return false
  end if
  denom $\leftarrow \vec{v}_r \cdot \vec{v}_n$
  if $|\text{denom}| < \varepsilon$ **then**
    return false
  end if
  $t \leftarrow \dfrac{(p_0 - p_r) \cdot \vec{v}_n}{\text{denom}}$
  if $t < 0$ **then**
    return false
  end if
  $p_i \leftarrow p_r + t\,\vec{v}_r$
  $p_i' \leftarrow \big((p_i - p_0) \cdot \vec{v}_x,\ (p_i - p_0) \cdot \vec{v}_y\big)$
  return IsPointInsidePolygon($V,\ p_i'$)
end function

Algorithm 3 Blocking test for a polygonal obstacle

function isBlocking($r(t),\ p_t$)
  if $l > R_s$ **then**
    return false
  end if
  denom $\leftarrow \vec{v}_r \cdot \vec{v}_n$
  if $|\text{denom}| < \varepsilon$ **then**
    return false
  end if
  $t \leftarrow \dfrac{(p_0 - p_r) \cdot \vec{v}_n}{\text{denom}}$
  if $t < 0$ **then**
    return false
  end if
  $t_{\text{target}} \leftarrow \dfrac{(p_t - p_r) \cdot \vec{v}_r}{\vec{v}_r \cdot \vec{v}_r}$
  if $t_{\text{target}} \leq 0 \ \vee\ t > t_{\text{target}}$ **then**
    return false
  end if
  $p_i \leftarrow p_r + t\,\vec{v}_r$
  $p_i' \leftarrow \big((p_i - p_0) \cdot \vec{v}_x,\ (p_i - p_0) \cdot \vec{v}_y\big)$
  return IsPointInsidePolygon($V,\ p_i'$)
end function

3.2 Cylindrical Obstacle

Polygonal obstacles can model any planar polygonal barrier within the site of the CRS. However, one inherent limitation of polygonal obstacles is their lack of volume as they are unable to capture thickness of three-dimensional objects. Non-flat objects like pipes or trees can be simulated using cylinders. Cylindrical obstacle is a finite cylinder defined by the following parameters P:

$\vec{v_a} \in \mathbb{R}^3$ Vector that defines direction of cylinder's axis. Any non-zero vector can be chosen so that any orientation of the cylinder is possible. It is then normalized i. e. $\vec{v_a} = \frac{\vec{v_a}}{\|\vec{v_a}\|}$

$h \in \mathbb{R}_{>0}$ Height of the cylinder from the bottom base to the top base of the cylinder (also referred to as caps) along the axis $\vec{v_a}$.

$r \in \mathbb{R}_{>0}$ Radius of the cylinder.

In case of cylindrical obstacle, position p_0 defines the position of cylinder's bottom base center in the global Cartesian system. We can then compute p_1, which is the exact position of the center of the cylinder's top base in 3D:

$$p_1 = p_0 + \vec{v_a} \cdot h$$

3.2.1 Ray-cylinder intersection

The approach for general cylinder-segment intersection described by Ericson [6, pp. 194–198] was followed and then adapted to perform ray-cylinder intersection, accounting for the flat finite caps located at p_0 and p_1 respectively. We build an equation of a point $x \in \mathbb{R}^3$ lying on a cylinder by first taking a vector from the bottom base to x

$$\vec{v_x} = x - p_0$$

Then we define a part parallel to the axis $\vec{v_a}$

$$\vec{v_\parallel} = \frac{\vec{v_x} \cdot \vec{v_a}}{\vec{v_a} \cdot \vec{v_a}} \cdot \vec{v_a}$$

After subtracting $\vec{v_\parallel}$ from $\vec{v_x}$ we are left with a part perpendicular to the axis

$$\vec{v_\perp} = \vec{v_x} - \vec{v_\parallel}$$

Finally, we define an equation of x on infinite cylinder: a point x lies on the surface of a cylinder if squared length of $\vec{v_\perp}$ equals radius r squared.

$$(\vec{v_x} - \vec{v_\parallel}) \cdot (\vec{v_x} - \vec{v_\parallel}) = r^2 \tag{6}$$

Now, we substitute x into the infinite cylinder Equation 6 using the ray Equation 4. This yields the following quadratic equation, solving which we can determine if ray $r(t)$ intersects infinite cylinder with radius r and axis $\vec{v_a}$.

$$a \cdot t^2 + 2 \cdot b \cdot t + c = 0 \tag{7}$$

$$\begin{aligned} a &= (\vec{v_a} \cdot \vec{v_a})(\vec{v_r} \cdot \vec{v_r}) - (\vec{v_r} \cdot \vec{v_a})^2, \\ b &= (\vec{v_a} \cdot \vec{v_a})(\vec{v_m} \cdot \vec{v_r}) - (\vec{v_r} \cdot \vec{v_a})(\vec{v_m} \cdot \vec{v_a}), \\ c &= (\vec{v_a} \cdot \vec{v_a})\big((\vec{v_m} \cdot \vec{v_m}) - r^2\big) - (\vec{v_m} \cdot \vec{v_a})^2. \end{aligned}$$

Where $\vec{v_m}$ denotes a vector from p_0 to ray's origin

$$\vec{v_m} = p_r - p_0$$

Solving Equation 7 yields two possible intersection parameters

$$t_1 = \frac{-b + \sqrt{b^2 - ac}}{a},$$

$$t_2 = \frac{-b - \sqrt{b^2 - ac}}{a},$$

The corresponding intersection points on the ray are further given by

$$r(t_1),\ r(t_2),$$

which represent the respective positions where the ray intersects the surface of infinite cylinder.

Since cylindrical obstacles are modeled using finite cylinders from bottom base p_0 to the top base p_1, we need to extend the approach by adding checks for finite caps. We begin by projecting intersection points along the axis $\vec{v_a}$

$$(p_r + t_{1,2} \cdot \vec{v_r} - p_0) \cdot \vec{v_a} = (\vec{v_m} + t_{1,2} \cdot \vec{v_r}) \cdot \vec{v_a}$$

Solutions t_1, t_2 projected along the axis must not exceed the height of the finite cylinder. Otherwise they are on the infinite cylinder, but not within the finite bounds set by p_0 and p_1. Therefore, the following condition must be satisfied

$$0 \leq (\vec{v_m} + t_{1,2} \cdot \vec{v_r}) \cdot \vec{v_a} \leq h \tag{8}$$

Having described general conditions of ray intersecting a finite cylinder, we can now implement procedures isShading and isBlocking for the cylindrical obstacle.

3.2.2 Cylindrical Obstacle Shading and Blocking

We begin intersection check with $r(t)$ by first computing the coefficients a, b, c of quadratic Equation 7. First, we check if $a = 0$, which indicates that $\vec{v_r}$ is parallel to the axis $\vec{v_a}$, since

$$\begin{aligned} a &= (\vec{v_a} \cdot \vec{v_a})(\vec{v_r} \cdot \vec{v_r}) - (\vec{v_r} \cdot \vec{v_a})^2 \\ &= |\vec{v_a}|^2 \, |\vec{v_r}|^2 - \left(|\vec{v_a}| \, |\vec{v_r}| \, \cos\theta\right)^2 \\ &= |\vec{v_a}|^2 \, |\vec{v_r}|^2 - |\vec{v_a}|^2 \, |\vec{v_r}|^2 \, \cos^2\theta \\ &= |\vec{v_a}|^2 \, |\vec{v_r}|^2 \, (1 - \cos^2\theta) \\ &= |\vec{v_a}|^2 \, |\vec{v_r}|^2 \, \sin^2\theta \end{aligned}$$

From this follows that $a = 0$ exactly when for the angle θ between the two vectors $\sin(\theta) = 0$, as $|\vec{v_a}| \neq 0$ and $|\vec{v_r}| \neq 0$ are non-zero by definition, since they represent directions of the cylinder axis and the ray respectively.

Next, we check whether $c > 0$ since in ray-parallel-to-axis case, $\vec{v_r}$ has no component perpendicular to $\vec{v_a}$ and distance between two vectors remains constant along their paths. Using the definition

$$c = (\vec{v_a} \cdot \vec{v_a})\left((\vec{v_m} \cdot \vec{v_m}) - r^2\right) - (\vec{v_m} \cdot \vec{v_a})^2$$

it follows that $c > 0$ means this constant distance is larger then the radius. Therefore if $a = 0$ and $c > 0$ we can stop the procedure as ray misses, returning `false`. Otherwise, in case $c \leq 0$, ray is parallel and may intersect both caps. Intersection with the cap happens when we satisfy the equation of either cap's plane

$$(p_i - r(t)) \cdot \vec{v_a} = 0, \quad i \in \{0, 1\}.$$

Therefore we compute

$$t_0 = \frac{(p_0 - p_r) \cdot \vec{v_a}}{\vec{v_r} \cdot \vec{v_a}}$$

$$t_1 = \frac{(p_1 - p_r) \cdot \vec{v_a}}{\vec{v_r} \cdot \vec{v_a}}$$

and if either $t_0 > 0$ or $t_1 > 0$ the parallel ray intersects the planes with cap discs on them. Otherwise the intersection happens behind the ray, rendering it invalid. Now, we check whether t_0 and t_1 are located within the boundaries of the corresponding discs, defined by circles with centers at p_0, p_1 respectively and with radius r. Therefore, we confirm the cap discs hits if the following additional conditions are met

$$t_0 > 0 \wedge \|(p_r + t_0 \, \vec{v_r}) - p_0\|^2 \leq r^2$$

$$t_1 > 0 \wedge \|(p_r + t_1 \, \vec{v_r}) - p_1\|^2 \leq r^2$$

If the ray is not parallel to the axis of the cylinder, we proceed by computing the discriminant $D = b^2 - a \cdot c$. If $D < 0$, there are no real roots as ray either misses cylinder completely or as in parallel case it may intersect a cap only. In this case we need to calculate the intersection with the cap discs once again. $D \geq 0$ means we have real roots to the quadratic equations and can compute either one or two intersection points $r(t_1), r(t_2)$. If both solutions $t_1, t_2 < 0$, intersection happens behind ray's origin. Otherwise we check for the smaller positive t first, checking if it satisfies caps condition in Equation 8. In case it does not satisfy the condition, we check another root. In the end, either both roots fail and $r(t)$ does not intersect the finite cylinder, or we find a root that satisfies all of the conditions, therefore tracing a hit. Intersection check isShading can be performed by following all of the above mentioned steps. This approach is summarized in Algorithm 4.

The implementation of isBlocking for cylindrical obstacles follows the same procedure described in the Algorithm 4, but, similarly to the polygonal obstacle's case, described in 3, we compare t_{target} of p_t to every intersection parameter t computed during the tracing procedure. If the resulting $t > t_{target}$, blocking is invalidated even if cylinder is actually intersected by the ray, as during the simulation this would mean that receiver is hit before the obstacle. The resulting runtime of both isShading, isBlocking procedures of cylinderical obstacles is constant: computing the coefficients, solving a single quadratic equation and performing cap discs checks are all in $\mathcal{O}(1)$.

Algorithm 4 Shading test for a cylindrical obstacle

function isShading($r(t)$)
  $a \leftarrow (\vec{v_a} \cdot \vec{v_a})(\vec{v_r} \cdot \vec{v_r}) - (\vec{v_r} \cdot \vec{v_a})^2$
  $b \leftarrow (\vec{v_a} \cdot \vec{v_a})(\vec{v_m} \cdot \vec{v_r}) - (\vec{v_r} \cdot \vec{v_a})(\vec{v_m} \cdot \vec{v_a})$
  $c \leftarrow (\vec{v_a} \cdot \vec{v_a})((\vec{v_m} \cdot \vec{v_m}) - r^2) - (\vec{v_m} \cdot \vec{v_a})^2$
  if $|a| = 0$ **then** ▷ Ray parallel to axis case
    if $c > 0$ **then**
      return false
    end if
    $t_0 \leftarrow \dfrac{(p_0 - p_r) \cdot \vec{v_a}}{\vec{v_r} \cdot \vec{v_a}}$
    $t_1 \leftarrow \dfrac{(p_1 - p_r) \cdot \vec{v_a}}{\vec{v_r} \cdot \vec{v_a}}$
    if $(t_0 \geq 0) \wedge \left(\|(p_r + t_0\,\vec{v_r}) - p_0\|^2 \leq r^2\right)$ **then**
      return true
    else if $(t_1 \geq 0) \wedge \left(\|(p_r + t_1\,\vec{v_r}) - p_1\|^2 \leq r^2\right)$ **then**
      return true
    else
      return false
    end if
  end if
  $D \leftarrow b^2 - ac$
  if $D < 0$ **then** ▷ Ray does not hit the surface case (caps hit still possible)
    $t_0 \leftarrow \dfrac{(p_0 - p_r) \cdot \vec{v_a}}{\vec{v_r} \cdot \vec{v_a}}$
    $t_1 \leftarrow \dfrac{(p_1 - p_r) \cdot \vec{v_a}}{\vec{v_r} \cdot \vec{v_a}}$
    if $(t_0 > 0) \wedge \left\|(p_r + t_0\vec{v_r}) - p_0\right\|^2 \leq r^2$ **then**
      return true
    else if $(t_1 > 0) \wedge \left\|(p_r + t_1\vec{v_r}) - p_1\right\|^2 \leq r^2$ **then**
      return true
    else
      return false
    end if
  end if ▷ Compute the roots and check whether they are within the boundaries of the finite cylinder
  $t_1 \leftarrow \dfrac{-b - \sqrt{D}}{a}$
  $t_2 \leftarrow \dfrac{-b + \sqrt{D}}{a}$
  if $(t_1 < 0) \wedge (t_2 < 0)$ **then**
    return false
  end if
  if $0 \leq (\vec{v_m} + t_1\vec{v_r}) \cdot \vec{v_a} \leq h$ **then**
    return true
  else if $0 \leq (\vec{v_m} + t_2\vec{v_r}) \cdot \vec{v_a} \leq h$ **then**
    return true
  else
    $t_0 \leftarrow \dfrac{(p_0 - p_r) \cdot \vec{v_a}}{\vec{v_r} \cdot \vec{v_a}}$
    $t_1 \leftarrow \dfrac{(p_1 - p_r) \cdot \vec{v_a}}{\vec{v_r} \cdot \vec{v_a}}$
    if $(t_0 > 0) \wedge \left\|(p_r + t_0\vec{v_r}) - p_0\right\|^2 \leq r^2$ **then**
      return true
    else if $(t_1 > 0) \wedge \left\|(p_r + t_1\vec{v_r}) - p_1\right\|^2 \leq r^2$ **then**
      return true
    else
      return false
    end if
  end if
end function

3.3 Model integration

The routines for checking shading and blocking of the rays by other heliostats are already implemented for every ray tracing method, except for convolution methods on GPU [2]. Same shading test procedure is used in every ray tracing technique: a ray aimed at the Sun is casted from every facet cell. This ray is then traced for intersections with other heliostats and with the tower structure. On the first such intersection the cell is marked as shaded, area of the cell A_{cell} is counted towards the sum of all shaded areas A_{shaded} and all the further calculations regarding the reflected flux on the cell are stopped.

$$A_{\text{shaded}} = \sum_{i=1}^{N_{\text{cells}}} \begin{cases} A_{\text{cell},i}, & \text{if the ray of cell } i \text{ intersects other heliostat,} \\ 0, & \text{otherwise.} \end{cases}$$

Otherwise, a blocking check is performed next. In constrast to the shading case, traced ray is now depending on the chosen ray tracing method: for every facet cell we trace either the ideal reflected ray (analytic, convolution) or the perturbed ray (Monte Carlo). This introduces variability to the calculated total blocked area A_{blocked} using Monte Carlo methods with results depending on the number of traced rays: more rays make A_{blocked} more accurate.

$$A_{\text{blocked}} = \sum_{i=1}^{N_{\text{cells}}} \begin{cases} A_{\text{cell},i}, & \text{if the reflected ray of cell } i \text{ intersects other heliostat} \\ 0, & \text{otherwise.} \end{cases}$$

The overview of these blocking and shading checks is given in [9] and they are further covered and evaluated in [14]. In this work, we expand these procedures to include shading or blocking caused by obstacles located on the simulation scene. In contrast to shading and blocking intersection tests for heliostats, which use pre-calculated lists of heliostats that could potentially shade or block other heliostats, any obstacle on the field can potentially block or shade any heliostat on the field, as this is the whole point of simulating the obstructions. Therefore, we define ordered list of all obstacles within the simulation scene $\mathbb{O} = \{O_1, O_2, \ldots, O_n\}$. If this list is empty, there are no obstacles to consider during the simulation. Otherwise, it contains at least one $O_i \in \mathbb{O}$. We then can iterate through each $O_i \in \mathbb{O}$, tracing the intersections on perturbed or representative ray $r(t)$, using isShading, isBlocking procedures of O_i.

After ray $r(t)$ was checked for shading by tracing the intersections with other heliostats, we add obstacle shading evaluation checks. We iterate through $\mathbb{O}$, calling isShading($r(t)$) of each $O_i \in \mathbb{O}$. If the ray indeed does intersect the obstacle, we stop the procedure, count the area of the cell towards the sum of all cell areas shaded by obstacles $A_{\text{obstacle shaded}}$ and additionally count it towards the sum of all cell areas shaded by obstacle $O_j \in \mathbb{O}$, $A_{\text{obstacle shaded j}}$. This allows us not only to calculate the mean

efficiency and consequently the energy loss caused by obstacles, but also inspect the individual contribution of every obstale towards that loss.

$$A_{\text{obstacle shaded}} = \sum_{i=1}^{N_{\text{cells}}} \begin{cases} A_{\text{cell},i}, & \exists O_j \in \mathbb{O}, O_j.\,\text{isShading}(\text{r(t)}) = \texttt{true} \in \mathbb{O}, \\ 0, & \text{otherwise.} \end{cases}$$

$$A_{\text{obstacle shaded j}} = \sum_{i=1}^{N_{\text{cells}}} \begin{cases} A_{\text{cell},i}, & \text{for } O_j \in \mathbb{O}, O_j.\,\text{isShading}(\text{r(t)}) = \texttt{true} \ \in \mathbb{O}, \\ 0, & \text{otherwise.} \end{cases}$$

In similar fashion, we expand blocking checks to include obstacle hit tracing. For the ray $r(t)$ aimed at the receiver, we additionally take its aiming point p_t when iterating through $\mathbb{O}$ and performing procedures $(\text{r(t)}, \text{p}_\text{t})$. Unlike in shading, were ray is aimed at the Sun and every object $r(t)$ intersects is considered to be between the Sun and the mirror, in case of blocking it is possible that the obstacle is located behind the receiver. By checking if p_t is not hit before the obstacle, as previously showcased in polygonal and cylindrical obstacles' implementations, we ensure that invalid cases, in which $r(t)$ hits the receiver and only afterwards reaches obstacle's surface, are not counted as blocking. Analogously to the shading checks, we compute total cell area blocked by obstacles $A_{\text{obstacle blocked}}$ and individual contributions $A_{\text{obstacle blocked j}}, O_j \in \mathbb{O}$.

$$A_{\text{obstacle blocked}} = \sum_{i=1}^{N_{\text{cells}}} \begin{cases} A_{\text{cell},i}, & \exists O_j \in \mathbb{O}, O_j.\,\text{isBlocking}(\text{r(t)}, \text{p}_\text{t}) = \texttt{true} \in \mathbb{O}, \\ 0, & \text{otherwise.} \end{cases}$$

$$A_{\text{obstacle blocked j}} = \sum_{i=1}^{N_{\text{cells}}} \begin{cases} A_{\text{cell},i}, & \text{for } O_j \in \mathbb{O}, O_j.\,\text{isBlocking}(\text{r(t)}, \text{p}_\text{t}) = \texttt{true} \in \mathbb{O}, \\ 0, & \text{otherwise.} \end{cases}$$

After the simulation is finished, we can analyze the results to determine the impact of obstacles in $\mathbb{O}$ on the solar power plant. We have renamed the previosly covered shading and blocking mean efficiencies η_{shading}, η_{blocking}, introduced previously in Equations 2 and 3 respectively, into η_{ms}, η_{mb}, standing for mirror shading and mirror blocking respectively, as this is already what they have represented prior to the introduction of the obstacle model. We then introduce new mean efficiencies η_{os}, η_{ob}, where we utilize same total area A_{total}.

$$\eta_{\text{shading}} = \frac{A_{\text{obstacle shaded}}}{A_{\text{total}}} \tag{9}$$

$$\eta_{\text{blocking}} = \frac{A_{\text{obstacle blocked}}}{A_{\text{total}}} \tag{10}$$

This separation allows us to directly model the energy loss from obstacles, while preserving the original method for output power computation. We thus modify the Equation 1:

$$P_{\text{out}} = P_{\text{in}} \cdot \eta_{\text{cos}} \cdot \eta_{\text{ref}} \cdot \eta_{\text{aa}} \cdot \eta_{\text{sp}} \cdot \eta_{\text{ts}} \cdot \eta_{\text{ms}} \cdot \eta_{\text{mb}} \cdot \eta_{\text{os}} \cdot \eta_{\text{ob}} \tag{11}$$

Additionally, we can quantify the contribution of each individual $O_j \in \mathbb{O}$ to the overall shading and blocking efficiencies $\eta_{\text{os}}, \eta_{\text{ob}}$. We define corresponding individuall efficiencies:

$$\eta_{\text{os},\,j} = \frac{A_{\text{obstacle shaded j}}}{A_{\text{obstacle shaded}}}, \qquad \eta_{\text{ob},\,j} = \frac{A_{\text{obstacle blocked j}}}{A_{\text{obstacle blocked}}}, \tag{12}$$

While these quantities do not affect the output power computed in the Equation 11, they are crucial for simulation's statistics, as they allow breaking down of energy losses caused by obstacles, identifying which obstacles provide higher, lower or no impact. Although we compute them individually using index j of the obstacle $O_j \in \mathbb{O}$, we additionally introduce a string attribute **name**, which is assigned to each obstacle during the initialization of $\mathbb{O}$. This attribute is stored alongside other parameters and allows more clear identification of the respective efficiencies $\eta_{\text{os},\,j}\eta_{\text{os},\,j}$. Consequently, in addition to the main mean efficiencies $\eta_{\text{os}}, \eta_{\text{ob}}$, representing the energy losses caused by obstacles, we can also read and compare influence of individual obstacles on the simulation.

3.3.1 GPU Acceleration

As it was previously noted, SunFlower allows GPU acceleration for each ray tracing method. Following the approach described in [2], we employ a Translator module, that plays a role of an interface between the host (CPU) and device (GPU). This module already covers export of all other optical entities like the receiver tower and heliostat to the memory of the device. For every obstacle in $\mathbb{O}$, position p_0 and parameters P are serialized on the host's side and transferred to the corresponding implemented device's structures. After implementing the general pipeline to make transfer of every obstacle in the model possible, we have implemented structures on the device that include parameters and configurations of polygonal and cylindrical obstacles. Additionally, the intersection routines from Algorithms 2, 3 and 4 were implemented on the side of the device. Finally, we have expanded shading and blocking checks for the GPU ray tracers, following previously established methodology, making the same analysis of obstacles' impact possible.

4 Facet Parameter Scheme

Before performing evaluation of the newly introduced obstacle model, in the following we will discuss some of the other important changes made to the SunFlower simulation

tool that were a part of this work.

Each heliostat consists of one or more facets, which form its reflective surface. Each facet is described individually by specifying its **shape**: rectangular or polygonal and its **position** within heliostat's local coordinates. Additional parameters may be required depending on chosen shape and other heliostat configurations. Among them are **position** within heliostats' local coordinates system, **rotation**, facet's **normal vector**, **surface form**, **focal length**, **reflectivity** parameter, **canting** and **reference angles**.

Although this scheme for facet parametrization was already introduced and adopted prior to this work, its adoption was incomplete. In particular, the scheme is not consistently supported across all parts of the SunFlower simulation tool, causing malfunctions in 3D-Printing and Tonatiuh script generation. Furthermore, while per-facet description is highly flexible, it can become unnecessarily labor-intensive when describing heliostats with a large number of facets.

Further in this section, we introduce the necessary changes done within this work to finish integration into 3D-Printing service in Section 4.1. Then, in Section 4.2 we describe the adjustments made to correct Tonatiuh script generation service. Finally, in Section 4.3 we present a new tool that automates parameterization given only a number of facets and general parameters, reducing manual effort, while also ensuring consistency.

4.1 3D-Printing

SunFlower includes a 3D-Printing service that allows simulation setups to be exported as physical models in STL format, using the same parameters in global Cartesian system as with the simulation scene and same terrain elevation data for reference. The printing tool assembles all relevant components of a CRS such as the site topography, central receiver tower and the heliostat field into a single three-dimensional scene. This scene is then written into an STL (stereolithography) file, which is a common 3D-Printing format.

Heliostat is added to the 3D-Printing scene as a meshable object composed of a triangular surface mesh. In this representation the overall reflective surface is depicted as a single flat mirror, rather than resolving the geometry of each polygonal or rectangular facet individually. The problem arises due to the legacy implementation relying on deprecated heliostat parameters such as **facet width**, **facet height** for rectangular mirrors or **number of corners** for polygonal mirrors. It also does not make use of the per-facet parametrization introduced in the scheme. This results in the 3D-Printing service not being able to generate heliostat meshes and consequently being unable to render the whole 3D-Printing scene. In this work this issue was addressed by developing an algorithm that iterates over the parameters of all facets and computes a bounding

box that encloses them. This box and its area form the missing reflective surface part of the heliostat mesh, which resolves the problem.

Given the list of facet parameters F Algorithm 5 computes the minimal bounding box defined by its four corners $x_{\min}, y_{\min}, x_{\max}, y_{\max}$. We update those values while we iterate through every facet $f \in F$ and its following values:

- **Shape** Determines whether the facet is rectangular or polygonal. Rectangular facets are described by width and height, polygonal facets by an explicit list of corner points.
- **Position** (x_f, y_f) The location of the facet within the local heliostat coordinate system. Provides the reference point for translating facet geometry before updating the bounding box.
- **Rotation** θ_f Defines the angle of facet's in-plane rotation. As this parameter is optional, its default is $0°$ (no rotation). However, if it is specified, we need to compute rotation of facet's corner points before adjusting the bounding box.
- **Dimensions** (w_f, h_f) Width and height values specific to rectangular facets, defining their size in local coordinates.
- **Corners** C_f A sequence of points defining the boundary of polygonal facets. These points are listed in counter-clockwise order and may optionally include holes, which, however, is irrelevant for the final bounding box, as holes can only be located inside the polygons themselves.

We begin by defining the set of facet's corners P. They are already specified for polygonal mirrors under the parameter C_f and we can compute them directly for rectangular mirrors by setting the first corner to origin $(0, 0)$ and setting others based on **dimensions** parameters w_f, h_f. Each corner point $(x, y) \in P$ is then projected into heliostat's local coordinate system by applying an in-plane rotation of angle θ_f followed by translation by the facet position (x_f, y_f). This rotation corresponds to a standard 2D rotation matrix.

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} x_f \\ y_f \end{pmatrix}.$$

Finally, for every transformed corner (x', y') the bounding box limits are updated by comparing against the current values $x_{\min}, x_{\max}, y_{\min}, y_{\max}$. After all facets have been processed, reflective surface of the heliostat mesh is given by the rectangle with width $W = x_{\max} - x_{\min}$ and height $H = y_{\max} - y_{\min}$.

Algorithm 5 Heliostat surface bounding box from per-facet parameters

$x_{\min} \leftarrow +\infty,\ x_{\max} \leftarrow -\infty,\ y_{\min} \leftarrow +\infty,\ y_{\max} \leftarrow -\infty$
for each facet $f \in \mathcal{F}$ **do**
  if f.shape = rectangular **then**
    $P \leftarrow \{(0,0), (0,h_f), (w_f,h_f), (w_f,0)\}$
  else
    $P \leftarrow C_f$
  end if
  for each $(x,y) \in P$ **do**
    $x' \leftarrow x_f + (x \cdot \cos\theta_f - y \cdot \sin\theta_f); \quad y' \leftarrow y_f + (x \cdot \sin\theta_f + y \cdot \cos\theta_f)$
    $x_{\min} \leftarrow \min(x_{\min}, x'); \quad x_{\max} \leftarrow \max(x_{\max}, x')$
    $y_{\min} \leftarrow \min(y_{\min}, y'); \quad y_{\max} \leftarrow \max(y_{\max}, y')$
  end for
end for
$W \leftarrow x_{\max} - x_{\min}; \quad H \leftarrow y_{\max} - y_{\min}$
return W, H

The correctness of the solution was verified by comparing its results against existing data and STL representations available within the SunFlower simulation tool prior to the scheme change. In this earlier version, complete heliostat meshes were generated for selected power plants. After integrating Algorithm 5 into 3D-Printing tool, identical dimensions of the mirror surface were reproduced, consistent with provided facet parameter lists from the updated scheme. One example is the PS10 solar tower plant, which has 624 Sanlúcar 120 type heliostats, each composed of 4×7 grid of rectangular facets with 121 m^2 reflective surface [11]. SunFlower includes facet parameter list for PS10 heliostats, which suggest their dimensions to be 3.21 m in width, 1.35 m in height and positions coordinates describing an additional gap of 0.04 mm between the facets. Although this data results in a close surface value of 125.58 m^2 for each heliostat, it does not exactly match original PS10 data. However, the algorithm correctly calculates the dimensions of all 624 heliostats' reflective surfaces to be 12.96 m in width and 9.69 m in height. With scheme integration problem solved, 3D-Printing service can now be utilized again.

With the facet scheme integration resolved, the 3D-Printing service of SunFlower can now be utilized again. This allows us to add obstacles to the printed scene. We have implemented following procedures which convert previously introduced obstacle types into meshable, composed of triangles objects which are later added to the output STL scene at position p_0. These implementations follow similar techniques which are already applied within the 3D-Printing service to build meshable surfaces for the mirrors of the heliostats and the cylindrical receiver tower structure.

For polygonal obstacles we already have means to compute position of each individual vertex from V in 3D, described in Section 3.1. We now can simply convert the surface

into a triangle fan, where the first vertex $v_0 \in V$ acts as a common anchor, and each pair of consecutive vertices (v_{i-1}, v_i) forms a triangle together with v_0. Figure 4.1 illustrates a mesh of a polygonal obstacle from Figure 3.1b consisting of two triangles.

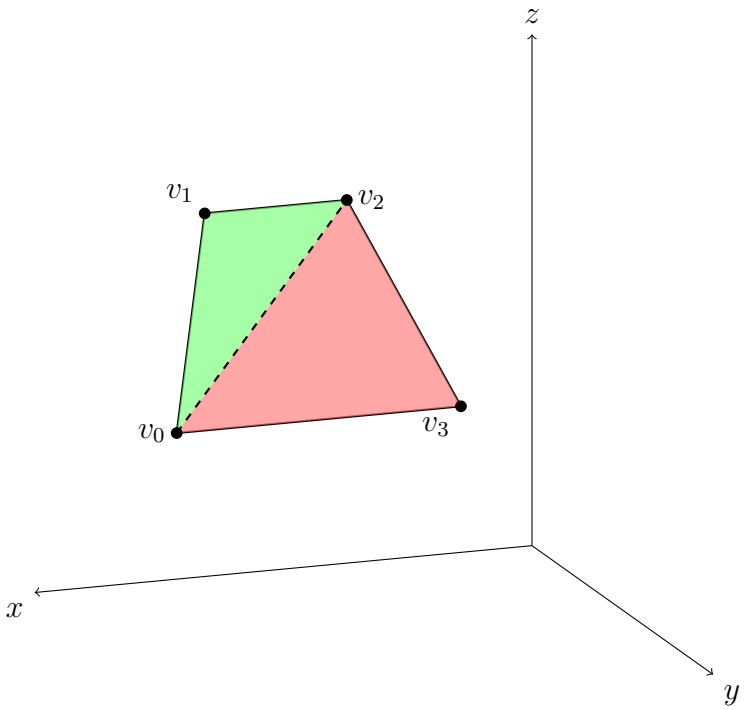


Figure 4.1: Triangle-fan mesh of a polygonal obstacle with vertices $v_0, v_1, v_2, v_3 \in V$. The surface is decomposed into two triangles: v_0, v_1, v_2 (green) and v_0, v_2, v_3 (red).

We implement a similar, although slightly more involved procedure for conversion of cylindrical obstacles into meshable, consisting of triangles cylinders. Existing implementation for the receiver tower structure constructs a meshable cone with both top and bottom radius set equal to the radius of the receiver tower. However, as previously described, cylindrical obstacle may be oriented along any direction defined by the vector $\vec{v_a}$, unlike the existing cone object, which can only be orthogonal to the ground. Therefore, while we follow the same aproach for constructing the caps of the cylinder: in order to build a circular ring using triangles, we sample a circle at fixed angular step (5°) and place two sampled rings in the planes orthogonal to $\vec{v_a}$, with their centers at p_0 and p_1. Both top and bottom caps are then built as simple triangle fans: starting at center (p_0 for the bottom cap, p_1 for the top cap), for every point sampled on the rings we add a triangle, connecting center, the point and the next sample point on the ring. Finally, we construct the lateral surface by constructing a strip of triangles between both caps: for each index k, add the triangles: one with bottom k, bottom $k+1$, top k and another with bottom $k+1$, top $k+1$, top k, repeating the first sample at the end so the side closes.

Figure 4.2 depicts an output STL with PS10 power plant scene and 5 added obstacles: an upscaled orthogonal to the ground trapezoid from Figure 3.1b and four cylinderical obstacles.

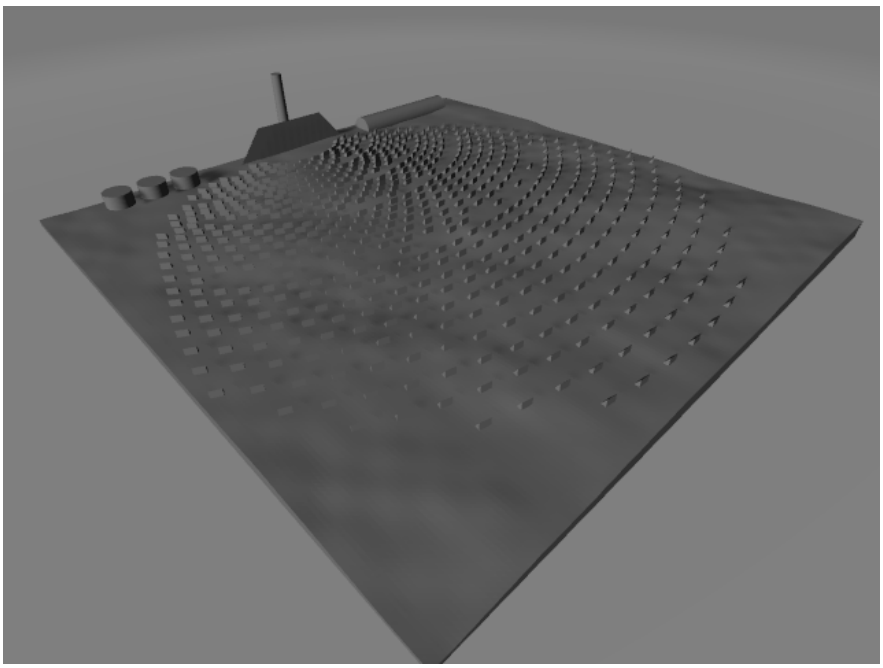


Figure 4.2: Exported STL model of the PS10 CRS with four cylindrical and one polygonal (trapezoid) obstacle. Cylinder behind the trapezoid obstacle is a separate object that represents the receiver tower. Generated by SunFlower's 3D-Printing tool and visualized in Microsoft 3D Viewer.

4.2 Tonatiuh Scripts

Tonatiuh is an open-source ray tracing software designed for the simulation of solar concentrating systems [3, 5]. It provides both a graphical interface and a script interface, which offer the means to generate and analyze complete power plant models. The SunFlower simulation tool includes a service that allows export of simulation data into Tonatiuh-compatible scripts for independent validation. However, similarly to the 3D-Printing service problem, this service used to rely on a deprecated parameter **shape** of the facets, which was defined globally for all facets in the legacy implementation. Since Tonatiuh requires each facet surface to be created with the correct geometric primitive (flat rectangle, parabolic rectangle, polygon), the script generator must explicitly distinguish between facet shapes during export. Without this check, Tonatiuh either assigns an invalid surface type or fails to render the heliostat facets properly. This issue was resolved by replacing the outdated global parameter with the current facet-specific attribute, iterating throught the facet parameters list F, thereby restoring correct facet orientation. Although the modification was small, it enabled the Tonatiuh script gen-

eration service to function properly and allowed the successful export and visualization of the PS10 solar power plant in Tonatiuh 2.2.4 (see Figure 4.3).

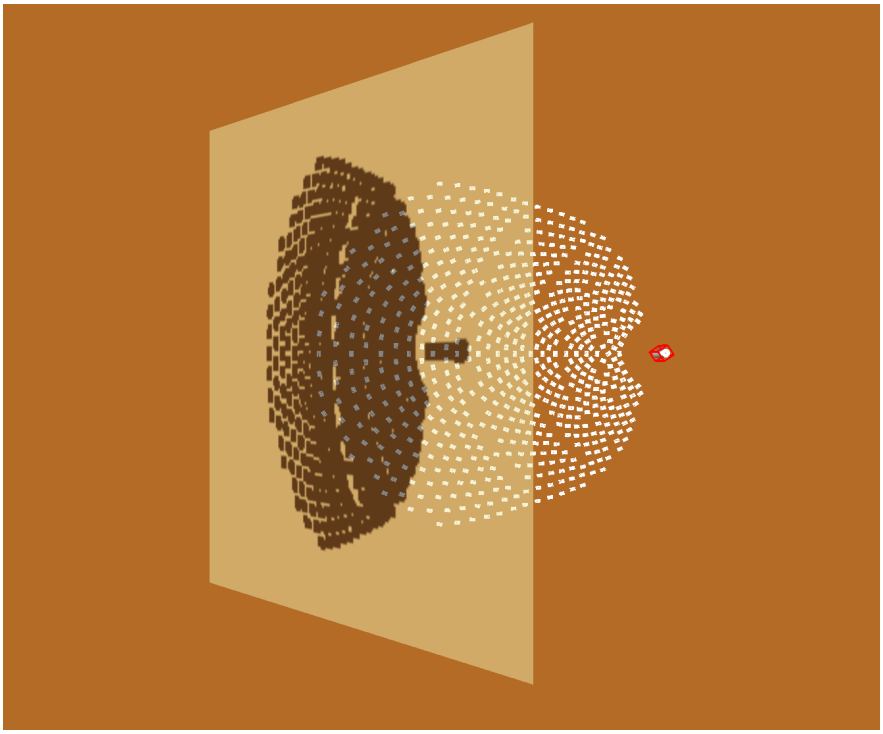


Figure 4.3: Visualization of the PS10 CRS with heliostat field, central tower receiver, and SunPlane[2], exported from the SunFlower simulation tool and rendered in Tonatiuh 2.2.4.

4.3 Automatic Facet Generation

While the established facet scheme provides a flexible description of heliostats' reflective surfaces, its direct application to multi-faceted heliostats can become problematic. Each facet must be defined individually with several previously covered specific parameters. For heliostats composed of dozens of facets, like previously mentioned in Section 4.1 Sanlúcar 120 type with its 28 facets, this may quickly lead to extensive and repetitive input structures.

To address this challenge, we have implemented an automatic generation procedure that constructs the full facet list based solely on a compact set of global configuration parameters. Instead of specifying each facet manually, the user may now specify the overall facet layout (rectangular grid or polygonal arrangement), as well as the geometric and optical parameters required for the chosen shape. From these inputs, the tool systematically derives the position, orientation, and shape description of every

[2]The SunPlane in Tonatiuh represents the surface that defines the source of incoming solar rays.

individual facet, by creating a list F filled with given number of facets. Most parameters of every $f \in F$ are identical and specified once. The new tool only adjusts their **positions** by provided global parameters that specify overall quantity of facets and a gap between them. The resulting tool serves as an alternative parameter input scheme. It was implemented utilizing procedures used previously for the legacy parameter scheme as previous model also relied on global parameters, while not being able to specify facets individually. Consequently the new service bridges the convenience of legacy parameter scheme with currently utilized scheme, presenting an alternative input model.

5 Evaluation

In this section we will review the overall correctness and efficiency of the introduced obstacle model. In Section 5.1 we design a minimal simulation scene consisting of a single heliostat and a receiver tower, into which we place polygonal and cylindrical obstacles at positions that should yield full, partial, or no shading/blocking. Further in Section 5.2 we investigate the impact of obstacles on the simulation's runtime, taking a look on the worst-case scenarios for tracing tests. All experiments were run on a workstation with an AMD Ryzen 5 5600X CPU and an NVIDIA GeForce RTX 3060 Ti GPU.

5.1 Shading and Blocking

Following the implementations of blocking and shading checks for obstacles, their validity was assessed within a controlled test environment, which consists of a single heliostat aimed at the receiver tower. We then introduce polygonal and cylindrical obstacles to this environment and systematically place them at different positions p_0, exactly where, given further parameters of each obstacle, they are expected to block or shade the heliostat either completely or partially. Additionally, we introduce cases in which obstacle is moved further away from the heliostat and the receiver, expecting it not to have any influence on the setup. For each such configuration we run three simulations: one using a convolution ray tracer and other two with a Monte Carlo ray tracer set to cast 100 rays per m^2 and a Monte Carlo ray tracer with 10000 rays per m^2. While in case of shading check all ray tracers perform same procedure on Sun rays, convolution serves as a deterministic baseline in blocking, since it checks ideal reflected, representative rays. In contrast, Monte Carlo perturbs the reflected rays and we additionally run it at 100 and 10000 rays per m^2, assessing sampling effects and convergence. Afterwards, we compare outputted statistical parameters such as $A_{\text{obstacle blocked}}$ and $A_{\text{obstacle shaded}}$ against A_{total}, ensuring the desired effect was reproduced.

5.1.1 Simulation setup

First, we define 800m × 800m field as a simulation site. Then, we add a tower of 100 m height and 2 m diameter at the center[3]. On top of the tower a 6 × 6 m flat receiver surface is mounted. At this surface we aim a single heliostat, positioned 25 m east of the tower. The heliostat has a single 1 × 1 m rectangular facet, mounted so that its center is located exactly at heliostat's position. Tracking of the Sun is performed using the standard azimuth-elevation algorithm. Since we use a single heliostat, the total reflective surface of the simulated power plant is $A_{\text{total}} = 1\,\text{m}^2$. Figure 5.1 shows a waterfall plot of energy loss breakdown, with mean efficiencies outputted as a result of running a convolution simulation with no obstacle on the field. Incident power before the losses set to $P_{\text{in}} = 100\%$. After we apply the calculated mean efficiencies we get $P_{\text{out}} = 73.12\%$. These losses are primarily contributed to the cosine effect, atmospheric attenuation, receiver spillage and a heliostat's reflectivity efficiency $\eta_{\text{ref}} = 0.88$. Now we are going to introduce polygonal and cylindrical obstacles into this environment and run the following simulations.

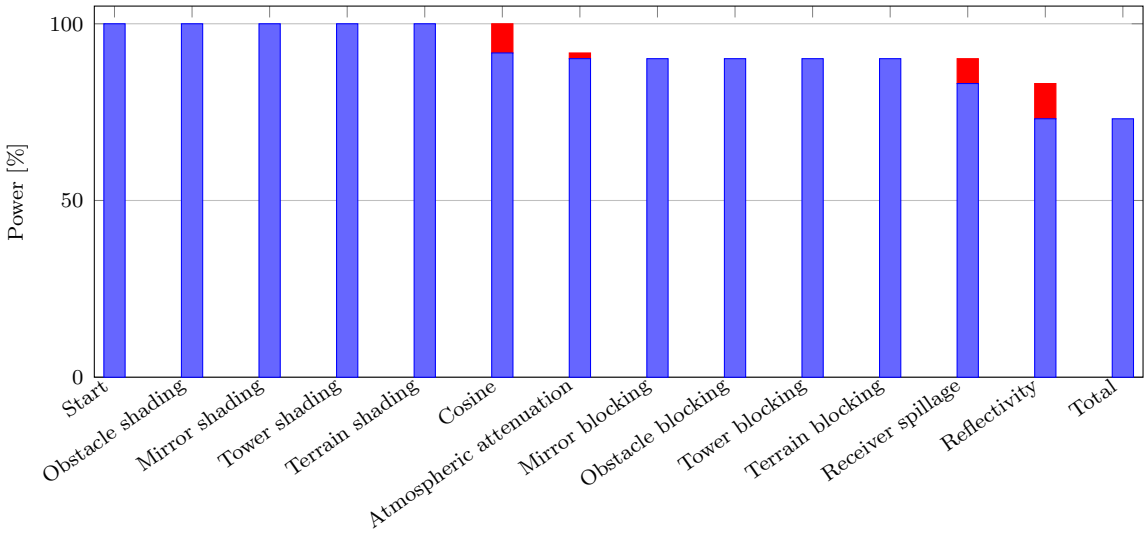


Figure 5.1: Optical losses from 100% incident power, outputted after running a simulation using CPU convolution ray tracer with no obstacles added to the scene. Blue bars represent remaining output power, while red segments are the per-step energy losses.

For the validation, we distinguish three representative shading conditions with according statistics:

1. Heliostat's reflective surface is shaded completely, expecting $A_{\text{total}} = A_{\text{obstacle shaded}}$.

[3]origin (0, 0, 0) in the global Cartesian system

2. Heliostat's reflective surface is shaded partially, expecting
   In this case, $A_{\text{total}} > A_{\text{obstacle shaded}} > 0$.

3. Heliostat's reflective surface is not shaded, expecting
   $A_{\text{obstacle shaded}} = 0$.

We then construct obstacles that replicate these conditions in the simulation setup. For polygonal shading, a 20m $\times$ 20m rectangle with

$$V = \{(-10, -10), (10, -10), (10, 10), (-10, 10)\}$$

and normal $\vec{v_n} = (0, 0, 1)$ is placed at different positions:

1. $p_0 = (25, 0, 10)$: centered above the heliostat, fully shading it.

2. $p_0 = (35, 0, 10)$: shifted 10m east, leading to partial shading, where approximately half of the facet is covered.

3. $p_0 = (-25, 0, 10)$: shifted west, yielding no shading.

Similar configurations are tested with a cylindrical obstacle of radius $r = 10$m and height $h = 5$m, oriented along $\vec{v_n} = (0, 0, 1)$:

1. $p_0 = (25, 0, 10)$: cylinder centered directly above the heliostat, fully shading its facet.

2. $p_0 = (35, 0, 10)$: cylinder shifted 10m east, partially covering the heliostat facet and thus reproducing partial shading.

3. $p_0 = (-10, 0, 10)$: cylinder shifted west, yielding no shading.

These polygonal and cylindrical obstacle placements are illustrated in Figure 5.2.

For blocking validation we distinguish three cases, similar to shading:

1. Full blocking: all rays reflected toward the receiver are intercepted, expecting $A_{\text{total}} = A_{\text{obstacle blocked}}$.

2. Partial blocking: only a subset of reflected rays is intercepted, expecting $A_{\text{total}} > A_{\text{obstacle blocked}} > 0$.

3. No blocking: no reflected ray is intercepted, expecting
   $A_{\text{obstacle blocked}} = 0$.

We replicate these conditions with a vertical polygonal obstacle (plane normal along the global x-axis) using the vertex set

$$V = \{(-2.5, 0),\ (2.5, 0),\ (2.5, 120),\ (-2.5, 120)\}, \quad \vec{v}_n = (1, 0, 0),$$

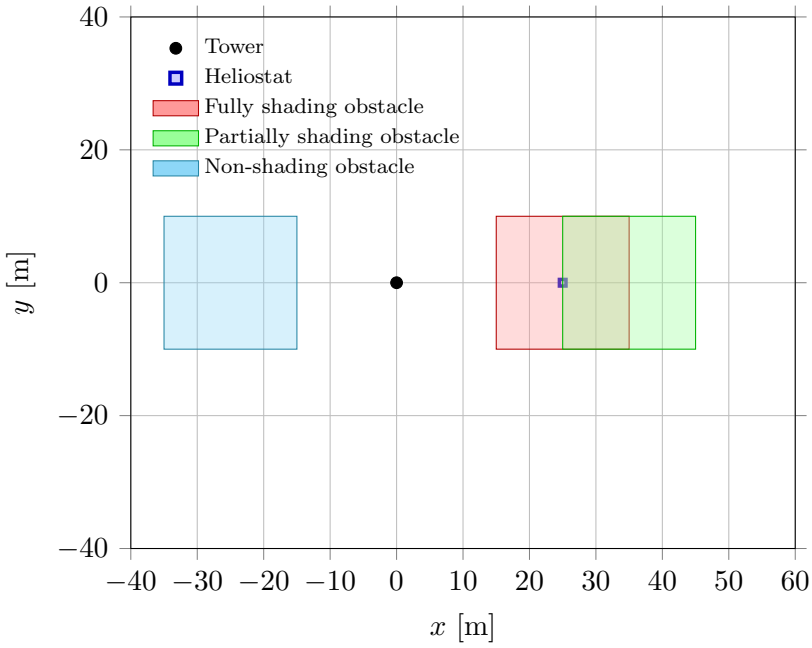
and different placements p_0:

1. $p_0 = (12.5, 0, 0)$: the obstacle lies on the reflected path between heliostat and receiver, all reflected rays are intercepted

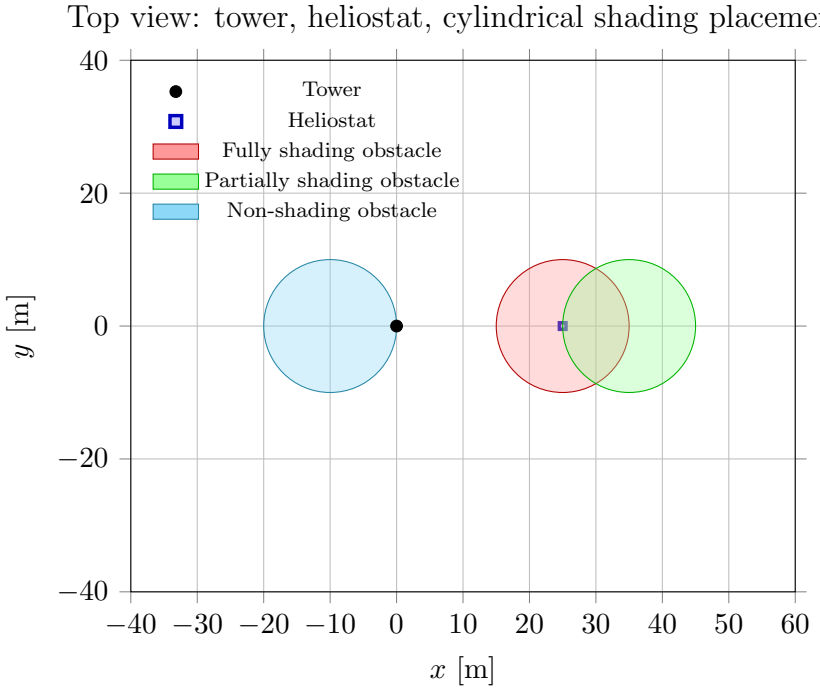
2. $p_0 = (15, 2.5, 0)$: the obstacle is shifted relative to the heliostat-receiver direction, intercepting only part of the reflected rays. Approximately half of the heliostat reflected area is blocked.

3. $p_0 = (-12.5, 0, 0)$: the obstacle is placed on the opposite side of the tower. Reflected rays are still hitting it, but they are expected to hit the receiver first, so there should be no blocking.

The corresponding polygonal and cylindrical blocking configurations are shown in Figure 5.3. On each of the 3 specified configured ray tracing methods we run 12 simulations: one for each scene with individually configured and positioned polygonal or cylindrical obstacle.

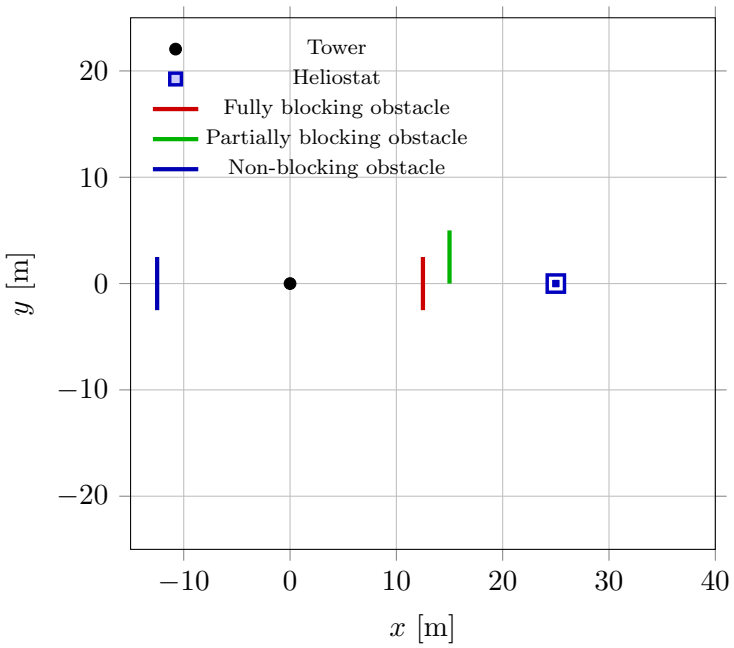


(a) Placements of polygonal obstacles used in shading validations. Identical rectangles were placed at different positions, 10 m above the heliostat.

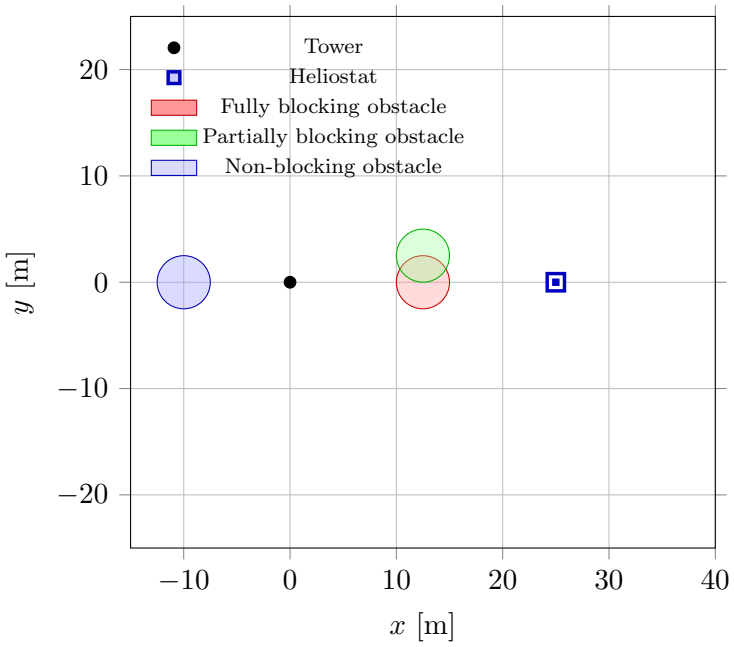


(b) Placements of cylindrical obstacles used in shading check tests. Identical cylinders were placed at different positions, 10 m above the heliostat.

Figure 5.2: Maps of the simulated scenes (top view) in shading validations. Every rectangle and cylinder was tested individually, not simultaneously.



(a) Placements of polygonal obstacles used in blocking validations. Identical $5 \times 120\,\text{m}$ rectangles were placed at different positions. Each line on the map represents a top view of an obstacle.



(b) Placements of cylindrical obstacles used in blocking validations. Identical rectangles were placed at different positions. Each circle on the map represents a top view of a 120m tall cylinder with radius of 2.5m.

Figure 5.3: Maps of the simulated scene (top view) in blocking validations. Every rectangle and cylinder was tested individually, not simultaneously.

5.1.2 Results

Tables 1, 2 and 3 show the resulting area values for simulation with different obstacle types and placements, for every ray tracing method. Across all placements, the outcomes match the goals: there are no shading or blocking effects when obstacle is positioned further away from heliostat, including special cases for blocking, when the blocking polygon or cylinder is placed behind the receiver tower. Obstacles also shade or block either the entire mirror or the half of the total mirror area, depending on a placement configuration used in a test case. Outputted total shaded area values $A_{\text{obstacle shaded}}$ are identical for every ray tracer due to shading check tracing the same Sun ray in every of the cases. While exactly the half of the mirror is reported to be shaded for polygonal obstacle, which corresponds to its placement, the value is slightly below $0.5\,\text{m}^2$ due to the circular shape of the cylinder over the heliostat. Furthermore, tracing ideal reflected representative rays, Convolution ray tracer calculates the exact blocked area to be $0.5\,\text{m}^2$, exactly the half of the area of heliostat's reflective surface. In contrast, Monte Carlo exhibits the sampling noise with tested reflective rays being perturbed. When using 100 rays / m^2 there are visible deviations from the expected blocked area. Increasing the number of rays to 10000 rays / m^2 brings the result into near agreement with the convolution baseline, as expected of Monte Carlo: increasing the number of rays improves the accuracy.

The mean efficiencies η_{os}, η_{ob} are also calculated and reported as the result of each simulation. Figures 5.4 and 5.5 show the breakdown of the output power for partially blocking and shading cases with polygonal obstacle using Convolution ray tracer: with obstacle added to the scene the remaining power drops by one half in both cases. The waterfall plot depiction clearly showcases the reduction caused by obstacle-induced shading or blocking.

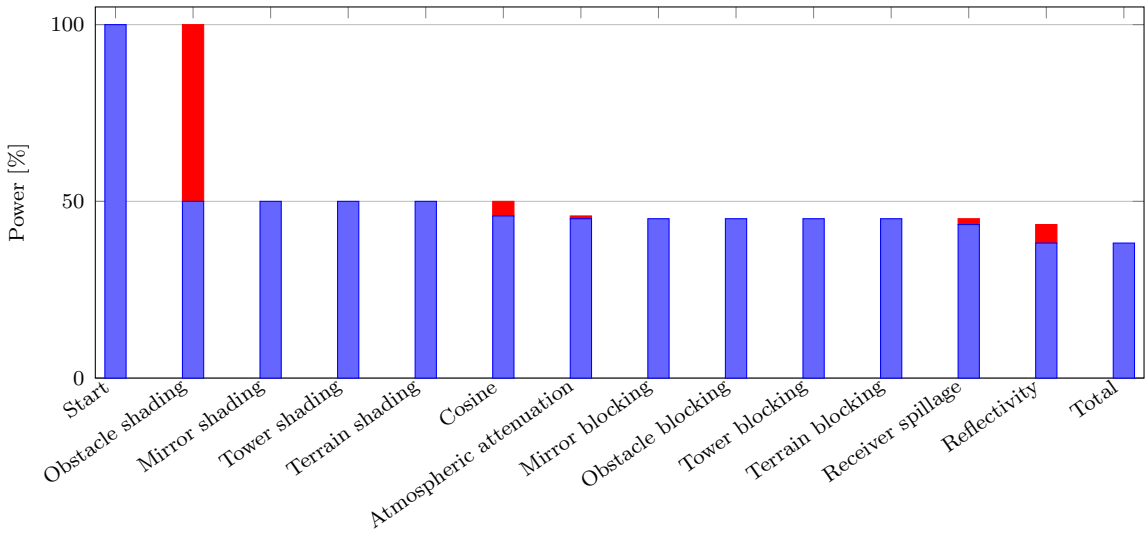


Figure 5.4: Optical losses from 100% incident power, outputted after running a simulation using CPU convolution ray tracer with a partially shading polygonal obstacle added to the scene. Blue bars represent remaining power, while red segments are the per-step energy losses. The $\approx 50\%$ loss from obstacle shading mean efficiency corresponds to polygon shading half of heliostat's reflective surface.

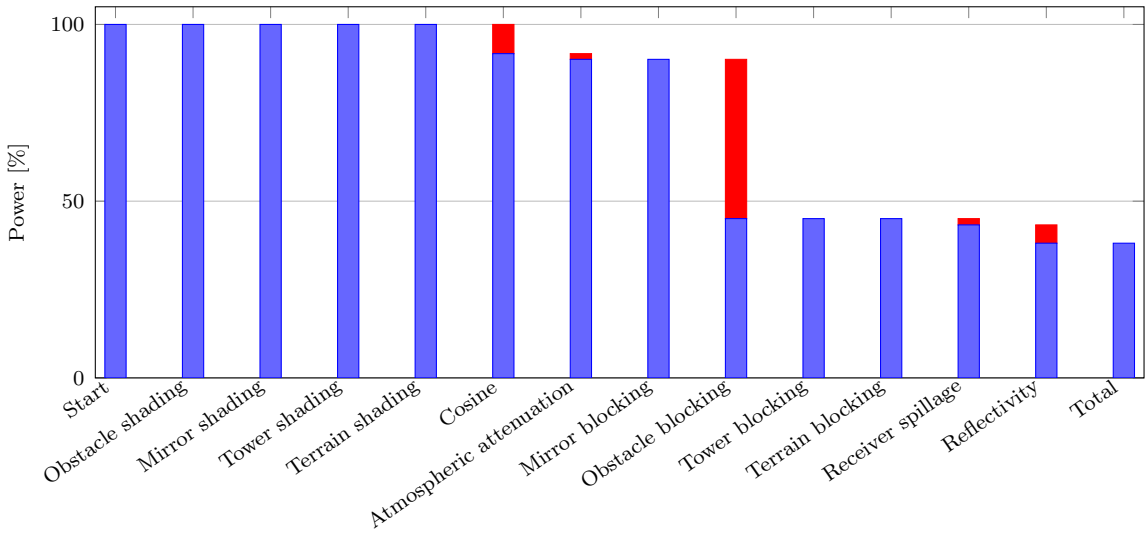


Figure 5.5: Optical losses from 100% incident power, outputted after running a simulation using CPU convolution ray tracer with a partially blocking polygonal obstacle added to the scene. Blue bars represent remaining power, while red segments are the per-step energy losses. The $\approx 50\%$ loss from obstacle blocking mean efficiency corresponds to polygon blocking half of heliostat's reflective surface.

Obstacle	$A_{\text{obstacle shaded}}$	$A_{\text{obstacle blocked}}$	A_{total}
Polygon obstacles			
Fully blocking polygon	–	$1\,\text{m}^2$	$1\,\text{m}^2$
Partially blocking polygon	–	$0.5\,\text{m}^2$	$1\,\text{m}^2$
Non-blocking polygon	–	0	$1\,\text{m}^2$
Fully shading polygon	$1\,\text{m}^2$	–	$1\,\text{m}^2$
Partially shading polygon	$0.5\,\text{m}^2$	–	$1\,\text{m}^2$
Non-shading polygon	0	–	$1\,\text{m}^2$
Cylinder obstacles			
Fully blocking cylinder	–	$1\,\text{m}^2$	$1\,\text{m}^2$
Partially blocking cylinder	–	$0.5\,\text{m}^2$	$1\,\text{m}^2$
Non-blocking cylinder	–	0	$1\,\text{m}^2$
Fully shading cylinder	$1\,\text{m}^2$	–	$1\,\text{m}^2$
Partially shading cylinder	$0.498\,\text{m}^2$	–	$1\,\text{m}^2$
Non-shading cylinder	0	–	$1\,\text{m}^2$

Table 1: Simulation results with different obstacle placements on Convolution ray tracer

Obstacle	$A_{\text{obstacle shaded}}$	$A_{\text{obstacle blocked}}$	A_{total}
Polygon obstacles			
Fully blocking polygon	–	$1\,\text{m}^2$	$1\,\text{m}^2$
Partially blocking polygon	–	$0.492\,\text{m}^2$	$1\,\text{m}^2$
Non-blocking polygon	–	–	$1\,\text{m}^2$
Fully shading polygon	$1\,\text{m}^2$	–	$1\,\text{m}^2$
Partially shading polygon	$0.5\,\text{m}^2$	–	$1\,\text{m}^2$
Non-shading polygon	0	–	$1\,\text{m}^2$
Cylinder obstacles			
Fully blocking cylinder	–	$1\,\text{m}^2$	$1\,\text{m}^2$
Partially blocking cylinder	–	$0.531\,\text{m}^2$	$1\,\text{m}^2$
Non-blocking cylinder	–	0	$1\,\text{m}^2$
Fully shading cylinder	$1\,\text{m}^2$	–	$1\,\text{m}^2$
Partially shading cylinder	$0.498\,\text{m}^2$	–	$1\,\text{m}^2$
Non-shading cylinder	0	–	$1\,\text{m}^2$

Table 2: Simulation results with different obstacle placements on Monte Carlo ray tracer, with 100 rays per m^2

Obstacle	$A_{\text{obstacle shaded}}$	$A_{\text{obstacle blocked}}$	A_{total}
Polygon obstacles			
Fully blocking polygon	–	$1\,\text{m}^2$	$1\,\text{m}^2$
Partially blocking polygon	–	$0.503\,\text{m}^2$	$1\,\text{m}^2$
Non-blocking polygon	–	–	$1\,\text{m}^2$
Fully shading polygon	$1\,\text{m}^2$	–	$1\,\text{m}^2$
Partially shading polygon	$0.5\,\text{m}^2$	–	$1\,\text{m}^2$
Non-shading polygon	0	–	$1\,\text{m}^2$
Cylinder obstacles			
Fully blocking cylinder	–	$1\,\text{m}^2$	$1\,\text{m}^2$
Partially blocking cylinder	–	$0.498\,\text{m}^2$	$1\,\text{m}^2$
Non-blocking cylinder	–	0	$1\,\text{m}^2$
Fully shading cylinder	$1\,\text{m}^2$	–	$1\,\text{m}^2$
Partially shading cylinder	$0.498\,\text{m}^2$	–	$1\,\text{m}^2$
Non-shading cylinder	0	–	$1\,\text{m}^2$

Table 3: Simulation results with different obstacle placements on Monte Carlo ray tracer, with 10000 rays per m^2

5.2 Obstacle Runtime

The runtime of any simulation is expected to increase when using obstacles as they only introduce new calculations during blocking and shading checks. This directly comes down to the efficiency of the corresponding isShading, isBlocking operations as well as the amounts of traced rays and the number of obstacles in $\mathbb{O}$. In order to reduce the amount of per-ray calculations when tracing for intersections with obstacles, we have already introduced pre-filtering checks within the algorithms: minimal bounding sphere in case of polygonal obstacle and early checks for parallelism and infinite cylinder is case of cylindrical obstacle. However, further impact of obstacle model on the simulation runtime depends on the specific configuration. Ray tracing methods differ in the amount of casted rays and obstacles themselves may be required in different quantities. The heliostat fields vary in their sizes across different solar power plants. An obstacle can be positioned close to the receiver, drawing a lot of rays into further post-filter calculations or it could be placed further away, affecting only a small portion of the heliostats. All of these factors combined can lead to different runtimes of the simulation. In the following, we are going to take a look at the upper bound of obstacles' impact on the runtime, by running simulations with obstacles specifically designed to trigger the most expensive, worst-case isShading, isBlocking calculations. Within the field of PS10 power plant [11], with data of its simulation scene provided within the dataset of the SunFlower simulation tool, we are going to place increasing amounts of obstacles, which will fail the pre-filter checks on all of the 624 heliostats, but also not block the ray at the end. Such configurations represent the worst-case for both isShading and isBlocking check routines, as we perform each isBlocking tracing without pre-filtering rejects, performing all the calculations up to the costly point-in-polygon check or up to solving the quadratic equation and performing cap checks. Since every isBlocking is going to be rejected at the end, ray will not be marked as blocked during the blocking check of the ray tracer and every consecuitve obstacle in $\mathbb{O}$ will be checked further up to the end, when the unblocked ray is further evaluated for its impact on the receiver. The aim of these tests is to evaluate the impact of the obstacle model on the simulation performance on both CPU and GPU in a worst-case scenario, and to assess the efficiency of the implemented intersection checks for both polygonal and cylindrical obstacles. In the following we introduce the obstacles which were used in runtime measurements.

As we have previously established, we have not introduced any collision checks for obstacles. Subsequently, any two $O_x, O_y \in \mathbb{O}$ can be parametrized to intersect with each other. Moreover, there is no check for two identical obstacles $O_x, O_y \in \mathbb{O}$ with $O_x = O_y, x \neq y$. Not only in this case a duplicated obstacle does not alter either the simulation environment, or the results, but if O_x fails the blocking check, it will be performed again for O_y for $x < y$, thus only adding the redundant calculations in the worst case. However, this effect allows us to easily define previously mentioned chains of worst-case computations. We define three obstacle cases: a cylinder obstacle, a polygonal obstacle with 4 vertices and an additional polygonal obstacle with 16 vertices,

so we can not only evaluate performance of polygonal obstacle against cylindrical, but also compare performance to the polygonal obstacle with more computationally expensive point-in-polygon test. For each of these obstacles we proceed as follows:

1. We adjust obstacle's dimensions, such as radius and height of cylindrical obstacle and positions of vertices on the local frame of a polygonal obstacle, and positions to specifically get every reflected ray to pass obstacle's pre-filter check and then fail the final check. As PS10's tower is 120m in height and 18m [11] in width we can place a cylindrical obstacle at tower's position and set radius to 20m, but also set height to 1m and with parallel to tower axis $(0, 0, 1)^T$ so that every reflected ray that is headed in receiver's direction passes the infinite cylinder check, but consequently not getting blocked after computing the caps check. In case of polygonal obstacles, we place both of them at $p_0 = (0, 20, 0)$, between the closest heliostat and the tower and set $v_n = (0, 1, 0)$ to make them parallel to the YZ plane. Vertices V of the rectangular polygon are then set to $V = \{(-350, 0), (350, 0), (350, 0.5), (-350, 0.5)\}$, as it makes it wider then the entire heliostat field and results in a very large bounding sphere radius, making each reflected ray pass the bounding sphere check, but, due to the barrier being only 50cm tall and located down on the ground, consequently fail the point-in polygon check. Same strategy is used for a polygon obstacle with 16 edges: with same p_0, v_n we set

$$\begin{aligned} V = \{&(-350, 0), (350, 0), (350, 0.6), (300, 0.5), \\ &(250, 0.6), (200, 0.5), (150, 0.6), (100, 0.5), \\ &(50, 0.6), (-50, 0.5), (-100, 0.6), (-150, 0.5), \\ &(-200, 0.6), (-250, 0.5), (-300, 0.6), (-350, 0.5)\} \end{aligned}$$

   which is a similar structure, but with additional vertices.

2. For each of the above mentioned obstacles we add a single instance of it to $\mathbb{O}$ and run the simulation using Monte Carlo ray tracer with number of rays per m^2 set to 100.

3. We repeat the same procedure, but this time with two identical instances of each obstacle added to $\mathbb{O}$.

4. We proceed to run simulations measuring raytracer's runtime while doubling the amount of identical obstacles each time, with last simulation containing 32 identical obstacles in $\mathbb{O}$.

As a result, we obtain the mean runtimes for obstacle set sizes of $1, 2, 4, 8, 16$, and 32 identical obstacles. This provides a clear view of how the simulation runtime scales under worst-case conditions, where each ray triggers the full sequence of pre-filter and intersection checks. The results allow us to quantify the computational overhead introduced by additional obstacles and to compare the relative performance of cylindrical

versus polygonal obstacles, as well as the effect of polygon complexity (4 vertices vs. 16 vertices) on runtime. These procedures are run on both CPU and GPU implementations of the Monte Carlo raytracer in order to assess whether the impact of additional obstacle checks is consistent across architectures. This comparison also highlights the relative performance scaling and efficiency of the obstacle model in different computational environments.

5.2.1 Results

Figure 5.6 shows the measured runtimes for $0, 1, 2, 4, 8, 16, 32$ obstacles on both CPU and GPU Monte Carlo ray tracers. As expected the runtimes increase approximately linearly on both CPU and GPU, as each additional obstacle introduces an equivalent number of additional intersection checks. Cylindrical obstacles consistently yield the lowest overhead, showing a marginal increase in computation time even at higher counts. In contrast, polygonal obstacles prove to be more demanding, especially with the larger number of edges. Therefore, to achieve the best performance, cylinders must be used to simulate obstructions whenever possible. If, however, a planar barrier must be represented, polygonal obstacles should be used, preferably with a lower number of edges: polygons with four edges results in lower runtimes than those with sixteen, as expected from the implemented point-in-polygon algorithm. On the CPU, the simulation time increased from approximately 1476 ms without obstacles to about 5464 ms for 32 4-edged polygons, whereas the same configuration with 16-edged polygons required around 6493 ms—corresponding to roughly a 19% increase. On the GPU, the simulation time increased from approximately 439 ms without obstacles to about 2501 ms for 32 4-edged polygons, whereas the same configuration with 16-edged polygons required around 4424 ms, corresponding to a 77% increase. While GPU simulations are still noticeably faster than the equivalent CPU setups, the significantly larger runtime difference between 4- and 16-edged polygons on the GPU when compared to CPU can be attributed to GPU-specific factors: in contrast to the CPU implementation, on the GPU large numbers of threads are executed in parallel, which can become less efficient when the computational workload per thread varies. As a result, more complex polygons lead to reduced parallel utilization and higher overall runtime, even though the total number of operations remains the same. A similar effect was described in the performance analysis of the GPU ray tracer [2].

On average the runtime increases by 214% on the CPU and 422% on the GPU when going from 1 to 32 obstacles. Apart from the scaling on GPU being less linear, notably for the polygonal obstacles, we must note that while these values represent the worst case for defined obstacles and more realistic simulations are therefore expected to have lower runtimes, similar results may be easily achieved when trying to construct complex objects by placing intersected obstacles close to each other. While this approach allows us to simulate more dedicated obstructions by, for example, building a 'box' out of six rectangular polygonal obstacles, the resulting cluster of obstacles might also face the repeated pre-filter fails and further failed intersection tests for every rectangle.

Although the described situation is not as extreme as having several identical obstacle occupy exactly the same position, it can nevertheless lead to a substantial increase in runtime, especially if obstacles are large and placed close to the receiver.

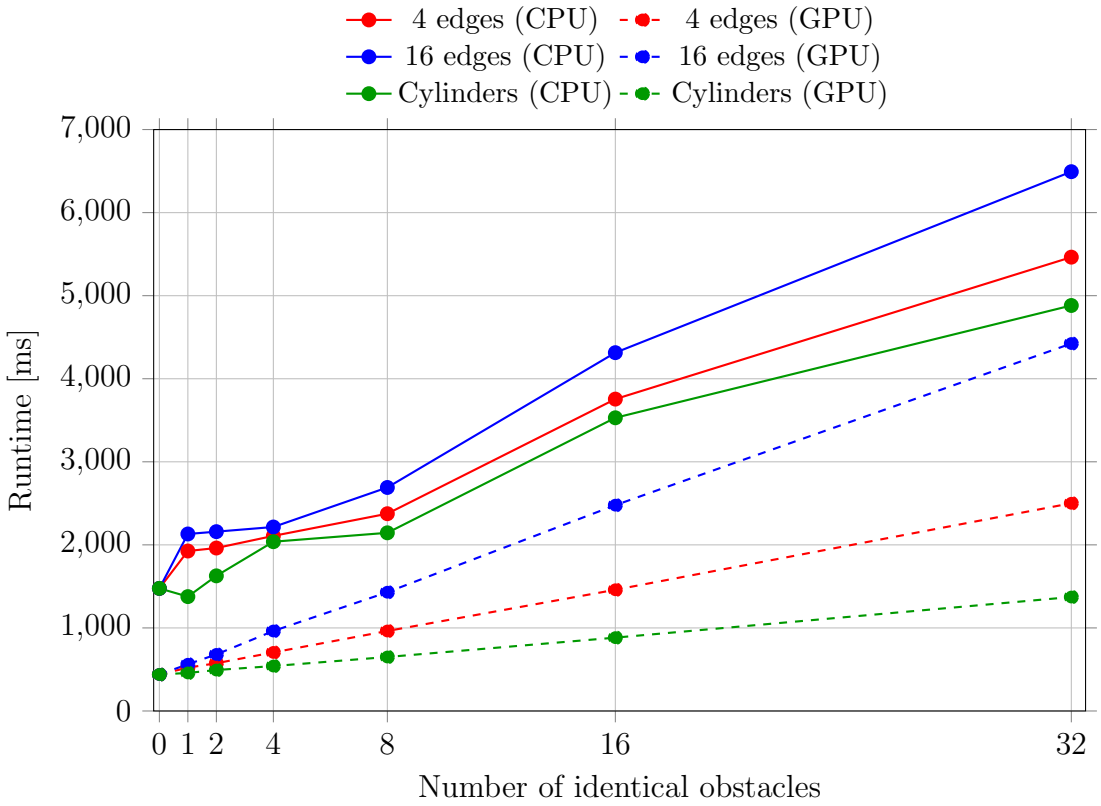


Figure 5.6: Runtimes of the worst-case obstacle simulations on the PS10 scene using Monte Carlo on CPU and GPU with 100 rays per m^2.

6 Summary

We have extended SunFlower simulation tool with an obstacle model that enables the simulation of on-site obstructions in CRS. Existing shading and blocking routines, which previously only covered mirrors being obstructed by either the tower structure or other heliostats, were extended to include tests of blocking and shading by the obstacles. Within the new model we have introduced two types of obstacles: finite cylinders and flat polygons. Parameters such as their positions, sizes and rotations are highly configurable. Both cylindrical and polygonal obstacles require only a minimal amount of memory and their respective tracing routines, which are used in blocking and shading checks during the simulation, use early-out pre-filtering to reduce the amount of per-ray calculations. Environmental obstructions can be represented within the simulation as a cylinder or a polygon and energy losses caused by it potentially shading and blocking heliostats can now be assessed. Furthermore, we evaluated the

impact of the newly introduced obstacles on the simulation runtime. It was concluded, that in order to achieve better performance, cylindrical obstacles should be prioritized. In case an obstruction has to be represented by a polygonal obstacle, polygons with lower amounts of vertices should be preferred. Placing obstacles of different forms, clustered together to represent a unique shape is possible. However, such practice is discouraged due to significant impact on simulation's runtime.

Additionally, several outstanding issues within the SunFlower have been resolved. Integration of a new facet schema has been finished, re-enabling the 3D-Printing and Tonatiuh export services. 3D-Printing service was then expanded by adding meshes that represent cylindrical and polygonal obstacles, providing the means to visualize the simulated solar power plant scene with added obstructions. Finally, we have introduced a new feature: automatic generation of facets. Rather than specifying position, shape, dimensions and further parameters of every facet on the heliostat, a compact configuration can be provided instead. In it, facet parameters are specified only once and the individual configurations are then generated automatically, given their quantity and eventual gap.

6.1 Outlook

While many on-site obstructions can be approximated by implemented cylindrical or polygonal obstacles, the obstacle model was designed to be extendable: ray tracing methods work with abstract list of all obstacles $\mathbb{O}$. Therefore, in future works the obstacle model can be expanded by introducing new shapes (e. g. cones, boxes, triangular prisms), given the corresponding intersection procedures isShading and isBlocking are implemented for CPU or GPU. Furthermore, with Tonatiuh export tool restored, it may be expanded with features that would enable import or export of obstacle shapes, enabling the cross-validation tests.

While both polygonal and cylindrical obstacles have a straightforward parameter schema, the updated CRS scene with added obstacles is currently viewable only by exporting an STL via 3D-Printing tool and viewing it in an external viewer. Future works could either add a native tool for visualization of modeled obstacles or implement support for import from common 3D modeling tools.

Finally, results of performance evaluation suggest room to optimize tracing of polygonal obstacles on GPU, as current implementation, which follows the CPU implementation, showed worse scaling with larger vertex count.

References

[1] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-Time Rendering, 4th*. A K Peters/CRC Press, 2018. ISBN 978-1138627000.

[2] Lukas Aldenhoff. Raytracer for central receiver systems using gpu. Master's thesis, RWTH Aachen University, 2021.

[3] Manuel J. Blanco, Juana M. Amieva, and Azael Mancillas. The tonatiuh software development project: An open source approach to the simulation of solar concentrating systems. In *ASME International Mechanical Engineering Congress and Exposition Proceedings Series*, 2005. doi: 10.1115/IMECE2005-81859.

[4] Paul Bourke. Calculating the area and centroid of a polygon. `http://paulbourke.net/geometry/polygonmesh/`, 1988.

[5] João P. Cardoso, Amaia Mutuberria, Costas Marakkos, Peter Schoettl, Tiago Osório, and Iñigo Les. New functionalities for the tonatiuh ray-tracing software. In *SolarPACES 2017: International Conference on Concentrating Solar Power and Chemical Energy Systems*, 2017. doi: 10.1063/1.5067212.

[6] Christer Ericson. *Real-Time Collision Detection*. Morgan Kaufmann, 2005. ISBN 978-1558607323.

[7] Florian Hövelmann, Lukas Aldenhoff, and Pascal Richter. Analytic ray tracer on gpu for central receiver systems. *Journal of Renewable Energy*, 2023. doi: 10.1016/j.solener.2023.112149.

[8] Florian Hövelmann, Lukas Aldenhoff, and Pascal Richter. Integrated convolution ray tracer on GPU for central receiver systems. *Journal of Renewable Energy*, 2023. doi: 10.2139/ssrn.4227667.

[9] Florian Hövelmann. Accelerated raytracer for solar tower power plants. Bachelor's thesis, RWTH Aachen University, 2019.

[10] International Energy Agency. Global energy review 2025. Technical report, International Energy Agency (IEA), Paris, 2025.

[11] Rafael Osuna, R Olavarría, Rafael Morillo, Marcelino Sánchez, Felipe Cantero, Valerio Fernández-Quero, Pedro Robles, T López, Antonio Esteban, Francisco Céron, et al. Ps10, construction of a 11mw solar thermal tower plant in seville, spain. In *Conference paper: SolarPACES*, 2006.

[12] REN21. Renewables global status report 2025: Global overview. Technical report, REN21 Secretariat, Paris, 2025. URL `https://www.ren21.net/gsr-2025/`.

[13] Pascal Richter. *Simulation and Optimization of Solar Thermal Power Plants*. Dissertation, RWTH Aachen University, 2017.

[14] David Elias Walter Schütte. Modeling and simulation of tilt-and-roll heliostats in central receiver systems. Bachelor's thesis, RWTH Aachen University, 2024.

[15] Tim Wendelin. Soltrace: A new optical modeling tool for concentrating solar optics. In *ASME 2003 International Solar Energy Conference*, 2003. doi: 10.1115/ISEC2003-44090.

[16] Haitao Zhang, Yiling Sun, Lu Wu, Xiaokun Zhang, and Yong Xiang. Tracking mechanism and cosine effect study of module-heliostat solar collector. In *4th International Conference on Machinery, Materials and Information Technology Applications (ICMMITA 2016)*, pages 469–474. Atlantis Press, 2017. doi: 10.2991/icmmita-16.2016.87.