Ray tracing UV light sterilization of hospital rooms

Oscar Fickel (6520553)

September 2022

0 Preface

This report is the result of a GMT Small Project performed for the hospital ZorgSaam, under the supervision of Frank van der Stappen.

Contents

1 Introduction

Researchers working with ZorgSaam hospital are looking into the feasibility of disinfecting hospital rooms with the help of UV radiation, and are finding promising results. This is usually done with the help of a simple robot that can navigate to predetermined positions in a room, which is outfitted with a vertical UV lamp.

In light of these developments, the aim of this project is to allow the hospital staff to more accurately and easily determine where and how long this robot needs to hold this UV light to have a minimal exposure of each visible surface to guarantee decontamination. This saves time from having to place sensors in the room to test for sufficient disinfection, and provides a more general intuitive overview. There already exists a model for simulating this, but the calculations are relatively rudimentary, and do not sufficiently take into account vertical surfaces and shadows. The idea is to solve this problem by ray tracing the UV light using a 3D model of the hospital room.

1.1 Program requirements

Solving the optimization problem of where and how long the UV light should be held is outside the scope of this project. Instead the focus will be on allowing users to do this manually by computing the amount of UV light spread out over the surfaces in the room when placing the robot at variable locations for variable amounts of time. However it should also be noted that the application is intended for practical use, not scientific validation. This subtle distinction highlights that it is more important for the application to be easy to use and quickly produce a valid general overview, than it is to produce a highly detailed simulation. Having a correct overview of the UV radiation is thus more important than for example knowing at exactly which point part of a wall is too far from the light to receive a specific dosage. On the other hand, while better performance (in terms of speed) will increase the usability of the program, it is not necessary for the ray tracing part of it to be real time, especially if the radiation is precomputed rather than continuously updated. The user should of course be able to navigate and inspect the scene in (at least semi-) real time, but the UV radiation does not need to be updated at the same frequency. Note also that the application is not intended for regular use, but for one time calculation and configuration of optimal light placement per room.

Apart from the lamp trajectory, the scene is entirely static. The effect of reflections or indirect lighting is disregarded. In the worst case the room might contain a big mirror or other reflective surface that causes the ray tracing program to suggest a longer waiting time for the UV light than necessary. Since reflections and indirect lighting can only increase the delivered UV doses, it is not possible for the ray tracer to suggest a lighting setup that is insufficient in cleaning the room due to not taking this into account. Therefore it is safe to ignore these factors. Moreover, inaccuracies in the mesh, like incorrectly angled surfaces, are amplified in reflections, as both the computed light intensity as the direction of the reflection will be erroneous, more so with each subsequent reflection. It would furthermore be challenging to determine the correct reflectance of different surfaces in the room and incorporating these into the model. As shown in studies from as far back as 1915 [7], reflectance (of metals) is inconsistent between different wavelength, and generally lower for lower wavelengths, as is the case for UV radiation. This also shows that while indirect lighting can make a room much brighter in the visual spectrum, this is not necessarily the case for UV light as well.

Ideally the ray tracer also would not just assume a static light in different positions, but take into account the movement of the robot as well. Due to time constraints however, this is left to the user by making them interpolate this movement themselves into static points and durations. It would also be possible to model the properties of a specific UV light robot in great detail, taking into account its exact shape and any patterns in the directional distribution of light it sends out. However, for the sake of generalizability and time constraints, the light source will instead be modeled as a simple vertical line/rod. This ensures that the program will be generic enough to be used with different types of UV robots, and thus not just reflect the model of a specific robot.

Lastly, ZorgSaam requested that in addition to the cumulative UV dosage map, a map of the maximum irradiation per surface would be generated, as this could potentially also affect the disinfection rate, and may be relevant in the future when determining optimal routes.

1.2 Deliverable specifications

In summary, the ray tracing program to be delivered must at least:

- Be able to compute and display a mapping of the (cumulative) dose of UV radiation delivered to the surface of an input mesh.
- Allow users to specify the power, length, height above the floor, position, and duration of the UV light.
- Display the light positions on the screen in a simple manner.

Ideally the program should also:

- Allow saving and loading the UV light positions and specifications.

- Compute maximum UV power map.
- Compute the dosage map within a reasonable amount of time (under 5 minutes).
- Implement a more intuitive way of editing the light positions than directly editing coordinates

The program will not:

- Model movement of the UV light rather than discrete static positions.
- Determine optimal light positions and durations automatically.

2 Approach

2.1 3D model

The first major obstacle in this project is the acquisition of sufficiently accurate 3D models of the hospital rooms for which the optimal UV robot trajectory must be calculated. If the 3D model does not accurately represent the room, then the results from the ray tracer might not be generalizable to the real room, and an optimal route for the virtual room might be far from the optimal route for the real room. However, performing a detailed 3D scan of a room is often expensive, so a more viable solution needs to be investigated.

Alternatively it would also be possible to recreate the hospital rooms in a 3D editing program. Since there is only a limited number of rooms for which this must be done, and most rooms for example share the same model of bed, it might be viable to create a few assets (some of which could even be obtained online) that can be placed and moved around in a model of an empty room. This would either require users to be able to use their own 3D editing software (e.g. Blender) to create and edit the models, or the ray tracer program would need to have a user friendly way of creating and editing hospital room 3D models by placing and moving the premade assets.

Ultimately, a 3D scan made with a phone using LiDAR technology proved to be sufficient, an example of which was provided by Loop Robots. This kind of scan is far from perfect; it can contain subtle inaccuracies like missing parts of complex shapes, or inserting parts that are not there. However, the method is very effective for obtaining the general shape of a room and the objects in it. For ray tracing the amount of UV radiation parts of the room receive, it may very well be accurate enough, especially considering the low acquisition costs. A disadvantage of this method is that it means that all objects in the room are fixed and baked into the mesh, which restricts users from experimenting with moving furniture around to optimise light placement. The model provided by Loop Robots contains 46,252 triangles, which is less than for example the Stanford bunny. It was created with Polycam, which allows for using phones to scan 3D meshes, either with or without LiDAR. LiDAR technology is currently only supported by the iPad Pro and by the Pro and Pro Max versions of the iPhone 12, 13, and 14. The free version of Polycam allows exporting to GLTF, which can be interpreted with C++ libraries like TinyGLTF. Therefore, to ensure that users can supply the program with 3D models as effortlessly as possible, the loading of GLTF files is supported. While the acquisition costs of such a LiDAR model are significantly lower than the alternatives, in practice it proved to be rather tricky for someone inexperienced with the process to obtain a model without glaring artifacts.

All in all, considering the still relatively low acquisition costs of the LiDAR scan compared to creating a 3D model from scratch, this is the method we chose to support. Perhaps some combination of the two methods to enable moving furniture would have been even better, but due to time constraints we decided this was beyond the scope of the project. If necessary, users could always decide to create a new scan or even edit a mesh themselves to use as input, as long as it is in the correct GLTF format.

2.2 UV radiation

Since we are not building a regular ray tracer, but rather one that models the effect of UV radiation, there some additional aspects to consider. For the model to be true to real life, it is essential that the properties of the UV light are correctly implemented. These properties include the intensity of the light and how this affects the dose of UV radiation a surface receives. Apart from the regular factors of light attenuation and surface hit angle, the UV dose a surface receives is also affected by the attenuation factor of the surface and the extinction coefficient of the air. It is possible that light extinction in the air is not negligible if the air in the hospital for example contains a significant amount of ozone, which can be used in some air disinfectants. As shown by Lai et al [9], this can have a significant impact, since the amount of time required for 274 and 222-nm wavelength UV-C sterilization increases to 2.2 and 1.5 times when assuming the worst case scenario of 20-ppm ozone in the air. However, for the sake of simplicity, and as suggested by our contacts at ZorgSaam, this factor is omitted in the ray tracer.

Considering that the wavelength of UV radiation lies outside the part of the visual spectrum covered by RGB values, a natural technique to consider might have been spectral rendering [4]. In spectral rendering, light transport is modeled with real wavelengths rather than RGB values. However, we only need to model the transmission of UV light to the surfaces, and do not reflect it any further to the screen. Therefore no computations with objects' emission spectrum or spectral reflectance curves will need to be done, and the wavelength only becomes relevant for the minimum UV dose required for disinfection, which should be supplied by the user. The UV radiation dose D for a given point at a distance r from the light source can be computed as simply the irradiance $\phi(r)$ at that point times the time duration.

Note also that ZorgSaam confirmed that pauses during which a surface does not receive UV radiation can be assumed to not affect the cumulative dose it receives, under the assumption that these pauses are not too long.

To ensure that UV dose that the ray tracer ends up calculating is accurate to the real world, an option is implemented to compute the irradiance for a specific measurement and scale the lamp power based on how the computed irradiance compares to the given measured irradiance. Since the ray tracer will likely not simply deviate linearly from the real measurements, this is only a limited form of calibration given the limited time frame. More extensive calibration and validation could be done in a future project.

2.3 General ray tracing method

For displaying the dosage map, the computation can either be done on the fly for each pixel on the screen, as is done with most ray tracers, or precomputed for certain points on the mesh surface. While the application does not require game-level real-time performance, users do need to be able to effortlessly inspect the scene mesh from different angles. It would be possible to implement an on the fly computation ray tracer that fulfills this requirement, but it would cost more time and effort to optimise than precomputing a dosage map. Therefore, to be certain of being able to generate a real-time inspectable heatmap at the end of the project, we chose to precompute a dosage map. The dosage map that needs to be displayed is quite suitable for precomputation, as the color values of the mesh surface that need to be displayed on the screen are completely independent of the camera position, since we do not handle reflections. Therefore a technique like photon mapping [8], which also does not take the camera position into account, seems like a good fit. On the other hand, backwards ray tracing techniques are also a viable method for precomputation.

2.3.1 Backward ray tracing

Backward tracing means tracing rays from points on objects in the scene towards the light source, rather than the other way around, as is the case with forward tracing. Lai et al [9] implement this by creating a uniform triangle mesh, so that there will be an uniform distribution of vertices for which the UV dose can be calculated. For each of these points the UV dosage can then be calculated by tracing rays towards the light source. Lai et al [9] model the lamp as a straight line made up of of individual point lights that shine more light in a certain direction, reflecting the build of the specific led strip they were modeling. Our ray tracer would have to simply use a uniform light distribution, to ensure being able to generalize over different types of UV lamps, and implement it within the limited time constraints. An alternative to a collection of point lights would be to use an area light. Area lights can be handled by using Monte Carlo integration, in this case over a vertical line, meaning that rays would be cast to random points on this line until the noise is reduced to an acceptable level. This would cost more computation time but would also more accurately simulate the soft shadows.

An alternative approach to tracing rays from each vertex position would be generating a number of detector points, for example using Poisson disc sampling [3] [5], which would allow for a more detailed dosage map.

2.3.2 Forward ray tracing

The most common implementation of precomputing light distribution with forward tracing is photon mapping. With classical photon mapping, a number of rays are sent from random positions on the light source into the scene. The position where each ray intersects the mesh is then stored in a photon map along with the ray intensity and incident vector. Then when the scene is visualised, rays are shot into the scene for each pixel on the screen from the camera. When these rays hit something, they look in the photon map how many photons are nearby to efficiently compute the flux at that position.

For our ray tracer, since we ultimately want a heatmap that is accurate, but not necessarily highly detailed, we might consider deviating from regular photon mapping in some ways. For example, each photon could instead compute its own dosage and place this into a dosage map. Then when visualising the scene, instead of gathering the number of photons at pixel world space positions, the nearest photon already knows the dosage at its position, which is then converted to color. This would offload more work from the visualisation stage to the precomputation stage, but would result in a less detailed visualisation. Unfortunately this would also lead to a higher computation cost per photon, which does not scale well to the multiple area lights that need to be accounted for, since these require many rays to be reasonably approximated.

An issue with classical photon mapping is that it is liable to some artifacts. The irradiance computed at a surface can be influenced by photons that have hit nearby surfaces with completely different normals. Simply discarding these photons from the calculation is not as simple as it seems, as the surface area part of the equation would need to be proportionally adjusted as well. This a difficult problem, and currently no perfect solutions exist, though some improved versions have been proposed [10]. For example by using ellipses instead of spheres to sample photons, edge bleeding is significantly reduced, yet not entirely eliminated.

A solution to this problem is to, rather than each frame recalculating the dosage per screen pixel, simply precompute the dosage per triangle. Therefore calculating dosage by counting the number of photons that hit each triangle and dividing by its surface area. This allows bypassing the creation of a photon map that needs to store photon positions in a data structure such as a kd-tree, and instead storing a photon count integer for each triangle in an array. This reduced computation cost allows shooting much more rays, which couples perfectly with the many area lights that need to be handled, which require many rays for proper Monte Carlo integration. The obvious drawback of this increased efficiency is that we only show one dosage color per triangle. However, we do show correctly this average dosage per triangle, and the level of detail and accuracy to the real room now only depends on complexity and accuracy of the mesh. Meshes made with Polycam for example do not show huge differences between triangle size within a single mesh.

Since this method produces a single color per triangle, the user can much more easily see how inaccuracies in the mesh affect the displayed dosage. It also ensures that we do not make any assumptions about for example the smoothness of the mesh, and the accuracy of the displayed dosage map (once converged with enough photons) relies purely on the accuracy of the mesh geometry and the properties of the lights.

One issue that per-triangle photon counting introduces is that tiny triangles will detect either extremely low or extremely high irradiance, as they are unlikely to be hit by a photon, but inflate their irradiance whenever they do get hit, since they divide their photon count by a tiny surface area. Luckily in practice enough photons can be traced within a reasonable amount of time to converge to a stable and accurate dosage map, as the power per photon is reduced and the number of photons per even the tiny triangles is big enough to prevent the fireflies from occurring.

2.3.3 Comparison

An advantage of the backward tracing method is that it has less random noise, since the different vertex/detector points in the dosage map calculate their received UV dosage by directly tracing rays to the different lights. The forward tracing method on the other hand has a level of noise mostly depending on the number of photons that are shot randomly into the scene from the lights. With enough photons this method minimises this noise to be negligible enough to match the accuracy of the backward tracing method, though the question remains whether this number of photons is low enough for forward tracing to be significantly quicker than backward tracing.

An advantage of the forward tracer on the other hand is that it shoots much fewer rays towards regions that are either obstructed or too far away, whereas backward tracing shoots just as many rays for those areas as the brightly lit areas. Areas that have very low irradiance do not need to have high accuracy, since their UV dose is going to be below the acceptable level anyway. Since a single light position rarely covers the entire mesh, or else a movement trajectory across the room would barely be needed, this is a significant advantage.

Another point in favor of forward tracing is that the computation cost per ray is much lower, which means that a lot more rays can be cast. This means that area lights can be more accurately approximated, of which there can be potentially very many. Also, if we were to perform backwards tracing per triangle, we would calculate the UV dose at the center of the triangle, but this does not necessarily reflect the average UV dose received by the entire triangle, as you would again need to cast more rays to approximate this better.

In theory it would be better to have a less detailed but more accurate dosage map, which backwards tracing can provide, than a detailed yet noisy one, as can be the case with forward tracing. However, to better handle the multiple area lights, many rays are needed, and combined with a proper acceleration structure like a BVH, per-triangle photon mapping converges fast enough to be superior to its backwards tracing alternative.

2.4 Visualisation

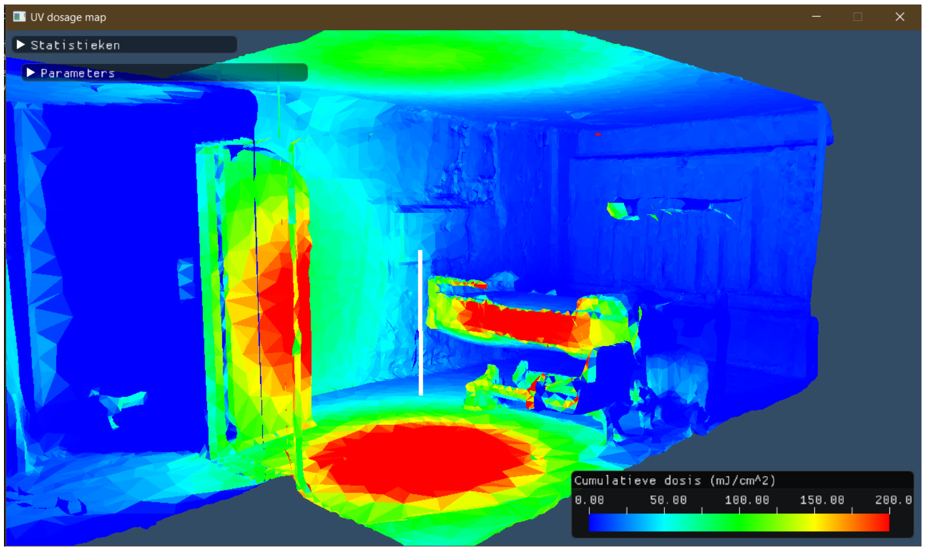


Figure 1: Heatmap for a single light position

To visualise the scene we could simply trace rays from the camera and intersect with the mesh as is usually done with ray tracing. However, ray tracing like this is not necessary in our case, since we have precomputed a color per triangle. This means that rasterizing the triangles would be much more efficient. OpenGL is the standard for rendering meshes using rasterization, so we opt to use this API directly, in favor of rendering an image via ray tracing. The UV dose for each triangle is normalized and scaled along a gradient from blue to red, where blue equals a low dose and red a high dose. The gradient is scaled such that a user-supplied minimum UV dose maps to the color green.

2.5 Data structures

Advanced sampling methods do not seem relevant, since no indirect lighting or reflections, and therefore no longer paths are considered. Considering that many rays will need to be cast to get an accurate dosage map, and the relatively complex meshes produced by 3D scanning, a BVH acceleration structure [6] is implemented. This structure subdivides the mesh into a binary tree of axis aligned bounding volumes, which allows for more quickly testing for intersections with larger areas of the mesh in order to find the primitive that is hit.

2.6 Previous ray tracer implementation

Loop Robots has built a similar ray tracer for simulating how a room is exposed to UV light, which makes use of CUDA. Since CUDA only works for NVIDIA GPUs, it might be better for our ray tracer to use something like OpenCL, so that the program is as widely useable as possible. The ray tracer built by Loop Robots is no doubt useful for studying their techniques, but likely not something that this small project's program could be built off of. Partly to maintain independence from Loop Robots, but also because their ray tracer was ultimately built to fulfill different goals than ours.

3 Implementation

The total UV dose D (in mJcm^{-2}) received per triangle is calculated as

$$D = \frac{I * \sum_{l=0}^{L} (\Delta t_l * n_l)}{A * \frac{N}{L}} * 0.1, \tag{1}$$

where I (in Watt) is the intensity of the light source, L is the number of light positions, n_l is the number of photons received by the triangle from light position l, Δt_l is the time duration per light position, A (in m^2) is the surface area of the triangle, and N is the total number of photons. N/L is then the number of photons per light position. The fraction is multiplied by 0.1 to convert from Jm^{-2} to mJcm^{-2}.

The maximum irradiation per triangle (in μWcm^{-2}) is calculated similarly as follows:

$$\phi_{max} = \frac{I * \max_{l=0}^{L} n_l}{A * \frac{N}{L}} * 100, \tag{2}$$

where I (in Watt) is the intensity of the light source, L is the number of light positions, n_l is the number of photons received by the triangle from light position l, A (in m^2) is the surface area of the triangle, and N is the total number of photons. The fraction is multiplied by 100 to convert from Wm^{-2} to μWcm^{-2}.

3.1 Code implementation

The program is written in C++ and makes use of both OpenGL and OpenCL, along with some additional libraries. C++ is a language well suited for ray tracing, as it allows programmers to be specific about how they want to utilise memory, which can have a big impact on performance. OpenCL is preferred to CUDA, as it is supported by more GPUs than just NVIDIA's. To get started, the graphics template by Jacco Bikker was used [1].

The final ray tracing program does most of its computation on the GPU, with the help of OpenCL kernels. In design it is inspired by the Wavefront path tracing algorithm [11], in that it consists of a loop over a few separate kernels. This means that it would lend itself well to a potential future expansion of tracing longer paths than just direct lighting. The photon counts per triangle are computed simultaneously for a single light position, but the counts for each light position are merged sequentially. Therefore the kernels are looped over for each light position.

The first kernel in this loop simply generates an array of photon rays for the given light position. To this end, first a random position on the line is picked. Then the direction is calculated by first setting the vertical y component to a random value between -1 and 1, then generating a random point on the unit circle in the x-z plane, and finally scaling these x and z components to match the y component. By doing this we ensure that the direction have a cosine distribution vertically along the light source, and are uniformly distributed horizontally around the light source.

The second kernel extends these rays and intersects them with the scene geometry using its BVH. The code for building and intersecting the BVH was directly adapted from [2]. It also immediately performs an atomic increment of the number of photons that the hit triangle has received.

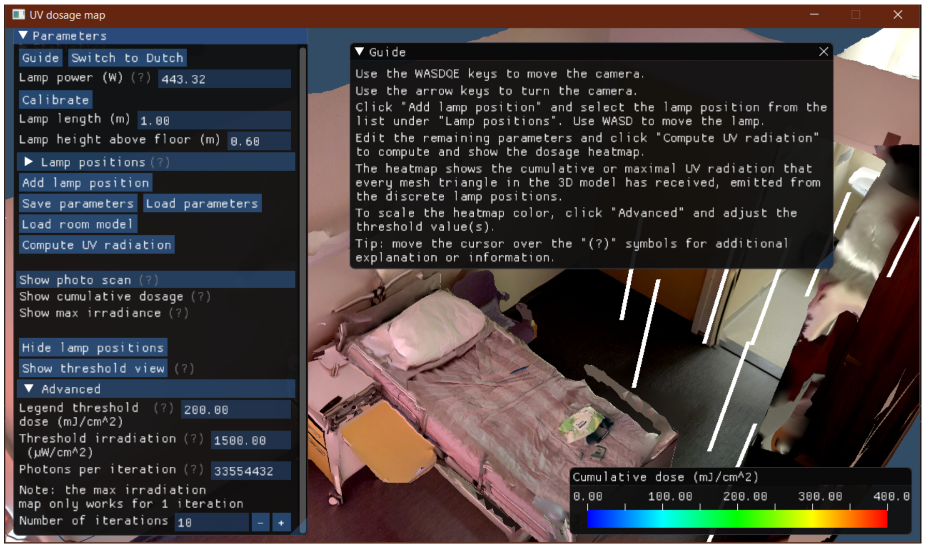
The last kernel in the loop then accumulates these numbers into an array for all the light positions together. For the cumulative dosage heatmap these values are multiplied by the duration corresponding to the light position, and then summed together. For the max irradiation heatmap on the other hand the number of photons is only replaced if the new value is larger.

After finishing this loop, the final two kernels compute the actual irradiance and UV dose by implementing equations 1 and 2. These values are then normalised and scaled over the heatmap gradient. The color values are inserted into an OpenGL vertex buffer which has been made directly available to the OpenCL kernel. This buffer is used by the OpenGL vertex shader to directly shade the vertex with the color stored in the buffer.

The entire computation can be spread out over multiple iterations, which is recommended to reduce the amount of memory that is used for the high number of photons. After 10 iteration of 2^{25} photons each, the effect additional iterations becomes barely noticeable. One drawback is that in the current implementation, the maximum irradiance map can only computed for one iteration.

4 Results

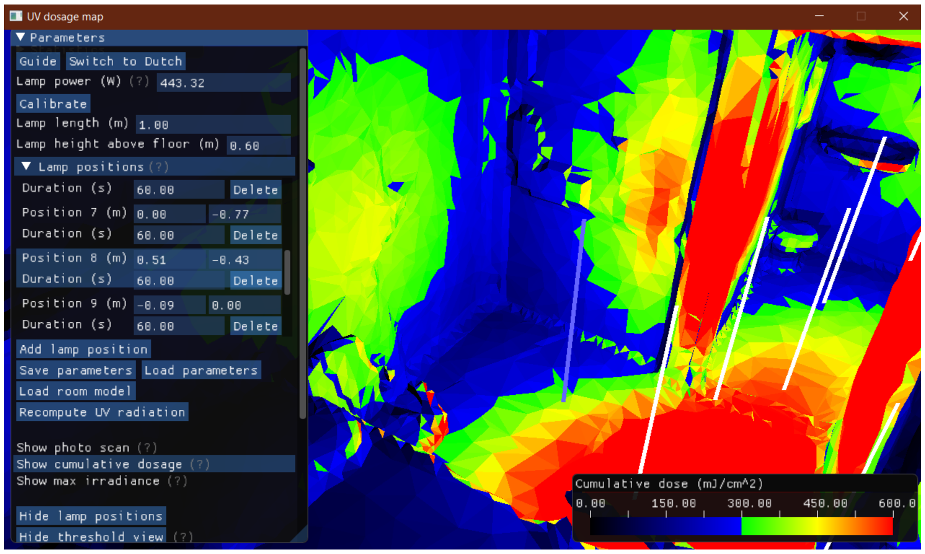
4.1 User Interface



```
Lamp power (W) (?) 443.32
Lamp length (m) 1.00
Lamp height above floor (m) 0.60
Legend threshold (?) 200.00
dose (mJ/cm^2)
Threshold irradiation (?) 1500.00
 (µW/cm^2)
Photons per iteration (?) 33554432
Note: the max irradiation
map only works for 1 iteration
Number of iterations 10
```

Use the WASDQE keys to move the camera.
Use the arrow keys to turn the camera.
Click "Add lamp position" and select the lamp position from the list under "Lamp positions". Use WASD to move the lamp.
Edit the remaining parameters and click "Compute UV radiation" to compute and show the dosage heatmap.
The heatmap shows the cumulative or maximal UV radiation that every mesh triangle in the 3D model has received, emitted from the discrete lamp positions.
To scale the heatmap color, click "Advanced" and adjust the threshold value(s).
Tip: move the cursor over the "(?)" symbols for additional explanation or information.

Figure 2: An overview of the UI



```
Lamp power (W) (?) 443.32
Lamp length (m) 1.00
Lamp height above floor (m) 0.60
Duration (s) 60.00
Position 7 (m) 0.00 -0.77
Duration (s) 60.00
Position 8 (m) 0.51 -0.43
Duration (s) 60.00
Position 9 (m) -0.09 0.00
Duration (s) 60.00
```

Figure 3: The dosage map of multiple light positions. One light position is selected from the list. This light position is also seen in the center of the screen as a purple line, whereas the other light positions are white lines. The threshold view is enabled, causing any dose below 300 mJ/cm^2 to be colored dark blue.

The model can be rotated using the arrow keys, and the camera can move around using the WASD keys as well. Light positions are visualised as vertical white lines in world space. In the bottom-right a legend is shown that assigns the appropriate values to the heatmap colors.

A typical walkthrough of how a user would use the application is as follows:

- Open the program.
- Select a different mesh file if necessary.
- Fill in the lamp height and length.
- Press "Calibrate".
- Fill in the height, distance from the lamp, and irradiance from a measurement done with the UV lamp.
- Press "Calibrate" to appropriately scale the lamp power.
- Create or edit the light positions and durations. These are shown in a scrollable list, and can be individually selected. Once selected, the WASD keys can be used to move the light position around the room. This provides a more user friendly way of creating a collection of light positions without having to manually enter coordinates for each of them. For more specific requirements, such as placing the light 2 meters from the door, the user

could use WASD to align the light with the door, and then subtract 2 meters from the x or y coordinate (assuming the door is axis-aligned, as seems to commonly be the case for meshes obtained via Polycam).

- Press "Compute UV radiation".
- If necessary, edit the legend threshold dose and/or irradiation This is used to scale the mapping of UV dose to color.
- Inspect the resulting heatmap by moving the camera around.
- If desired, switch between showing a view of the room colored with the scanned mesh texture, the cumulative UV dose heatmap, or the maximum irradiation heatmap.
- Edit the light positions and durations where necessary.
- Press "Recompute UV radiation".
- Repeat the previous five steps until satisfied.
- Save the parameters.
- Use the light positions and durations as a guide when creating a new route for the UV light robot in its own application.

4.2 Limitations

As with all simulations, some concessions had to be made that keep the program from generating an exact representation of the real world. Here all the potential sources of inaccuracy are summarized into a brief overview. Note that only the first three points can lead to a potential overestimation of the UV dosage.

- The accuracy of the mesh scan. This affects whether surfaces are occluded or not, and inaccurate slopes mean inaccurate irradiation.
- The number of photons. Shooting less photons means averaging over a smaller sample size and increases the level of noise in the resulting heatmap.
- The lamp approximation. The lamp is modeled as a single vertical line that distributes light evenly around it. The real lamp is often more complex, consisting of multiple light tubes, which results in a larger radius of the area light and a non-uniform pattern in the directions it sends out light.
- The parameters entered by the user. If the user enters for example an incorrect height or length for the light, this inaccuracy is naturally reflected in the end result.

- Lamp movement. Currently we only simulate a collection of light positions, but do not take into account the time in which the robot moves between positions. This results in a more conservative estimation of the UV dosage.
- Air and surface conditions that can hinder the propagation of the UV radiation. Since these are ignored, they might affect the strength of UV doses and the calculated UV dosage might be more conservative that in reality.
- Reflections and indirect lighting. Since these are not modeled, the resulting heatmap will be more conservative.
- A UV lamp typically requires around 2 minutes to warm up to get to a consistent power output. This means that part of the room will have already received some UV radiation before the route is truly started, which the ray tracer does not take into consideration.

5 Future work

While the ray tracing program presents a big improvement in accuracy compared to previous work which ignored shadows and vertical surfaces, there are still many ways in which it could be expanded upon. The most obvious of these are measures to reduce some of the inaccuracies that are still present, as mentioned in the previous section on the limitations.

For instance, cylinder light sources rather than vertical lines would more accuratly approach the geometry of the actual UV lamps. These were initially not implemented since they increases complexity of the simulation, and require more photons for the simulation to converge. Implementing cylinder light source would essentially lead to more accurate soft shadows.

Another improvement could be more advanced calibration and validation of the UV doses received at certain distances. Several measurements have already been taken, which can currently be individually used to automatically do a naive linear scaling of the lamp power based on how they compare to the simulation. To ensure that the UV doses are accurate to the real room, it would be good to investigate more thoroughly whether this is accurate enough, and to validate this with other measurements as well.

On another note, the current ray tracer efficiently computes the UV dose per triangle through precomputation. An alternative approach would be implementing a ray tracer that recomputes the dose every frame, but does it per screen pixel rather than per triangle. This would result in a more detailed heatmap, but would require some proper optimisation to achieve real time speeds, which might be challenging given the multiple area lights. This might restrict computers with lower-end GPUs from being able to properly use the program.

Modeling the movement between light positions would also be a significant improvement. This takes away pressure from the user to add additional light positions to interpolate this movement themselves. A moving vertical line light can be treated as a plane from which random ray origin points can be picked. Once an origin point is picked, the photon ray is treated as if coming from a rod to determine its direction. The delta time for the UV dose is then simply the time in which the lamp moves from one end of the plane to the other. To improve the uniformity of the light sample points, stratification could be employed, which means subdividing the light area into uniform strata and sampling a random point within each of those strata.

Lastly, the ultimate goal of this project was to create a tool that makes it easier to create a route for robots mounted with UV lamps to travel, to disinfect a room with UV light. However, the current program only really shows the effect of given light positions, but it is left to the user to find the optimal light positions. Therefore a nice addition would be to create an additional layer of abstraction that solves this optimisation problem of using as little time and UV light to disinfect the room with a given minimal UV dosage.

5.1 Known bugs and issues

At the end of the project there were still a few issues left, partly due to time constraints and their lower priority:

- The light lines are drawn in front of all geometry, and also cause lines too be drawn when they should be outside the camera's view.
- For a photon count of 2^{27} and for some values above, the heatmap becomes black. This is most likely a memory issue which corrupts the dosage color vertex buffer. It seems to also occasionally happen to values slightly below this on the first run. Due to the high memory use of multiple gigabytes in these cases, photon counts this high should be avoided regardless.
- The maximum irradiance map is only computed over 1 iteration.

References

[1] Bikker, J. advgrtmpl8. `https://github.com/jbikker/advgrtmpl8`, 2022.

[2] Bikker, J. bvh_article. `https://github.com/jbikker/bvh_article`, 2022.

[3] Cline, D., Jeschke, S., Razdan, A., White, K., and Wonka, P. Dart throwing on surfaces. *Computer Graphics Forum 28*, 4 (June 2009), 1217–1226.

[4] Cook, R. L., and Torrance, K. E. A reflectance model for computer graphics. *ACM Transactions on Graphics (TOG) 1*, 1 (1982), 7 – 24.

[5] Corsini, M., Cignoni, P., and Scopigno, R. Efficient and flexible sampling with blue noise properties of triangular meshes. *IEEE transactions on visualization and computer graphics 18* (01 2012), 914–24.

[6] Gunther, J., Popov, S., Seidel, H.-P., and Slusallek, P. Real-time ray tracing on gpu with bvh-based packet traversal. In *2007 IEEE Symposium on Interactive Ray Tracing* (2007), pp. 113–118.

[7] Hulburt, E. O. The reflecting power of metals in the ultra-violet region of the spectrum. 205.

[8] Jensen, H. W. *Realistic Image Synthesis Using Photon Mapping.* A. K. Peters, Ltd., USA, 2001.

[9] Lai, P.-Y., Liu, H., Ng, R. J. H., Wint Hnin Thet, B., Chu, H.-S., Teo, J. W. R., Ong, Q., Liu, Y., and Png, C. E. Investigation of sars-cov-2 inactivation using uv-c leds in public environments via ray-tracing simulation. *Scientific Reports 11*, 1 (2021).

[10] McGuire, M., and Luebke, D. Hardware-accelerated global illumination by image space photon mapping. In *Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), HPG '09, Association for Computing Machinery, p. 77–89.

[11] van Antwerpen, D. Improving simd efficiency for parallel monte carlo light transport on the gpu. HPG '11, Association for Computing Machinery, p. 41–50.

Appendices

A Planning

There were weekly meetings with the supervisor to discuss work done during the past week and work planned for the next week. Meetings with ZorgSaam happened around every 3 weeks. The first two to three weeks mainly consisted of literature study. Then came the actual implementation, followed by slightly over week to finish up the report and to give a final presentation.

A.1 Feature planning

A planning edited to show when features were finished.

Week	Date	Feature
1	9-9	
2	16-9	
3	23-9	Finished literature review & planning of the ray tracer implementation
4	30-9	Created initial basic program that can load and display GLTF meshes
5	7-10	Visualized a handmade dosage map using OpenGL
6	10-10	Meeting ZorgSaam/Microvida
	14-10	Filled dosage map with basic light intensity by tracing from the light positions
7	21-10	Implemented per-triangle photon counting
8	28-10	Allowed users to enter parameters and to add and save multiple light positions
		Created max power map
9	3-11	Meeting ZorgSaam/Microvida
	4-11	Implemented BVH
		UI improvement, added heatmap color legend
10	10-11	Final presentation
	11-11	Finish report
		Finish final tweaks to program

B Reflection

The project has been a lot of fun developing a ray tracing application, despite the actual ray tracing in the end product being relatively simple. Working together with people from a different discipline has been a valuable experience. Doing a solo project in its own right helped me get better at planning a (relatively) large project, keeping up with and reevaluating features and deadlines. The planning could have been smoother, as around halfway through I realised some issues with the direction that the ray tracer was heading, which were better solved using a different ray tracing method. Luckily not too much time was lost on this pivot. Perhaps this could have been prevented by spending more time on research and thinking over the ray tracing design before moving on to the implementation, but much time was already spent in that regard, and sometimes things just become much more obvious once implemented. Another aspect in hindsight is that I could have used the second examiner Jacco Bikker more for advice, but hadn't taken him up on this offer due to being unable to formulate a specific question regarding the ray tracer. Perhaps an occasional check-in with the ray tracing method as a whole would have been a valid alternative that could have prevented some hiccups in the planning.

C Cumulative dose vs Maximum irradiance

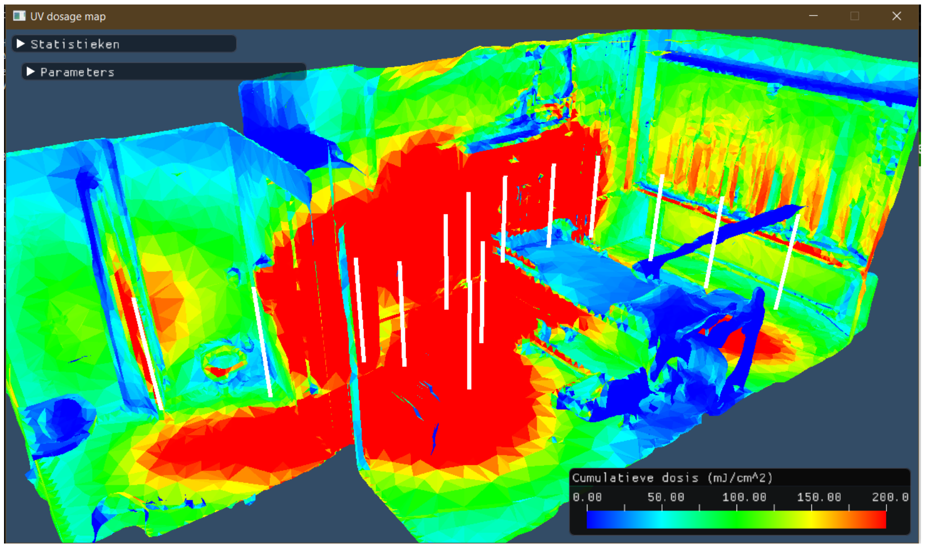


Figure 4: A cumulative dosage map for multiple light positions

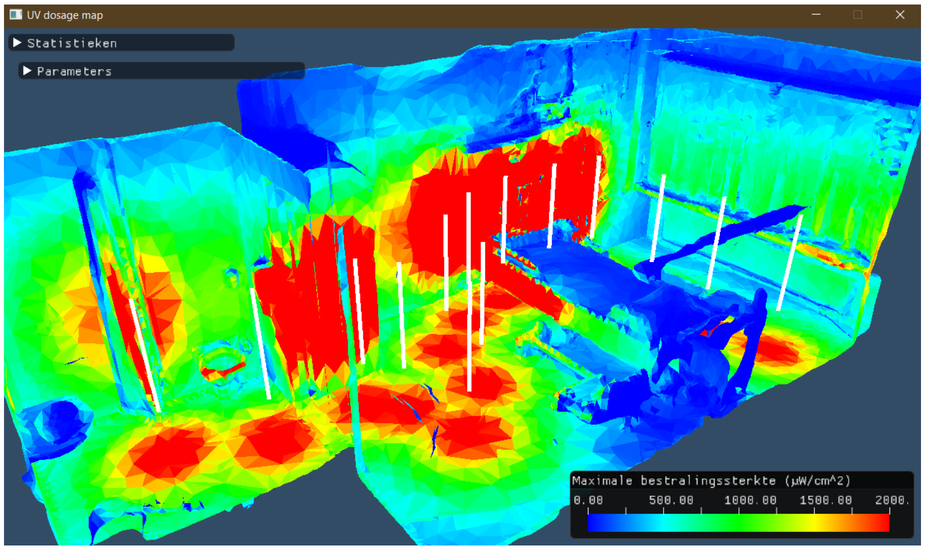


Figure 5: A maximum irradiance map for multiple light positions