university of groningen

faculty of science and engineering

Virtual Ray Tracer: Distribution Ray Tracing

Bachelor's thesis

July 14, 2022

Student: J.R. van der Zwaag

Primary supervisor: Prof. Dr. J. Kosinka

Secondary supervisor: Dr. S. D. Frey

Abstract

Visualizing ray tracing helps to understand how ray tracing works. C.S. van Wezel and W.A. Verschoore de la Houssaye made an application in their bachelor's thesis that does exactly that, named 'Virtual Ray Tracer'. But simple ray tracing using just point lights has drawbacks. It cannot render phenomena such as soft shadows and not all lights can be approximated by point lights.

In this thesis, we discuss how distributed ray tracing, implemented with Monte Carlo methods, and different lights can be visualized by extending the aforementioned program. Users can see how super-sampling works, have more control over how rays are visualized and learn about different types of lights and light attenuation. The goal is to help users understand these concepts.

We conducted a user study to find out how useful the application is in teaching users the newly added concepts. From the user study, we can derive that the program helps users with some Computing Science knowledge to understand the added features and that different ways of visualizing rays are much appreciated. Most users with little to no background in IT found it very difficult and complex. We discuss these results and propose solutions and research ideas to further improve Virtual Ray Tracer.

Contents

Chapter 1

Introduction

In the field of Computer Graphics, ray tracing is a technique used to render realistic images. However, this comes at a big computational cost. In the last few years, big steps have been made towards real-time ray tracing and the importance of ray tracing has been rising with it. But truly understanding ray tracing is easier said than done.

To aid with learning ray tracing, Chris van Wezel and Willard Verschoore de la Houssaye developed a tool [1, 2, 3] that visualizes ray tracing. This program, called 'Virtual Ray Tracer' (VRT), shown in Figure 1.1, is aimed at Computer Graphics students to help them understand ray tracing.

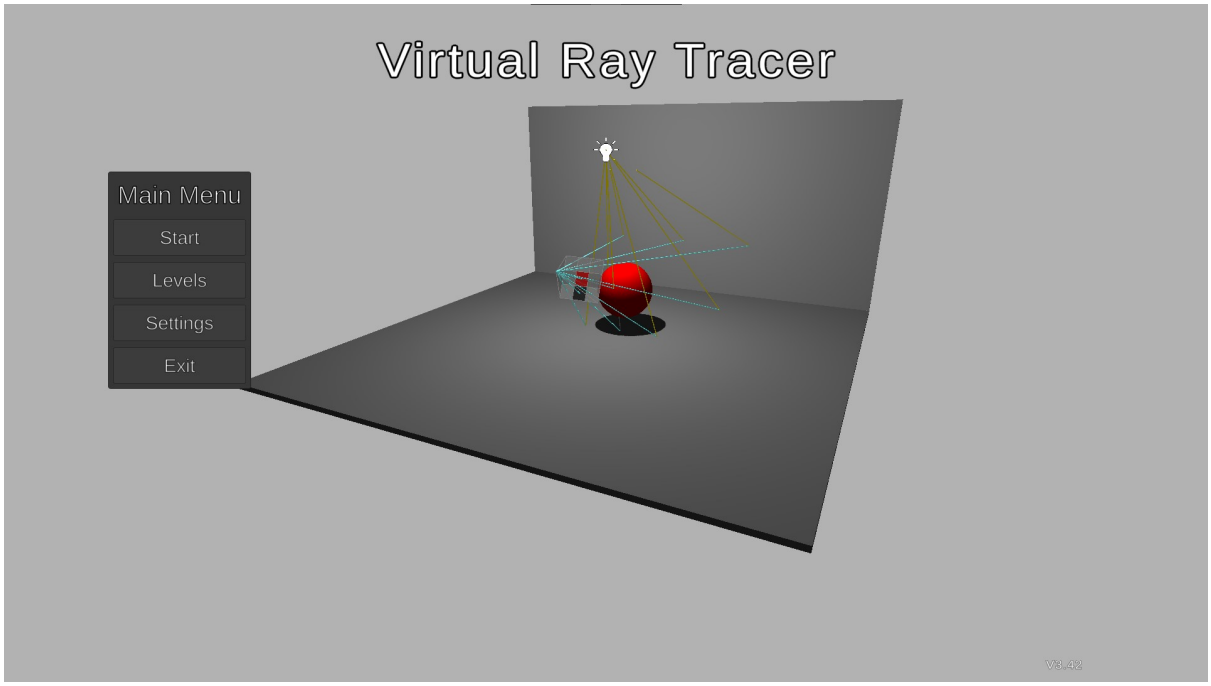


Figure 1.1: Virtual Ray Tracer [3] program developed by Chris van Wezel [1] and Willard Verschoore de la Houssaye [2]

This tool explains ray tracing in steps, using a total of 7 levels. At the start of each level, there is an explanation about either how to use the program or how a certain aspect of ray tracing works. After that, the user can explore the explained features or concepts by playing with the tool in that level. The last level is a 'sandbox' level, in which the user can create their own scene and explore for themselves. When the user has gone through

all the levels, they should have a better understanding of what ray tracing is and how it works.

However, the program only covers the basics. To get results that look even more realistic, advanced ray tracing has to be used. For example, not all lights can be approximated with point lights. To create the phenomenon called 'soft shadows' we need an area light. This requires different ray tracing techniques.

To resolve these shortcomings, we extended the tool with 'distributed ray tracing' features and some additional features. Distributed ray tracing covers many different phenomena, and not all can be (easily) visualized. Therefore, we chose to go with two that can also be visualized: anti-aliasing by using super-sampling, and soft shadows. The aim is to help the users to understand these concepts by visualizing them. This led to the following research questions:

1. How can Monte Carlo / distributed ray-tracing be visualized?
2. Can the Virtual Ray Tracer program be modified to also visualize Monte Carlo / distributed ray tracing?

Alongside this project, four other students were also working on Virtual Ray Tracer. In particular, Roan Rosema worked on making VRT work in the browser and on mobile devices, and Peter Jan Blok worked on a gamified VRT [4]. In my project, I also used (part of) their work.

In this thesis, we start with information about distributed ray tracing and ray tracing visualization in Chapter 2. In Chapter 3 we discuss the used distributed ray tracing concepts and how they can be taught and represented visually. How all of that is implemented is described in Chapter 4. Chapter 5 contains our user study and its findings followed by a conclusion in Chapter 6. We conclude with potential future work in Chapter 7.

Chapter 2

Background

In this chapter we discuss concepts of (distributed) ray tracing and present ray tracing visualization tools.

2.1 Ray Tracing

Ray tracing is a rendering technique that can result in very realistic images, as it is close to how light works in the real world. Light sources emit light particles, called photons, and these travel in a straight line until they hit an object. What happens at that hit-point depends on the object's material. If it is transparent it could go through the object, if it is a mirror it will be reflected. Eventually, some of these photons will reach our eyes, or a camera, where they are processed into an image.

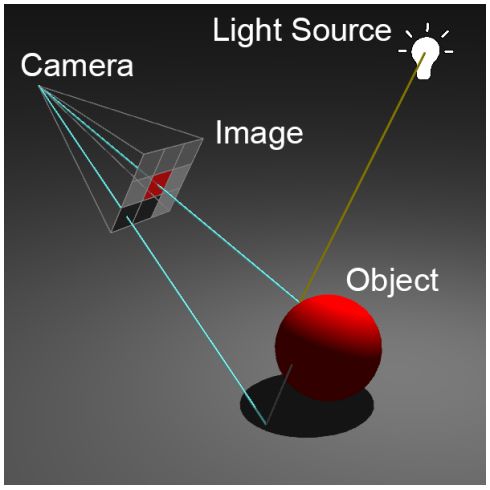


Figure 2.1: Ray tracing example.

This process, but then in reverse, is called ray tracing [5]. A depiction of the ray tracing process can be seen in Figure 2.1. In front of the camera (or eye) is a screen. This screen has a certain resolution, e.g. it could be 3 pixels wide and 3 pixels high. So-called 'rays' are cast from the camera (or eye) through a pixel. A ray may hit an object. The object's material and the location where the ray hit the object determine the color of the pixel. Depending on an object's material, new rays may be cast, for example, due to reflection or refraction, which further influences the pixel's color. This process ends when the ray does not hit anything or when it reaches a light source. It could be that a

ray gets reflected/refracted indefinitely, so often there is also a limit to how many times a ray can be reflected or refracted. Repeat this for every pixel and we have an image.

2.2 Distributed Ray Tracing

The term 'Distributed Ray Tracing' was first coined by Robert L. Cook in 1984 [6]. It essentially comes down to integrating (or 'distributing') rays over some domain to create certain phenomena. For example, soft shadows can be created by distributing rays over the area of an area light source. Another example is an interval of time. By distributing rays over time, motion blur can be rendered. But to actually render an image with distributed ray tracing, an equation must be solved which includes integrals. These integrals appear because the rays are distributed over an interval or range. Unfortunately, even for computers, these integrals are difficult to compute.

2.2.1 Monte Carlo Methods

An intuitive solution is to not compute the integrals but approximate the result. A way to efficiently approximate integrals is the Monte Carlo method [5]. It takes random values from the interval and uses the outcomes to approximate the real value. For soft shadows, it would take random points on the area, average the result and use that as an approximation. The downside is that many samples are needed for an accurate approximation, but the upsides are that the method itself is quite simple and that its computational cost can be adjusted depending on the desired quality.

2.3 Ray Tracing Visualization

Several ray tracing visualization tools already exist, but not all are aimed to educate users about ray tracing. One of the existing tools is Rayground [7]. Rayground is an online, web-based, interactive tool where the user can program and test their own scenes and ray tracing implementation; see Figure 2.2. This requires the user to have experience in coding and it, unfortunately, does not visualize the rays themselves.

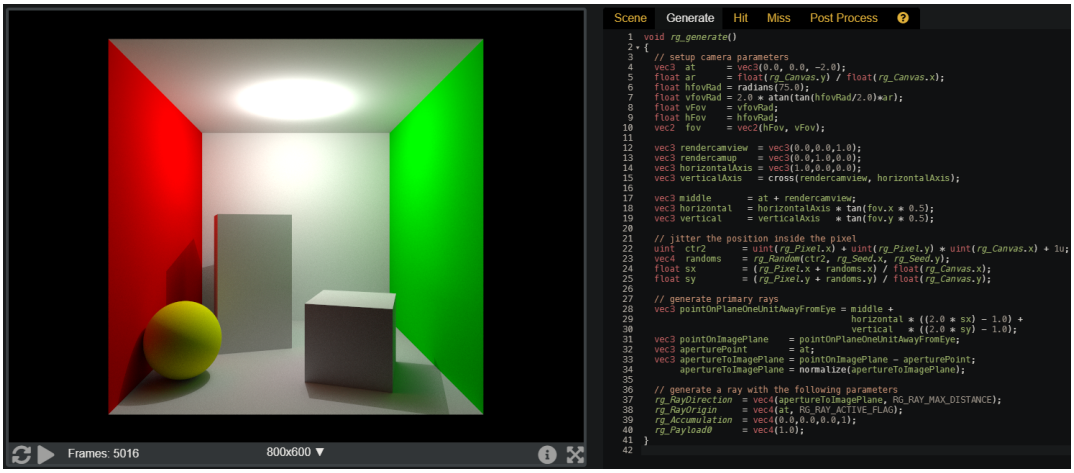


Figure 2.2: The interface of Rayround [7] showing the rendered scene (left) and the editor (right).

Another existing tool is the 'Ray Tracing Visualization Toolkit' [8]. This tool allows you to analyze ray-based rendering algorithms, see Figure 2.3. It works as a plugin for the user's own ray-based rendering program, after which the recorded rays can be visualized. This might be very useful for experienced users, but not for users relatively new to ray tracing. Another downside is that it is not interactive.

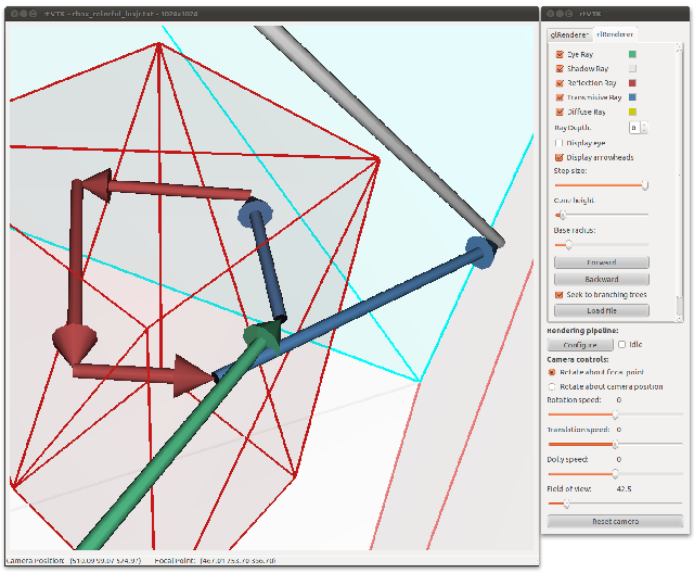


Figure 2.3: The interface of the Ray Tracing Visualization Toolkit [8] showing part of a scene (left) and controls (right).

Virtual Ray Tracer in Figure 2.4 is specifically aimed at users new to ray tracing. It allows for dynamic scenes with animated visualization of the rays. Throughout several levels with different scenes, the basics of ray tracing are both explained and every traced ray can be visualized. It already contains one distributed ray tracing concept, super-sampling, but this is not visualized.

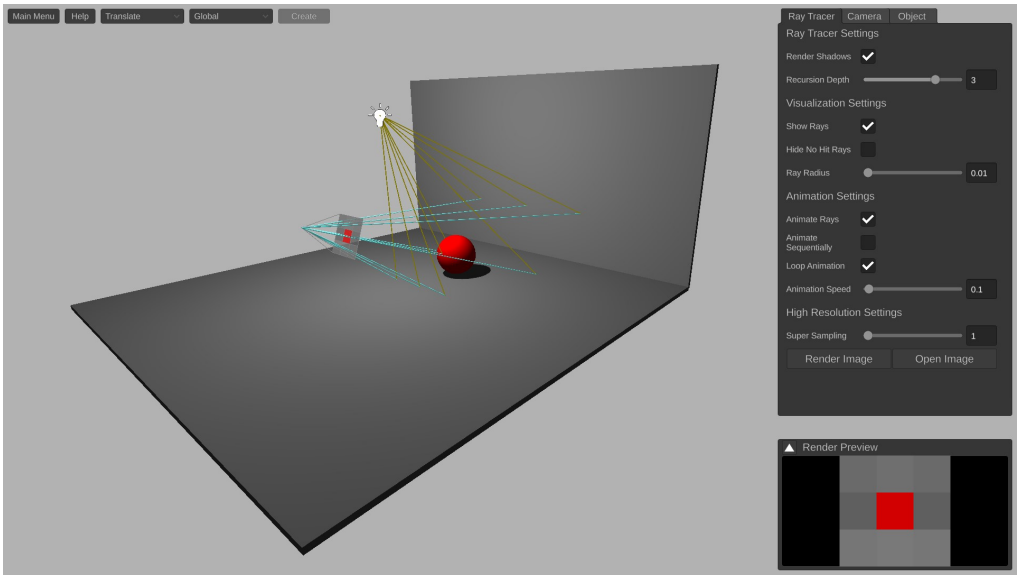


Figure 2.4: VRT's interface with a scene on the left and controls on the right.

Chapter 3

Concept

In this chapter, we go over the distributed ray tracing concepts that we implemented in VRT. We also discuss why we chose these features and only these features. We also go over additional features added to VRT.

3.1 Super-Sampling

Super-sampling is a form of anti-aliasing. Aliasing is an effect where edges are jagged. This effect occurs when the resolution of an image is simply too low to correctly display an object. This quickly happens when an edge is round or diagonal, as pixels of an image are laid out horizontally and vertically. Super-sampling reduces this effect with an intuitive approach: simply render at a higher resolution. The normal render approach in ray tracing is to determine the pixel color by shooting a ray through the middle of the pixel. A visual of this approach can be seen in Figure 3.1.

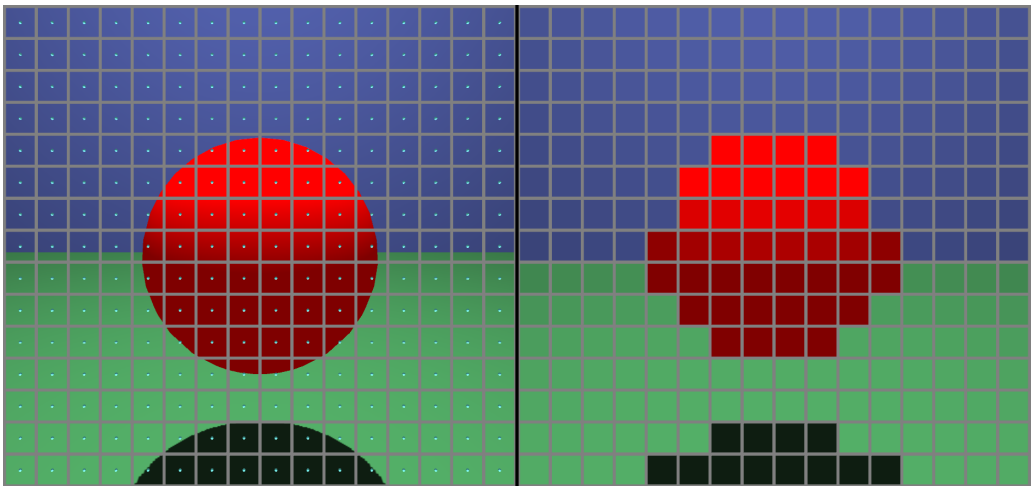


Figure 3.1: Rendering with no super-sampling. Left is the scene with a screen overlay where each pixel has one ray in the middle of the pixel, right is the rendered image.

With super-sampling, we do not shoot a single ray through the middle of the pixel, but multiple rays distributed over the pixel's area. Every ray counts as a sample, and each ray contributes $\frac{1}{\#\text{samples}}$ to the pixel's final color. There are many different algorithms to determine how we distribute the rays over the pixel's area. We chose to go with uniform sampling. This means that all samples are spread out evenly, not only within the pixel,

but across the entire image, as can be seen in Figure 3.2. We chose this method because it is simple and also great to visualize, as we can shoot a ray for each sample and all those rays are evenly distributed.

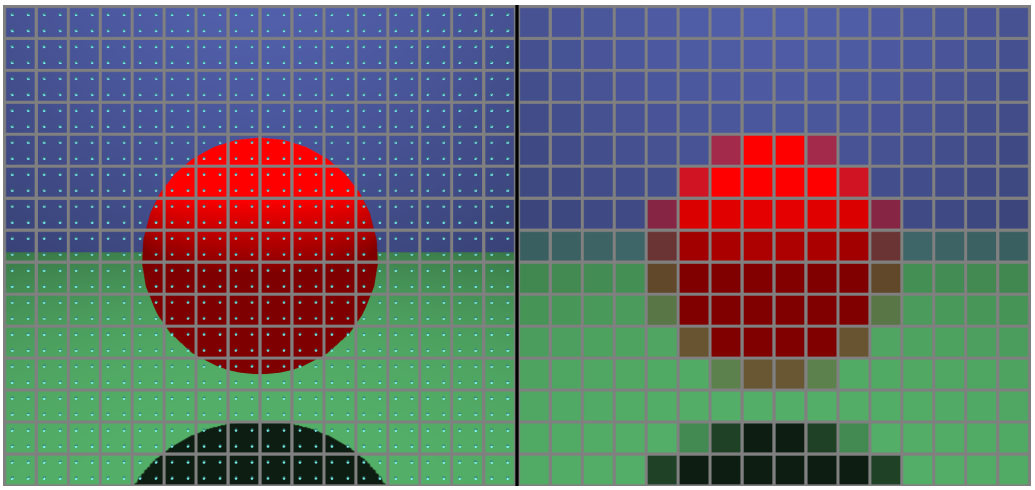


Figure 3.2: Rendering with super-sampling using 4 samples per pixel. Left is the scene with a screen overlay where each pixel has 4 rays distributed over the pixel, right is the rendered image.

3.2 Soft Shadows

There are two major types of shadows: hard shadows and soft shadows. Hard shadows have hard edges. A point can either reach the light source (no shadow), or a point cannot reach the light (full shadow). There is nothing in between. Soft shadows do have this in-between where it gradually goes from no shadow to full shadow. Hard shadows are often seen as unrealistic because almost all shadows in real life are soft shadows if you look close enough. However, there are reasons to go with hard shadows. They are realistic enough in some scenarios, very simple to understand, and much easier to compute compared to soft shadows.

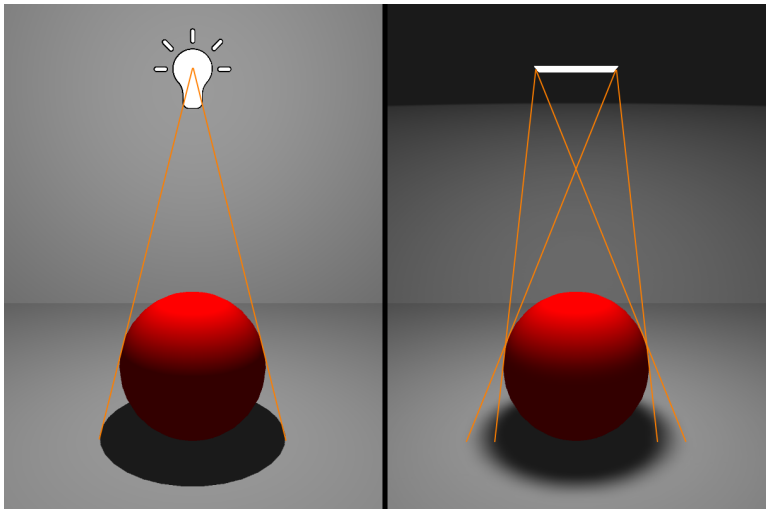


Figure 3.3: Hard shadows (left) vs soft shadows (right)

This high computational cost of soft shadows has to do with how soft shadows arise. They come from a light source that does not illuminate from a point, but from an area, an area light. If we look from a random point in a scene, the area light might be fully visible or not visible at all. Or, it can be partially visible. The regions in a scene from where the light source is partially visible also get partially illuminated, and therefore are partially in shadow. For a visualization of this concept, see Figure 3.3.

To compute soft shadows, we need to determine how much of the light's area is visible. That is unfortunately easier said than done. To get an exact answer, we need to solve an integral equation over the light source's area. As mentioned before, this is very hard. To make this process faster, we use the previously mention Monte Carlo methods. Instead of calculating the integral, we approximate it by taking samples on the area light.

3.3 New Light Types in VRT

As said, soft shadows are an effect of area lights. So Virtual Ray Tracer has to be extended with this new type of light. Point and area lights have almost nothing in common, except that they are lights. However, point lights and spot lights have in common that both are an infinitely small point, and spot and area lights have in common that they have a direction and also do not illuminate in all directions. Therefore, we also add spot lights as it offers a step-wise explanation.

3.3.1 Spot Lights

With spot lights, just like point lights, the light source itself is an infinitely small point. It differs itself from the point light with two additional properties: 'direction' and 'spot angle'. They work very much like flashlights.

Point lights do not have a direction, they illuminate a scene in all directions. Spot lights shine light in a cone form in a certain direction. In this direction, it may illuminate a wide region or a very narrow region. This is dependent on its spot angle. A wide spot angle means it illuminates a wide region, a small spot angle results in a small illuminated region. An example can be seen in Figure 3.4.

Spot lights are simple to visualize as the user can see the difference in the illuminated region. As the light source itself is an infinitely small point, just like the point light, the rays work exactly the same as with point lights.

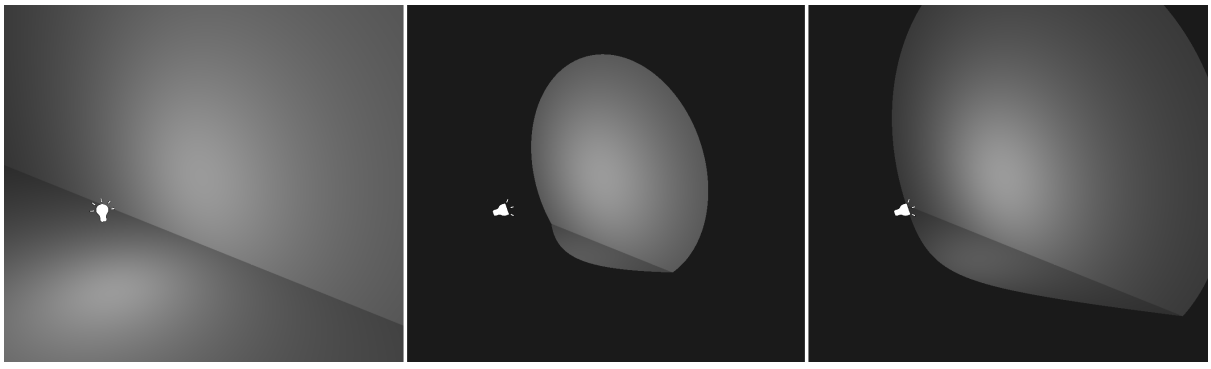


Figure 3.4: Point light (left) vs 90° spot light (middle) vs 120° spot light (right)

3.3.2 Area Lights

Area lights have some things in common with spot lights. They also have a direction and an 'angle'. This angle is however fixed at 180°. Unlike the other lights, the area light is no longer an infinitely small point, but an area. As this is a better approximation of most real-world lights, it really adds to the realism of scenes. The biggest downside of area lights is the high computational cost due to sampling.

An area light can be visualized by a simple rectangle with the front side colored as the light's color and the back side colored black. An example can be seen in the right side of Figure 3.3. As for the rays, we consider two options. We can either show every ray as a ray from the point to the sample location or one big cone from a point to the area light source. The latter can reduce the number of rays dramatically and therefore improve performance. The first can visualize the sampling process. We chose to go with a combination of the two. How this is implemented is discussed in Chapter 4.

3.4 Light Attenuation

To further increase realism, we add light attenuation. Light attenuation means that the light's intensity gradually drops down according to some formula. We use two types of light attenuation: distance attenuation and angle attenuation.

3.4.1 Distance Attenuation

Distance attenuation is a simple way of attenuating light that is realistic because it happens in real life too. It has to do with the 'inverse square law'. The intensity of the light decreases in line with the square distance from the light source. This can be seen in the formula of the area of a sphere: $4\pi r^2$ where r is the radius. As the light travels further, i.e. the radius increases, the area covered by the exact same amount of light particles (photons) increases quadratically. An illustration of this principle can be seen in Figure 3.5

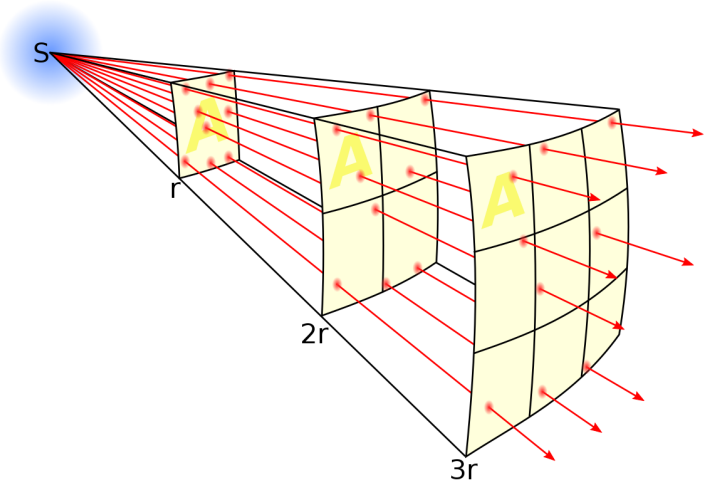


Figure 3.5: Visualization of the inverse square law.
Source: https://en.wikipedia.org/wiki/Inverse-square_law#/media/File:Inverse_square_law.svg

3.4.2 Angle Attenuation

Angle attenuation is a type of attenuation that only applies to light sources that have a direction, so spot and area lights in our case. This effect can also be linked to the flashlight analogy. If you shine a flashlight at a point (point A), that point will be brighter than a point (point B) to the side of point A (assuming the same material properties, no additional light sources). This is because the angle between the line from the flashlight to point A and the light's direction is much smaller than the angle between the line from the flashlight to point B and the light's direction. Similarly, we can lower the spot light's intensity in proportion to the angle between the spot light's direction and the direction to the point from the spot light. A visualization of this concept can be seen in Figure 3.6.

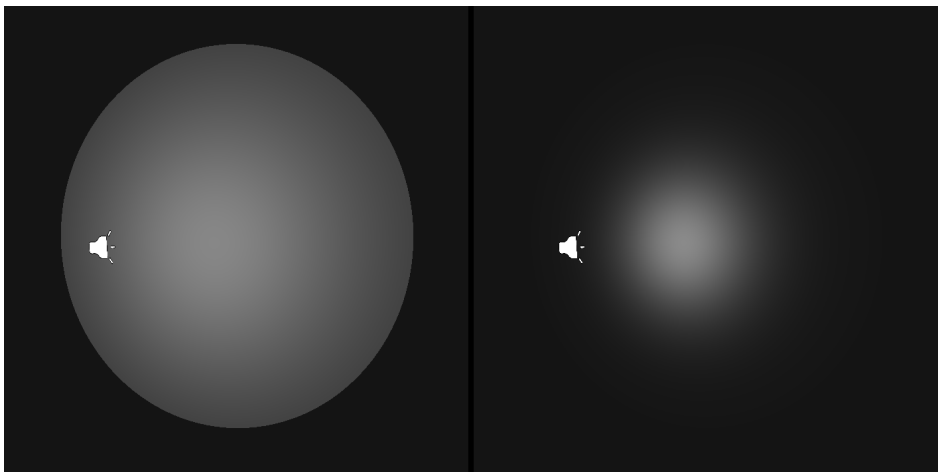


Figure 3.6: Example of no angle attenuation (left) vs angle attenuation (right).

3.5 Ray Visualization

One of the consequences of showing individual rays for area lights and visualizing supersampling is that at some point there may be too many rays to clearly see what is going on. To better show what is going on, we add the ability to change how rays are visualized.

Currently, rays have a color corresponding to their type. All rays have the same radius, controllable by the user, and only rays that do not hit anything can be hidden. We add several additional options to change the rays' appearance to help the user figure out what is happening in the scene.

3.5.1 Hide Negligible Rays

Some rays are really important, as they greatly impact the pixel's color, and some are nearly negligible, as they contribute almost nothing to the pixel's color. There may be a lot of these rays that have barely any impact. To declutter the scene of too many rays, the user may want to hide these rays. To do this, we add an option to hide negligible rays which hides rays that contribute less than a set threshold.

3.5.2 Ray Transparency

By making rays transparent, we can show lots of rays whilst still being able to see the scene. How transparent a ray should be is subjective to how visible the user wants to scene to be and how visible they want the rays to be. The user may not want the rays to be transparent at all. So we add the option to make rays transparent and make the level of transparency controllable by the user.

3.5.3 Dynamic Ray Radius

As said before, some rays are more important than others. Another option to distinguish between these rays is to simply make their size, which is in this case their radius, depend on their contribution. We add the option to make the ray's radius depend on its contribution, which maps every ray's contribution to a radius between a certain minimum and maximum radius set by the user.

3.5.4 Contribution-Based Ray Color

Lastly, we can also visualize where the pixel's color comes from. We add the possibility to change the ray's color to the color it contributes to the pixel's color. This way, the user can visually see how the pixel gets its color.

Chapter 4

Implementation

In this chapter we start with a short explanation of how relevant existing features from VRT are implemented, followed by how we implemented the distributed ray tracing features and extras in the original Virtual Ray Tracer [1, 2, 3].

4.1 Virtual Ray Tracer

Virtual Ray Tracer is an open-source program made with Unity. Unity is a popular, free-to-use game engine that also allows additional script/code written in `C#`. Any additional features added to VRT are also made using Unity and the `C#` language. The application is not ray-traced but uses rasterization to render all objects. The ray tracing is done separately using a custom script. This means we have two pipelines: A ray tracer pipeline that renders only objects in a scene (at a lower resolution) using ray tracing, and a rasterization pipeline from Unity that visualizes the scene, the ray tracer's rays, and the User Interface (UI).

4.1.1 The Design

One of the problems is that Unity uses its own approach of representing a scene that can be used for the rasterization part, but that is not usable for the ray tracer. The ray tracer should, for example, ignore UI elements. So the ray tracer should get a filtered and translated version of the Unity scene data. On top of that, the scene is also dynamic, elements can change. So we should also make sure that Unity sends updated scene data if changes are made.

Unity interacts with two parts of the ray tracer: it handles the ray tracer's input, but also its output. In VRT, this is separated. The input is handled through the `Scene Manager`, which contains the scene data for the ray tracer, and the output is handled through the `RayManager`, which manages the rays that Unity can visualize. Consequently, when it comes to adding new lights, we had to change the `Scene Manager` and its components, and with respect to changing how rays are drawn, we had to change the `RayManager` and its components.

W.A. Verschoore de la Houssaye made an illustration of the general structure of VRT in his thesis [2], shown in Figure 4.1.

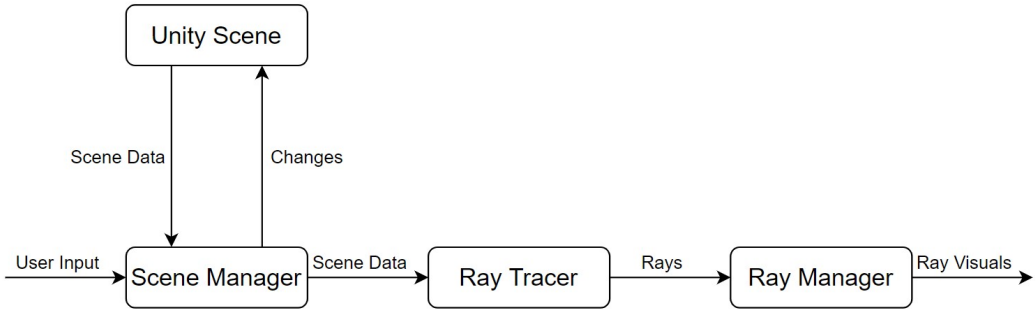


Figure 4.1: A design for the general structure of the application.
Source: W.A. Verschoore de la Houssaye's thesis [2].

4.1.2 Ray Tracer

VRT's ray tracer script contains two (similar) ray tracers. One simply renders an image, the other one returns a tree structure containing all rays created by the ray tracer. The latter is used for the visualization of rays and usually calculates only a few pixels. These rays are stored in `Ray Objects`. Such an object contains the ray's data, originally its origin, direction, type, and color. The color in the ray's data is the color it contributes to the pixel, so we can use that for turning the rays to the color they contribute. How much a ray contributes to the final color is not yet stored, so we had to modify the ray tracer and the `Ray Object` to handle that as well since we need that for transparency and dynamic ray radius.

Currently, each pixel gets its own tree of rays. In order to visualize super-sampling, each pixel needs multiple trees. So we also have to change how the trees are used in the program in order to visualize super-sampling.

4.1.3 Ray Visualization

To visualize rays, the aforementioned `Ray Objects` are used. But because the scene can change, and we do not want to destroy and create new `Ray Objects` every time, a `Ray Object Pool` is used. It handles all the Unity objects in the scene that represent rays. Instead of destroying and creating those Unity objects every time, this object pool (de)activates Unity ray objects in the scene. Every one of these Unity ray objects can be assigned data from an actual `Ray Object` from the ray tracer. Activating and deactivating Unity objects is much faster than destroying and creating objects. Therefore, this approach allows hundreds of rays to be changed without losing much performance.

4.2 New Additions

We added several new features to Virtual Ray Tracer. We start by explaining how we implemented all the new lighting-related features, followed by the ray visualization changes.

4.2.1 Lighting

We added two new types of lights. The first step was to generalize lights. At some points in the code where it accesses lights, it does not matter which type of light it accesses,

as some properties are shared, i.e. position and color. However, some properties are not shared. You can rotate an area light, but not a point light. Therefore, we first made an abstract base class `RTLight` from the `RTPointLight` class, made the `RTPointLight` class extend the `RTLight` class and moved all properties to the base class and left only point light specific properties in the `RTPointLight` class. We also introduced a new enum field `RTLightType` so when lights are accessed, the type can always be determined. After that, we made sure the rest of the code used the base class when appropriate and `RTPointLight` only when necessary. With these changes, it is easier to add the area and the spot light. As distributed ray tracing was the priority, we opted to first implement the area light and then the spot light.

Area Light

A problem with Unity's rendering pipeline used for VRT is that it does not support real-time area lights. So we have to make our own area light approximation with Unity's point or spot lights. This is not necessarily bad, because we can somewhat show the ray tracer's area light approximation as it does not integrate over the area but takes samples. As said in Chapter 3, spot and area lights have in common that they have a direction (and therefore a rotation) and an 'angle', although for the area light it is static at 180°. Hence, we use a number of Unity's spot lights to approximate an area light.

We first made an `RTAreaLight` object in Unity, a so-called 'prefab' (a template object with default settings) that has an image of a rectangle. The area light has a back and front. If the user is in front of the area light source, we make the image the same color as the light source's color. This way, the area light itself is visualized.

Next, we introduced a `lightSamples` variable and make the script distribute Unity's spot lights over the area light's area. The `RTAreaLight` object itself has a position, rotation, and scale. If we add sub-objects, so-called children, they are also impacted by these properties. So if the `RTAreaLight` prefab is a 1 by 1 rectangle with the front-facing along the positive z-axis, we can simply place spot lights accordingly in this rectangle and they will automatically be positioned and rotated correctly in the scene. This makes it quite easy. We uniformly distribute a total number of `lightSamples` $*$ `lightSamples` Unity spot lights over the area, make them face along the positive z-axis and they will illuminate the scene correctly.

The next step is to also make sure it works in the ray tracer. We use jittered sampling to approximate the area light's illumination. With jittered sampling, we divide the area light's area in `lightSamples` $*$ `lightSamples` equally sized rectangles and take a random point within each rectangle. So when we want to ray trace an area light, we take a total of `lightSamples` $*$ `lightSamples` samples from the area light and treat each sample exactly like we would treat a spot light, except this time we divide their color by the number of samples.

Lastly, in the case it does not render an image but traces rays for the visualization and we use 25 or more samples, we check if all rays are of the same type (light or shadow) and if so, combine them to a single 'area-ray', see Figure 4.2. This reduces the number of rays significantly and improves performance. This also means that we had to change the `RayObjectPool` to support two different types of ray objects. As this area-ray is a big object, it is always transparent.

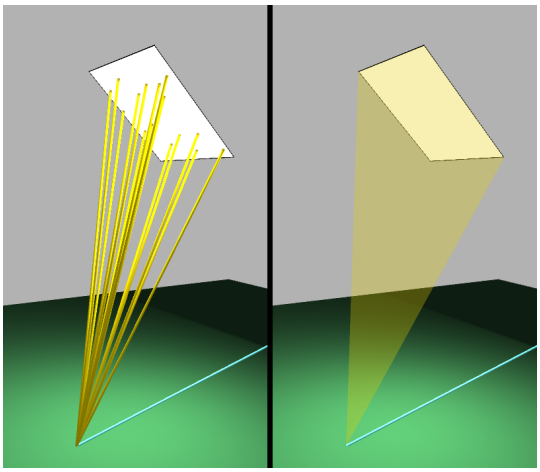


Figure 4.2: An area light sampled using 16 samples visualizes every ray (left) and an area light with 25 or more samples shows only a single 'area-ray' (right).

One thing to note is that the area light does not actually have a spot angle of 180° in VRT, but 175°. This is because Unity's default rendering pipeline cannot correctly render shadows at the edges of Unity's spot lights that have the spot angle at 180°(see Figure 4.3). There is a setting that allows the shadows to appear, but this creates other visually distracting artifacts, so we opted to set the area light's 'spot angle' to 175°.

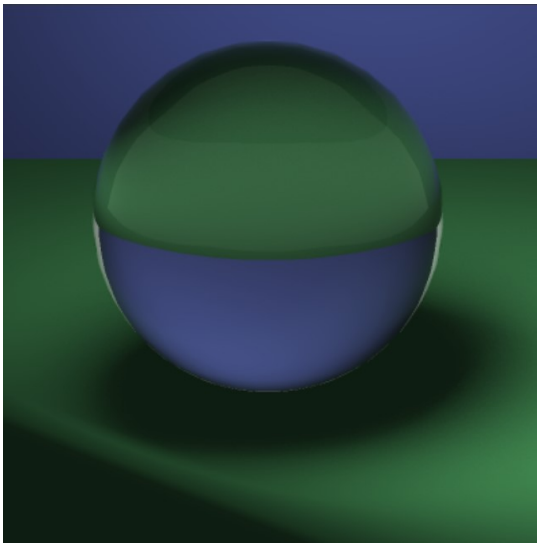


Figure 4.3: Unity not showing a shadow at the edge of an area light's reach when Unity's spot lights have an angle of 180°
Note: This image is taken in a development stage and this effect does not occur in VRT as the spot angle is set to 175°.

Spot Light

For the spot light we simply combined features from the point and area light. The image that depicts the spotlight could be rotated around one axis, as it is a point and not an area. For this, we used a similar approach as the existing point light. As Unity has a real-time spot light, like we used for the area light, everything worked on the side of Unity

automatically. As for the rest of the code, the area light already contained a direction and angle, so this also worked directly after only a few tweaks. To make it work with the ray tracer, we simply had to combine the point and area light's approach again. We trace it just like a point light, except we also check if the point is within the spotlight's reach.

For the same reason the area light's spot angle is set to 175°, we limited the range of the spot angle from 1° to 170°.

Light-Distance Attenuation

Light decreases quadratically in intensity as it travels further. We can easily implement this by dividing the color of a ray by the light-point distance squared. However, as this effect is quite dramatic and to prevent dividing by zero, often a formula like $a + bd + cd^2$ is used, where a, b and c are constants and d is the distance. In this program we opted to use $0.04 + 0.1d + 0.06d^2$. Light objects have their own properties panel that the user can access by selecting the light, and in order to better visualize the difference this makes, we added a toggle to this panel to enable/disable light-distance attenuation per light.

Angle Attenuation

Angle attenuation only applies to the spot and area light. It means that if these lights do not look directly at a point, but under an angle, the intensity also drops off. A common way of using angle attenuation is to multiply the color by $\cos(\alpha)^p$, where α is the angle between the light's direction and the direction to the point from the light and p is a non-negative number. We used this approach and gave the user an extra slider for spot and area lights to modify p.

Light Intensity

The two attenuation methods decrease a light's intensity, but there was no option to increase it. Increasing the intensity would mean simply multiplying the final color. We added an additional slider in the light object properties panel with which the user can adjust the light's intensity.

A complete overview of all the light properties can be found in Figure 4.4

4.2.2 Visualizing super-sampling

Super-sampling was already implemented in the ray tracer for when the user wanted to render an image. We only had to copy that to the ray tracer's function that also returns the traced rays. Visualizing was not trivial as the code expected a single tree of rays per pixel. To fix that, we changed the code to expect a single 'base-ray' first that should only be used for the pixel's color. We could then intentionally skip this ray for visualizing. Then we can give it multiple children, one for each sample, that will then simultaneously be visualized as they all come from the same parent. This approach also makes sure that when the user selects a single pixel, which makes the program only show the ray(s) associated with this pixel, all rays used for super-sampling in that pixel are shown.

The program already contained a ray tracer properties panel with settings related to the ray tracer and the visualization, see Figure 4.5. We added a toggle to these

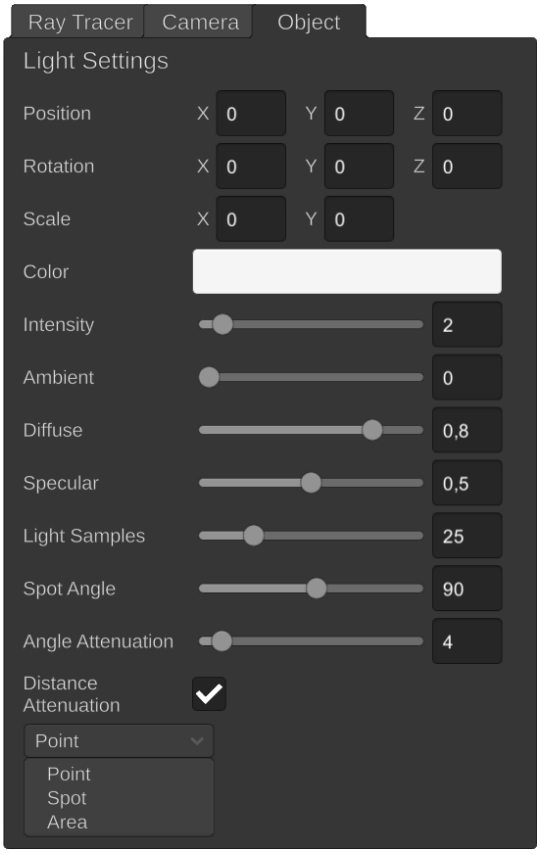


Figure 4.4: A screenshot of all possible light properties in VRT. Some properties are not visible or available for certain light types.

properties so the user can decide for themselves whether or not they want to visualize super-sampling.

4.2.3 Ray visualization

In the program, if the user decides to visualize super-sampling and the scene also contains (an) area light(s), there may be too many rays. As previously mentioned, to make this better visible for the user and to enhance the visualization in general, we added a few extra options.

Ray Contribution

One of the new `RayObject` properties needed for the new visualization options is the contribution the ray has to the final pixel color. This cannot be fully determined while tracing the rays as each ray's relative contribution to the parent depends on the rest of the parent's child rays. The parent's color depends on the material interaction at the hit-point plus any color returned by child rays. So once a parent ray is done tracing all child rays, we can set the contribution for all child rays with respect to the parent. This is important because if each ray knows its contribution with respect to its direct parent, we can recursively determine its contribution to the final pixel color. For example, if a ray contributes 25% to the pixel's color and it has two child rays that contribute 40% and 20% to this ray (assuming 40% of the color is from the material interaction at the

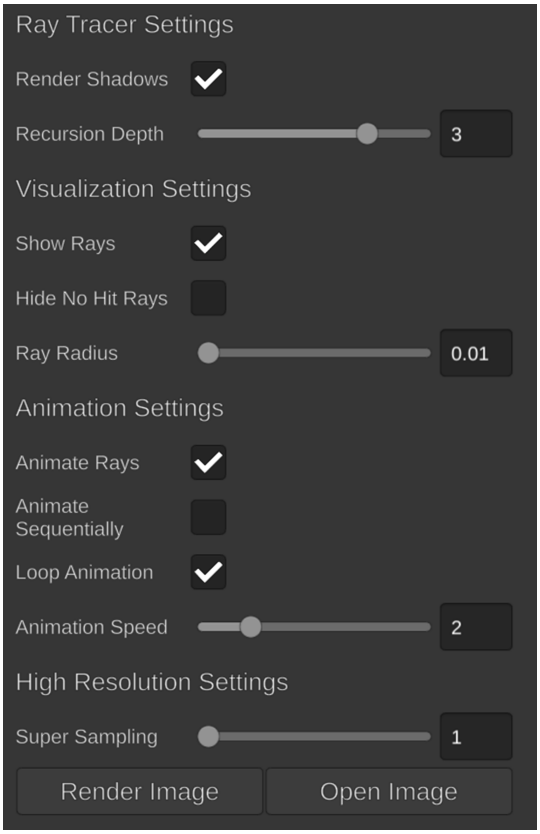


Figure 4.5: A screenshot of all preexisting ray tracer properties in VRT.

hit-point), then their contribution to the pixel's color is 10% and 5% respectively. This means that after we traced the rays, we recursively multiply each ray's contribution with that of its parent and then all rays have a contribution value that we can use for the visualization.

Hiding Negligible Rays

An intuitive step to declutter the scene of many rays is to not show negligible rays. We give the user two extra controls: a toggle to hide negligible rays and a slider to set the threshold that determines when a ray is negligible. Any ray with a contribution less or equal to this threshold will not be drawn.

Ray Transparency

In order to make rays transparent, they need new materials that are transparent instead of opaque. This is more computationally expensive than opaque rays. As transparent rays are harder to see, we remove all components except the ambient component of the transparent materials, as diffuse and specular reflections are not that noticeable. This makes them actually easier to render for Unity compared to opaque rays. We add two options to the ray tracer properties panel with which the user can control ray transparency: a toggle to enable ray transparency and a slider to set the level of transparency. If ray transparency is enabled, the `RayManager` will assign transparent materials to the rays instead of opaque materials. Every ray gets a unique material as the level of transparency depends on both the ray's contribution to the pixel's color and how transparent the user

wishes the rays to be.

Dynamic Ray Radius

Currently, each ray's radius is set to the `rayRadius` variable in the `RayManager`. We can instead make a function that returns the desired radius for the ray and make it depend on the ray's contribution. We give the user three extra controls: a toggle to enable dynamic ray radius and two sliders for the minimal and maximal ray radius. If the user wishes the ray's radius to be dynamic, the aforementioned function returns a radius between the minimal and maximal radius based on the ray's contribution. Else it returns the `rayRadius`. The way this is implemented gives the user another unique way of visualizing the rays, as the minimal ray radius does not have to be smaller than the maximal radius. If this is not the case, negligible rays are simply bigger than important rays, which gives the user the ability to easily find rays that almost contribute nothing.

Contribution-Based Ray Color

The ray tracer already stores the color that each ray contributes to the pixel in each `RayObject`. Just like the `RayManager` can return a unique transparent material for each ray, it can also give a uniquely colored material for each ray, optionally also transparent. We add yet another toggle to the ray tracer properties panel to enable contribution-based ray colors. If this toggle is enabled, the `RayManager` will not return a material based on the `RayObject`'s type, but on the color it contributes to the pixel.

All different visualization options are shown in Figure 4.6.

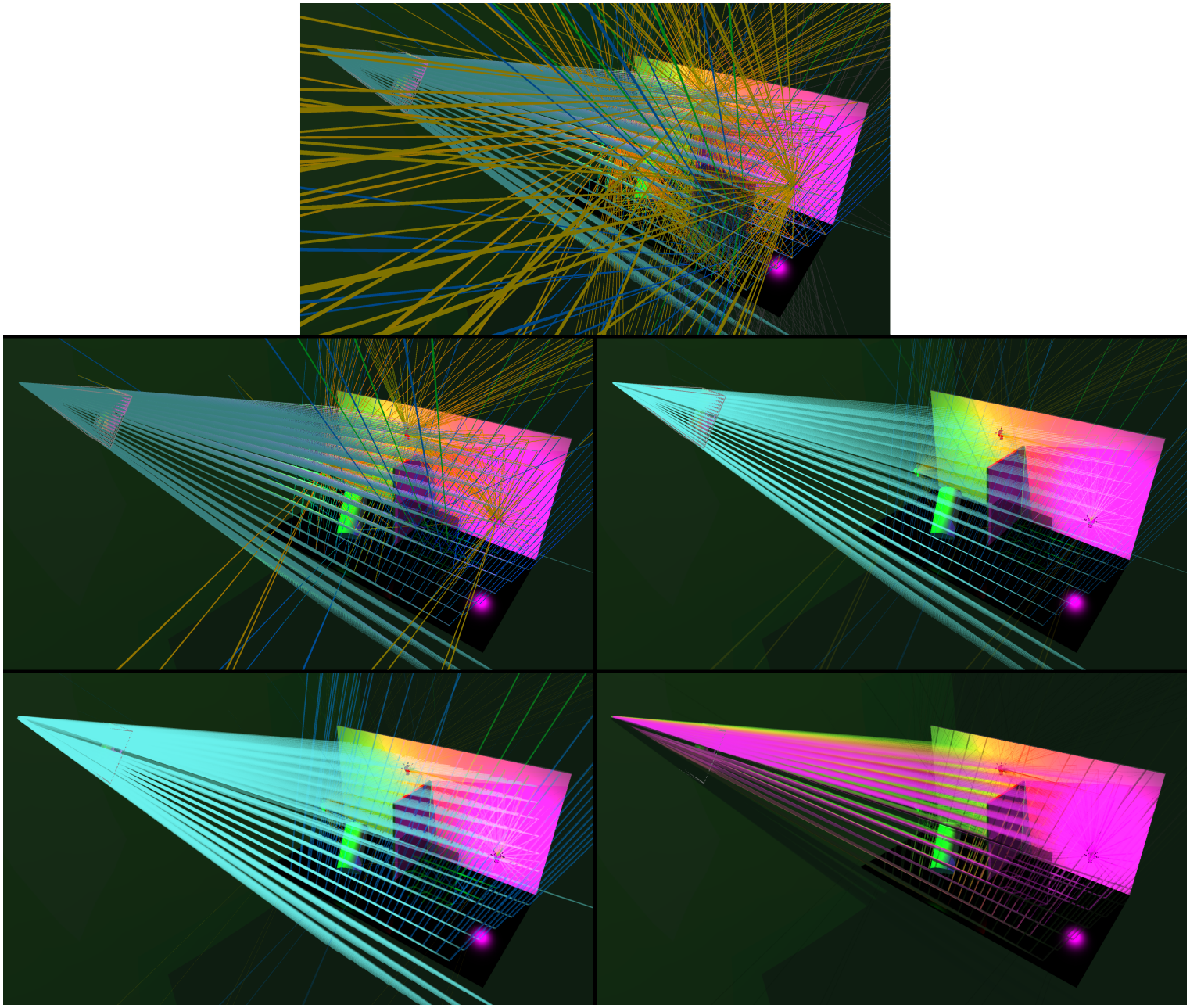


Figure 4.6: All ray visualizations options in VRT. From left to right, top to bottom, each time an extra option is enabled: all disabled, hide negligible rays, ray transparency, dynamic ray radius, contribution-based ray color.

4.2.4 Extra Controls

Lastly, we also added some extra controls. We added toggles for each type of light that disables those types of lights completely. This allows the user to experiment, for example, with different light types at the same positions. We also added a button that 'flies' the user to the virtual camera that the ray tracer uses. This allows the user to compare the rasterized scene with the ray traced scene. All the extra controls added to the ray tracer properties panel are shown in Figure 4.7.

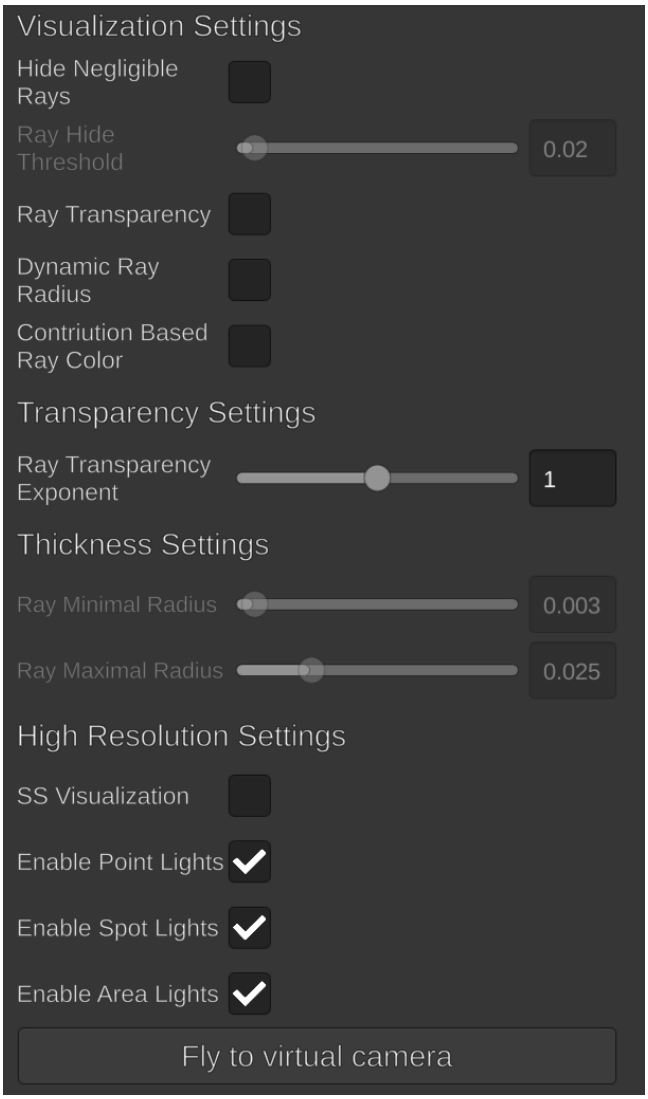


Figure 4.7: A screenshot of all new ray tracer properties in VRT.

And as rendering a scene with an area light and super-sampling can take long, we also added a progress bar to the render screen so the user can estimate how long they have to wait before the render is done, shown in Figure 4.8. We only did the implementation of the progress bar for the ray tracer. The progress bar itself was made and designed by P.J.T. Blok for his thesis about Gamification in VRT [4].

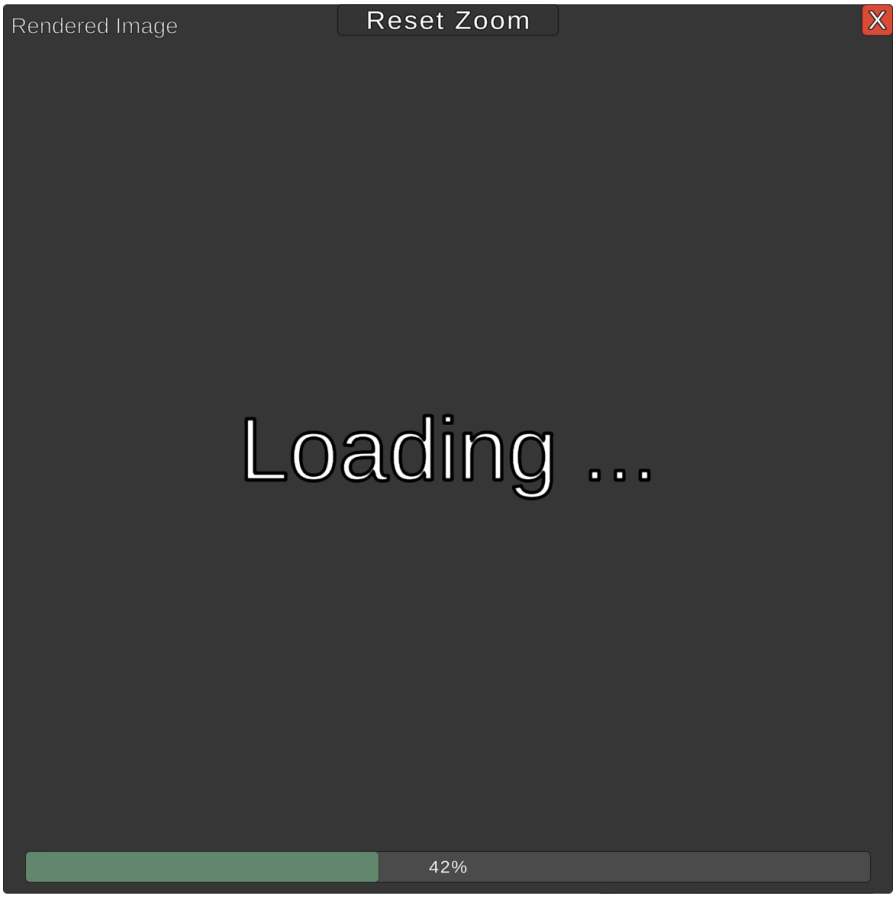


Figure 4.8: The loading screen in the render windows of VRT with a progress bar.

4.3 New Levels and Gamification

The existing ray tracer used a big pop-up message at the start of each level to explain concepts. This is not the most fun way of learning ray tracing. For that reason, alongside this project, a gamified version of VRT was made by P.J.T. Blok [4]. As my version included more features and concepts that needed to be explained, we added more levels to explain everything. This meant that the user had to go through even more long texts which has downsides [4]. For that reason, besides adding new levels, I merged my project with the gamified version to enhance the learning experience for the user.

Chapter 5

User Study

In this chapter, we go over how we tested if users actually learned the newly added and visualized concepts with a user study. The user study itself can be found in Appendix A.

5.1 Questions

The goal of the user study is to find out if users also understand the new concepts added and explained in VRT and to see if the extra ray visualization features offer any help. We were hoping to get at least around 20 participants. Luckily, the VRT program no longer needs to be installed but also works in the browser on a desktop computer. This made it easier to spread the survey around. We asked participants to spend around 20 minutes with the program, as they first need to learn about the already implemented basic ray tracing aspects before they move on to the new concepts and features.

Context Questions

First, we wanted to know a little about the participant so we can put the answers in context. For example, we would expect a participant with a master's degree in Computing Science to understand ray tracing and the program itself better than someone who is in high school that has little interest in computer technology. Therefore, we asked the user what their highest level of education is (including current studies), how skillful they consider themselves to be with computers on a scale of 1 to 10, how familiar they already are with ray tracing on a scale of 1 to 5, how many minutes they actually spend with the program, and optionally, their age.

Educative Questions

We wanted to know if the user learned the new ray tracing concepts explained in the tool and whether the extra visualization possibilities offered any help. As the user already had to spend quite some time with the program, we wanted to keep the user study concise and we chose to go with eight questions. Six questions asked the user to indicate on a scale of 1 to 5 how much the tool helped with understanding a certain concept, with two open questions where the participant was asked to explain how the tool did (not) help.

Additional feedback

Lastly, we asked the participant what they thought of the complexity of the program and if they had any other additional feedback or general or technical remarks. It is good to know if the participant could not successfully complete the (mandatory) tasks[1] in each level of the program due to technical issues, or if they had ideas to further improve the program.

5.2 Results

The user study was conducted in three groups. The first group, gathered from friends, consisted of 6 participants. The second group of participants was gathered from SurveySwap[2]. Unfortunately, not all of the responses could be used as some did not follow the instructions appropriately, some did not fill out the survey correctly or gave made-up answers, and lastly, some said they did not learn anything as they already knew everything. After validating the responses, we were still left with a total of 55 responses. We discuss these two groups as one as they do not differ much, both demographically and in responses. The last group consisted of 31 high school students that have IT as a subject in their programme. We chose to also test if they could understand these concepts as they may be a little complex, but do not actually require any knowledge of Computer Science courses taught ahead of the Computer Graphics course, which the main target demographic, Computer Graphics students, do have. We again needed to filter the responses, for the same reasons as mentioned before.

The results of the multiple-choice questions can be found in Figure 5.1.

Friends and SurveySwap participants

Almost all participants had little to no preexisting knowledge about ray tracing, which is good because it means that we can test if the program successfully teaches them ray tracing from the ground up. On average, they spent around 22 minutes with the program. The tool helped participants understand ray tracing quite well, with some exceptions. The difference between the types of light sources was understood pretty well, whilst the light attenuation was harder to understand. Soft shadows were understood quite well as well and nearly all users found the new ray visualization options nice to have, especially the ray transparency and contribution-based ray color.

Looking into individual answers, there does not seem to be much difference in understanding a certain concept compared to another concept. Either the participant understood all aspects of the program, or they did not understand any of them, with the exception being the ray visualization options, which were generally liked.

The main complaints were that it was 'too complex/complicated' and 'laggy'. The latter can occur on some lower-end systems which are likely to obstruct the learning process. The complexity, however, was not only about the explanations but also about the program itself. Some participants did not seem to understand what was expected when

[1] In the gamified VRT [4], users cannot directly go through the levels, they first have to complete tasks. Some of these tasks are mandatory, some are optional. Once the mandatory tasks have been completed, they can proceed to the next level.

[2] `https://surveyswap.io/` is an online service where users can fill out surveys and in turn, the user will also get responses on their own survey(s)

the program prompts them to perform some task, some found it simply too difficult to understand. None of the users that had problems understanding ray tracing had followed Computing Science-related studies. From the results, we can also see that if the user spends more time with the program, they are more likely to understand everything. Younger users (<25 years old) also had fewer complaints than older users.

In general, from this group, we can conclude that this program is not usable to teach (distributed) ray tracing to the average person but expects the user to have some affinity with Computing Science. But if the user is able to work with the program, they are likely to quickly understand advanced ray tracing. Users that had a background in Computing Science had no problems using the program and understanding the concepts.

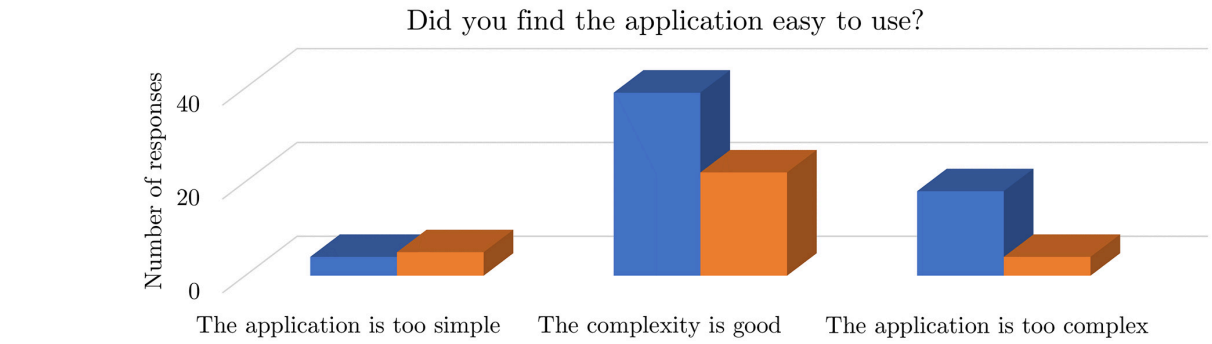
High School Participants

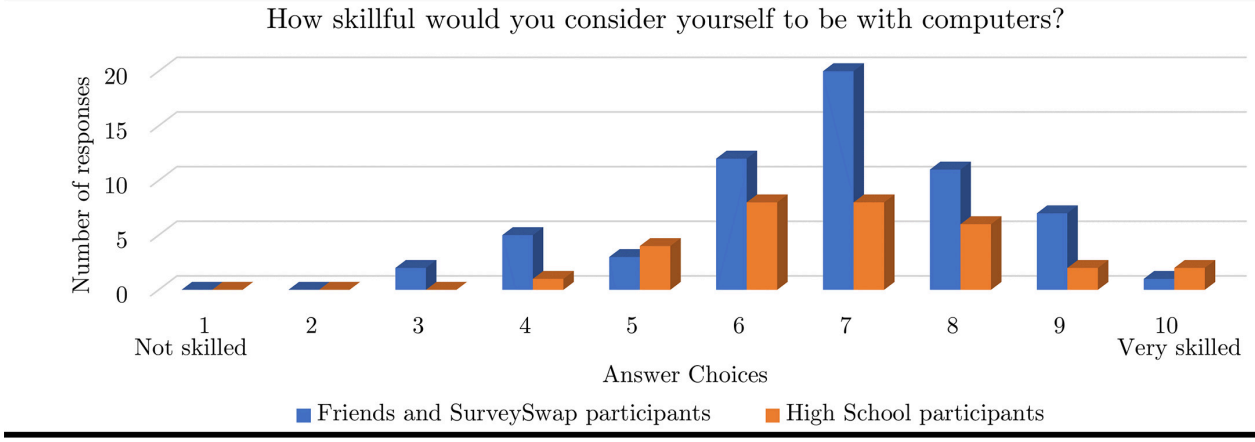
The high school participants from the Ubbo Emmius high school in Stadskanaal, Netherlands, got the same survey, except the questions were translated into Dutch. The tool and the mentioned concepts in the survey remained in English. All of the students have the IT subject in their programme. Important to note is that not all of these students have an affinity for computers, some choose this subject to get more familiar with computers. Hence we do expect some users to have trouble understanding these concepts.

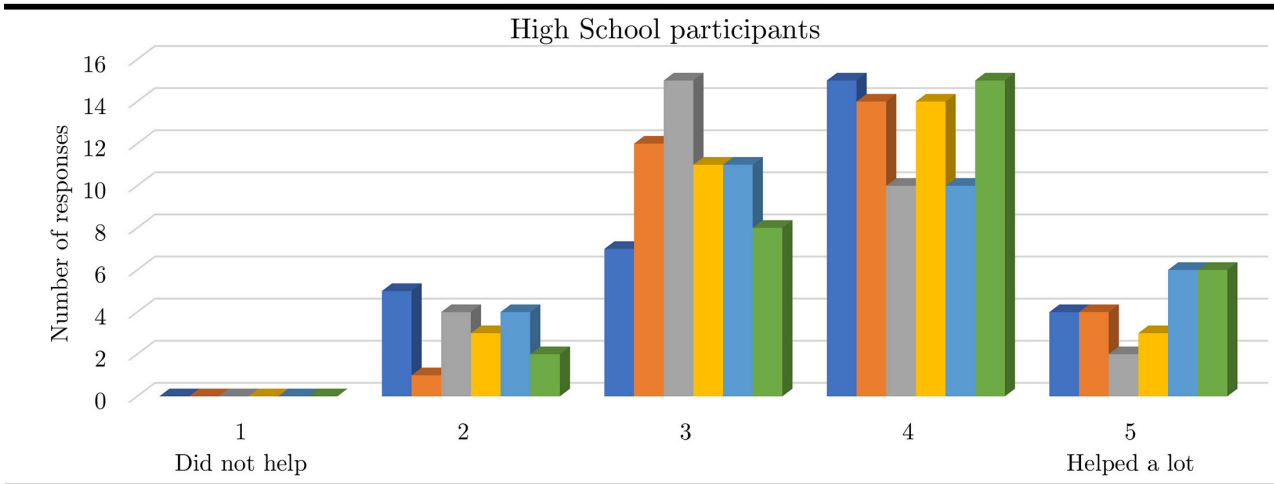
Overall, the results are almost on par with the 'Friends and SurveySwap' participants. On average, they spent around 17 minutes with the program. The tool helped the students understand ray tracing quite well, although some admitted they had a bit of a struggle understanding both the language and the terminology. Nevertheless, none of the participants thought the program did not help at all in certain aspects, it at the very least taught them something in the limited time that they used the tool.

Once again, the individual answers revealed that all concepts were always understood quite evenly. For example, if a student did not understand a certain aspect, they did not understand other concepts well either, and the ray visualization options were again appreciated very much. The complaints were also similar to that of the previous group, mostly that it is sometimes a bit complex and that the program does not run smoothly on all computers.

The age of this group is significantly lower but the responses are quite similar compared to the previous group. This goes to show that if the user has some interest and intuition on how a program like this would work, they can learn it very well. It also shows that if users would like to learn about ray tracing, but do not have any experience with programs that have similar features as VRT (i.e. 3D software and games), they find it more difficult to learn as they struggle with using the program itself.







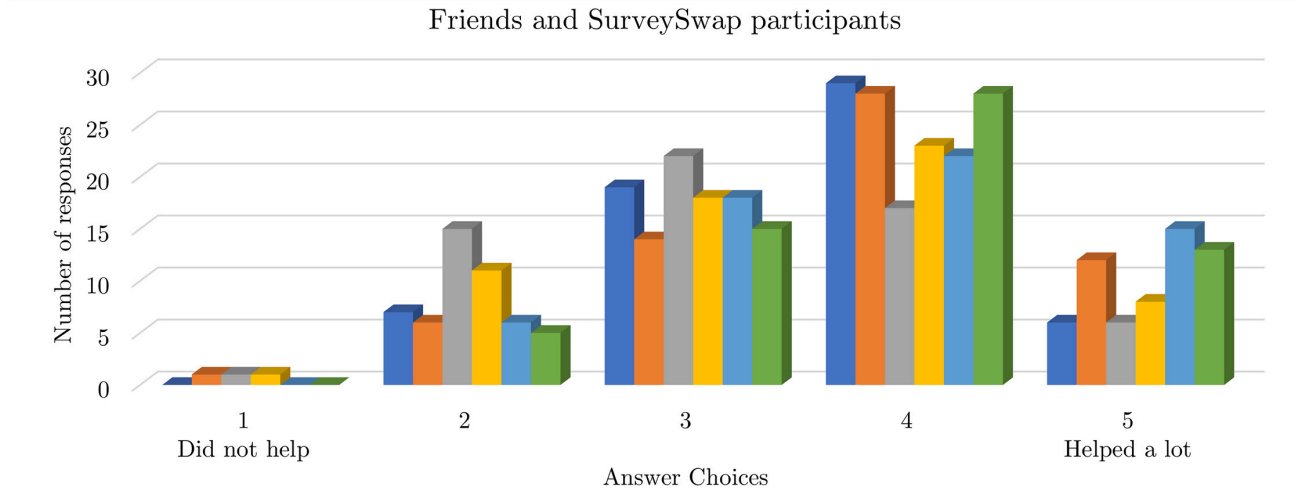


Figure 5.1: A summary displaying the responses for the multiple-choice questions of the user study.

Chapter 6

Conclusion

This thesis had the objective to improve on the original Virtual Ray Tracer application by extending it with distributed ray tracing features. We implemented and visualized two distributed ray tracing concepts, super-sampling and soft shadows, and also extended the program with additional features to aid the user with the learning experience and to make the program more complete. These extra features include a spot light, light-distance and angle attenuation, new ways of visualizing rays, and some miscellaneous features.

From the user study, we can conclude that this extended version of the gamified [4] Virtual Ray Tracer generally helps users to understand the new features. It also shows that more than just Computer Graphics students can learn ray tracing from this application, but that definitely some basic knowledge and interest are required.

For users with a Computing Science background, the program was very usable and the concepts were easy to understand. For other users, it fluctuated. Some users were able to use the tool and were surprised by how much they learned about ray tracing in just around 20 minutes. Others were not so lucky. The two main obstacles were the information and how it was explained, and how to interact with the program. For some people, working in a 3D environment like VRT is nothing special, others already struggled there. However, the program and its controls may be an obstacle for some users, most of them are not the target demographic (Computer Graphics students). So it could be argued that this is to be expected and not worthwhile to improve on, as the program does work for the intended audience.

Chapter 7

Future Work

In this chapter, we discuss potential improvements to the gamified VRT with distributed ray tracing features.

Program Interaction

One of the main complaints from people that struggled to learn ray tracing using VRT was the program itself and how to interact with it. By making the program more accessible for users that never worked with other programs that have similar features, the tool might help even more people. Whether this is worth the work would depend on if those users benefit from learning ray tracing.

Extra Explanations

Most concepts are explained in a simple and easy way to make them straightforward to learn. However, sometimes this is not enough for certain users, and others find it too simple and like to have a more in-depth explanation. Adding the ability to quickly access either a more extensive or more in-depth explanation could benefit some users.

Performance

VRT is not optimized well. Especially with area lights, the performance on a mid-tier laptop might be so poor that the user is limited in learning from VRT because they cannot run it. This could be improved on by further optimizing the code and by changing the quality settings in Unity, which is currently static on 'Ultra'. The performance can also be enhanced by changing the way rays are rendered in Unity, as is discussed in the next point.

Ray visualization

The transparent rays in VRT currently have only an ambient component, meaning they do not interact with any light sources. This is the reason that enabling ray transparency may significantly improve the performance of VRT. This does, however, have the downside that it is harder to see how far away rays are, as they have no shadow or illumination. Finding a balance between having the rays look as nice as possible and the rays not costing any computational power to render, could improve both.

Visual Guidance

Some users mentioned that some visual guidance might be helpful when explaining certain concepts. Part of understanding the concepts is being able to play with them and if they do not understand how to navigate through the program, it is quite hard. An example would be a flashing circle around a button the user has to click.

Controls

The controls could be simplified to make it easier for the user to interact with the program. Some mice (Mac-mouse and laptop track-pads) do not have a (clickable) scroll wheel for example, and some users are not used to controlling a 3D application.

Non-rectangular area light

Not all area lights are rectangular. Adding circular and potentially other types of area lights might be nice for the user to play around with.

Color Banding

One of the phenomena currently in VRT that does not look real is color banding. Color banding is the effect caused by colors rounded to the nearest integers RGB-values. Especially when the colors are darker, these jumps can be noticeable, as can be seen in Figure 7.1. A possible fix to this is dithering. Dithering is intentionally applying noise to the image in order to randomize the 'quantization error', which is the round-off error. I.e. when we round 2.8 to the value 3, the quantization error is 0.2. Without dithering, the colors would, for example, first be rounded down first (quantization error of -0.5) and the further we move it would be rounded up (quantization error of 0.5). So the quantization error is linearly proportional to the color change. By applying noise and therefore randomizing this quantization error, the color change will not be linear and the color banding effect will decrease, albeit at the cost of some noise.

(a) Dark color banding. (b) Bright color banding.

Figure 7.1: Color banding in VRT.

Acknowledgements

I would like to thank my supervisors prof. dr. Jiří Kosinka and dr. Steffen Frey for making time to help with this project by answering questions and giving valuable feedback. I would also like to thank Chris van Wezel for introducing Virtual Ray Tracer and giving extra information and support. I also want to thank Peter Jan Blok for making the gamified VRT, which I used in my project. At last, I would like to thank my friends, the users of SurveySwap and Frank van het Hof from the Ubbo Emmius high school and his students for distributing and taking part in the user study, respectively.

Bibliography

[1] C. van Wezel, "A virtual ray tracer," Bachelor's Thesis, University of Groningen, 2022. [Online]. Available: http://fse.studenttheses.ub.rug.nl/id/eprint/26455.

[2] W. Verschoore de la Houssaije, "A virtual ray tracer," Bachelor's Thesis, University of Groningen, 2022. [Online]. Available: http://fse.studenttheses.ub.rug.nl/id/eprint/24859.

[3] W. A. Verschoore de la Houssaije, C. S. v. Wezel, S. Frey, and J. Kosinka, "Virtual Ray Tracer," in *Eurographics 2022 - Education Papers*, J.-J. Bourdin and E. Paquette, Eds., The Eurographics Association, 2022, ISBN: 978-3-03868-170-0. DOI: 10.2312/eged.20221045.

[4] P. Blok, "Gamification of virtual ray tracer," Bachelor's Thesis, University of Groningen, 2022. [Online]. Available: http://fse.studenttheses.ub.rug.nl/id/eprint/27596.

[5] P. Shirley and S. Marschner, *Fundamentals of Computer Graphics*, 3rd. USA: A. K. Peters, Ltd., 2009, ISBN: 1568814690.

[6] R. L. Cook, T. Porter, and L. Carpenter, "Distributed ray tracing," *SIGGRAPH Comput. Graph.*, vol. 18, no. 3, pp. 137–145, Jan. 1984, ISSN: 0097-8930. DOI: 10.1145/964965.808590.

[7] N. Vitsas, A. Gkaravelis, A.-A. Vasilakis, K. Vardis, and G. Papaioannou, "Rayground: An Online Educational Tool for Ray Tracing," in *Eurographics 2020 - Education Papers*, M. Romero and B. Sousa Santos, Eds., The Eurographics Association, 2020, ISBN: 978-3-03868-102-1. DOI: 10.2312/eged.20201027.

[8] C. Gribble, J. Fisher, D. Eby, E. Quigley, and G. Ludwig, "Ray tracing visualization toolkit," in *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ser. I3D '12, Costa Mesa, California: Association for Computing Machinery, 2012, pp. 71–78, ISBN: 9781450311946. DOI: 10.1145/2159616.2159628.

Appendix A

User Study Questions

Q1: What is your highest level of education? (Including current studies)

Q2: How skillful would you consider yourself to be with computers?
R: 1–10, with 1 1 'Not skilled' and 10 'Very skilled'.

Q3: How familiar were you with ray tracing before this survey?
R: 1–5, with 1 'Not familiar' and 5 'Very familiar'.

Q4: How many minutes did you spend with the program?

Q5: How well did the tool help you with understanding ray tracing (better)?
R: 1–5, with 1 'Did not help' and 5 'Helped a lot'.

Q6: How well did the tool help you with understanding the different types of light-sources?
R: 1–5, with 1 'Did not help' and 5 'Helped a lot'.

Q7: How well did the tool help you with understanding how 'Angle Attenuation' and 'Light-Distance Attenuation' works?
R: 1–5, with 1 'Did not help' and 5 'Helped a lot'.

Q8: How well did the tool help you with understanding how soft shadows are created?
R: 1–5, with 1 'Did not help' and 5 'Helped a lot'.

Q9: How well did the tool help you with understanding how super-sampling works?
R: 1–5, with 1 'Did not help' and 5 'Helped a lot'.

Q10: Please explain why you did (not) understand the ray tracing aspects explained in the tool.

Q11: Do you think the different possibilities of visualizing rays helps?
R: 1–5, with 1 'Not at all' and 5 'Very much'.

Q12: You can leave feedback on the ray visualization here.

Q13: Did you find the application easy to use?

R: 'The application is too simple. More controls and settings would be an improvement', 'The complexity is good', 'The application is too complex. There are too many unnecessary settings and controls'.

Q14: If you have any other feedback, general or technical remarks, you may leave them here (optional)

Q15: What is your age? (optional)