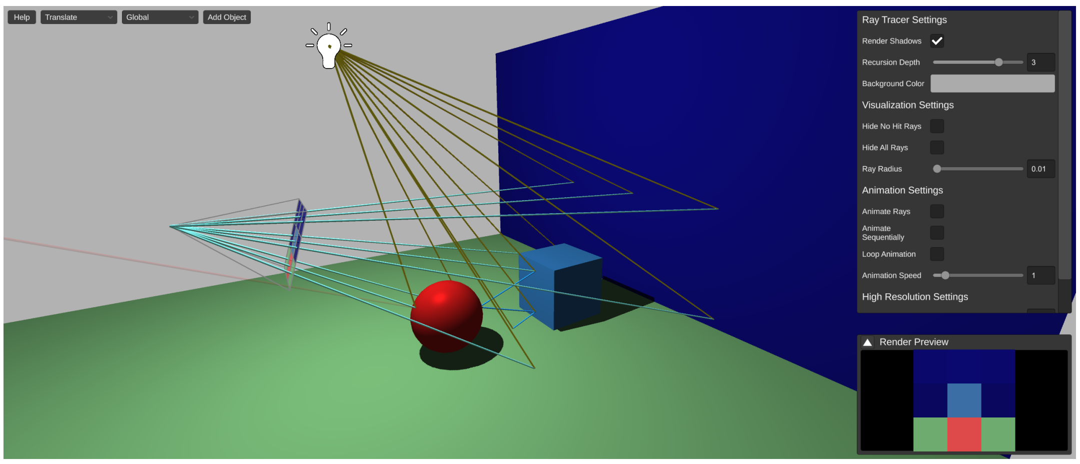
Virtual Ray Tracer

W. A. Verschoore de la Houssaije, C. S. van Wezel, S. Frey, and J. Kosinka

Bernoulli Institute, University of Groningen



Ray Tracer Settings
Render Shadows ✓
Recursion Depth 3
Background Color
Visualization Settings
Hide No Hit Rays
Hide All Rays
Ray Radius 0.01
Animation Settings
Animate Rays
Animate Sequentially
Loop Animation
Animation Speed 1
High Resolution Settings

Figure 1: *A screenshot of Virtual Ray Tracer showing a scene with UI elements enabled. The scene contains a virtual camera and screen with* 3×3 *pixels, and a reflective sphere and cube. Primary and reflection rays are shown in green, shadow rays in orange.*

Abstract

Ray tracing is one of the more complicated techniques commonly taught in (introductory) Computer Graphics courses. Visualizations can help with understanding complex ray paths and interactions, but currently there are no openly accessible applications that focus on education. We present Virtual Ray Tracer*, an interactive application that allows students/users to view and explore the ray tracing process in real-time. The application shows a scene containing a camera casting rays which interact with objects in the scene. Users are able to modify and explore ray properties such as their animation speed, the number of rays as well as the material properties of the objects in the scene. The goal of the application is to help the users—students of Computer Graphics and the general public—to better understand the ray tracing process and its characteristics. To invite users to learn and explore, various explanations and scenes are provided by the application at different levels of complexity. A user study showed the effectiveness of Virtual Ray Tracer in supporting the understanding and teaching of ray tracing. Our educational tool is built with the cross-platform engine Unity, and we make it fully available to be extended and/or adjusted to fit the requirements of courses at other institutions or of educational tutorials.*

CCS Concepts

• ***Social and professional topics*** $\rightarrow$ ***Computer science education;*** • ***Computing methodologies*** $\rightarrow$ ***Ray tracing;***

1. Introduction

Ray tracing [HAM19,MSW21] is an important rendering technique in Computer Graphics. It is capable of producing realistic images and animations, albeit at a high computational cost. In real-time rendering applications, where performance is vital, ray tracing is often too slow and other rendering techniques such as rasterization [MS21] are used. While these techniques are fast, they often produce less convincing results. Because of this, ray tracing has seen much use in offline rendering applications such as animated films, but recent advances in graphics hardware are also making ray tracing suitable for real-time rendering applications [DNL*17].

DOI: 10.2312/eged.20221045

Due to its prevalence and importance, ray tracing is widely taught in Computer Graphics courses. While the core idea of ray tracing is simple, more advanced ray tracing techniques can become quite complicated. To aid in the understanding of these techniques it is often useful to visualize them. Simple illustrations are frequently employed[†], but they are 2-dimensional and static, which limits their effectiveness as an educational tool. To address this issue, we have developed *Virtual Ray Tracer*, an interactive application that visualizes the ray tracing process. Our aim was to develop a suitable 3D way to visualize ray tracing and to evaluate the effect the application has on the learning process. The application has the potential to help computer graphics students understand ray tracing faster and better than they would without interactive visualizations.

The main demographic for the application is the students following the Bachelor-level Computer Graphics course at the University of Groningen. The application will be used in future iterations of the course to help teach students about ray tracing. Nevertheless, the application is built to be accessible to as wide an audience as possible, so anyone interested in ray tracing is enabled to understand and explore its underlying concepts.

We start by reviewing related work (Section 2). In Section 3 we discuss the general design of the application while Section 4 covers the implementation details. In Section 5 we evaluate the results of the user study we conducted. We state our conclusions and discuss potential future work in Section 6.

2. Related Work

Our application aims to improve the learning process by providing a visualization of ray tracing techniques. While applications visualizing certain aspects and metrics of ray tracing exist [SHP*19, SAH*16, SJL15, ZAD15], they are not specifically aimed at education. The abundance of such applications does suggest that visualization is a useful tool in understanding ray tracing. Conversely, there are many applications aimed at teaching ray tracing, but they do not directly visualize the ray tracing process itself [SSM02, VGV*20].

Nevertheless, the idea for an educational application that visualizes ray tracing is not entirely unique. One of the first implementations comes in the form of a set of Java Applets developed in 1999 [Rus99]. Unsurprisingly, its age means that the application is rather simple by today's standards and thus unlikely to be useful for teaching ray tracing today. This is cemented by the fact that the application seems to be no longer available online. Its age and unavailability also mean that it is not possible to extend the application to meet modern standards of graphical fidelity and interactivity.

A more recent application similar to ours is the Ray Tracing Visualization Toolkit (rtVTK) [GFE*12]. As the name suggests, rtVTK is a toolkit for the visualization of ray tracing. The main goals of rtVTK are to aid in the development of ray tracing applications and to help with ray tracing education. However, the authors themselves admit that rtVTK may be too complicated for the latter. The main issue is that rtVTK requires users to hook up the toolkit with their own ray tracing application. While this approach means the toolkit can be used in nearly any ray tracing application, it is not exactly trivial. For an educational application meant to be accessible to as many people as possible this is not ideal.

[†] E.g. the Wikipedia illustration at `https://commons.wikimedia.org/wiki/File:Ray_trace_diagram.svg#file`

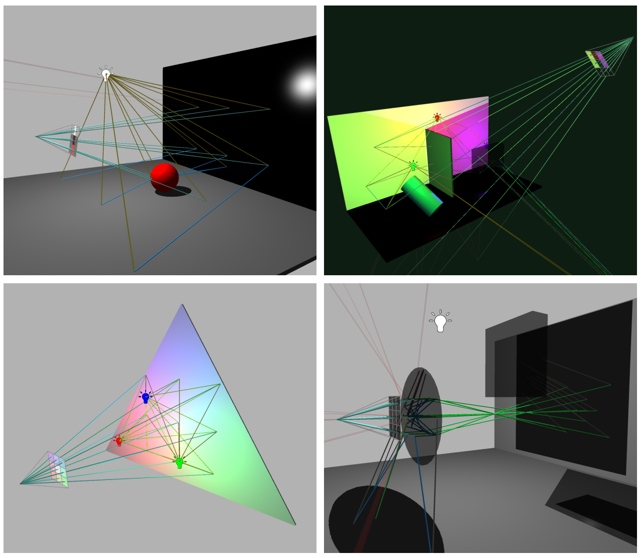


Figure 2: *Several examples of preset and customised scenes in Virtual Ray Tracer. Each scene comes with a virtual customizable camera and screen. A scene can contain an arbitrary number of shapes and light sources with adjustable attributes (material/color, ambient/diffuse/specular coefficients, etc.). The traced rays, color-coded based on their type (primary, shadow, etc.), are shown as well.*

Consequently, there is room for a dedicated and user friendly educational application that visualizes the ray tracing process.

3. The Application

The default setup for the application is a scene with some objects, lights and exactly one camera. Figure 1 shows a simple scene. Rays slowly shoot from this camera, one ray through each pixel of the camera's screen. When a ray intersects an object in the scene, a shadow, reflection and/or refraction ray may be traced from that intersection point. Each different type of ray has its own color. For example, reflection rays are blue while refraction rays are green. Several example scenes are shown in Figure 2.

The user can interact with the objects, lights and camera in the scene by clicking on one to select it. This opens a properties panel. Properties of the selected object changed by the used are immediately reflected in the scene visuals and the rays being traced. The results of the visible rays are displayed on the camera's screen and in the bottom left preview window. Again, this only shows the results of the rays being visualized, so it is very low resolution. The user can press the "Render" button in the general properties panel to open a high resolution ray traced rendering.

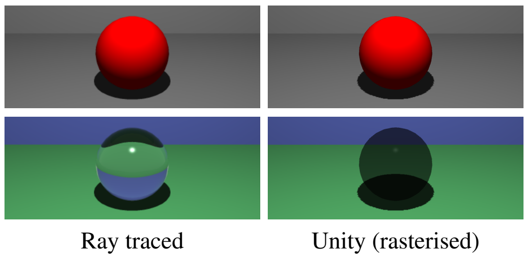


Figure 3: *A comparison of different materials in the Unity scene and in the ray traced render. Diffuse materials look nearly identical (top row), but transparent materials are only roughly approximated in Unity (bottom row).*

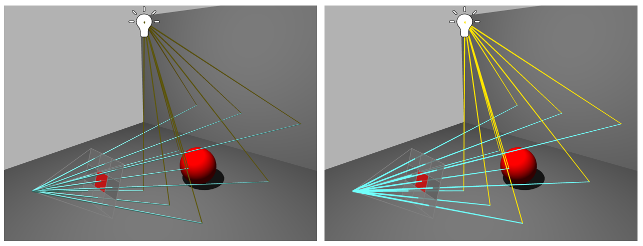


Figure 4: *A comparison of lit and unlit ray materials. Left: Rays affected by scene lights (our default). The rays look 3D and their positions are easily determined. Right: Rays unaffected by lights. The rays look 2D and their positions are difficult to determine.*

3.1. Visuals

The core of the application is the visualization of ray tracing. This comprises the ray-traced scene from the point of view of the virtual camera (shown within the scene on the camera's screen and in a separate widget, see Figure 1), the scene itself from the user's point of view (Section 3.1.1), and most importantly the visualization of the rays traced in the scene (Section 3.1.2).

3.1.1. Scene Visuals

Ideally, the scene visuals should match the final ray traced image when viewed from the camera's viewpoint. At the same time, it is crucial in our context that the application remains responsive at all times even for high ray counts on current commodity hardware. We have therefore opted to render the scene visuals via rasterization, which is computationally much less demanding than current real-time ray tracing methods [MSW21]. This yields smooth and highly interactive operation on current commodity hardware—which students and the general public can be expected to have—while still conveying the visual appearance of the result. This intuitively allows users to immediately get an impression of the result of the setup within the scene without needing to view a separate window.

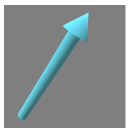
By employing a custom shader based on the same Phong illumination model [Pho75] used by the ray tracer, we can make diffuse reflections and specular highlights look nearly identical. The visuals based on tracing recursive rays such as reflections and refractions can generally not be replicated as easily; see Figure 3. The only exception is shadows, since Unity provides built in support for them.

An alternative would be not to try to match the ray tracer visuals, e.g. by rendering each object with the same uniform color, regardless of the material settings provided to the ray tracer. Then there would be no confusion about some materials not lining up with the ray traced image. However, this would make editing object materials much worse, because none of the changes would be reflected in the scene visuals. This is arguably more confusing than the slight differences between the rasterized and ray traced visuals.

3.1.2. Ray Visuals

In order to visualize ray tracing in an intuitive and appealing way, we need to carefully consider how we draw the rays. It needs to be clear where a ray is coming from, where it is going, and what kind of ray it is. All this combined with the fact that it should be possible to draw many rays at once makes for an interesting design and engineering challenge.

The core idea behind the ray design is simplicity. We want the user to be able to clearly see the rays, but we do not want to make the rest of the scene difficult to see. This becomes especially true when the number of visualized rays is large. Therefore, the rays should have a simple shape and material.



The simplest shape for a ray is a cylinder. We could argue that a cylindrical arrow (see the inset) is a better option because, while more complex, it also indicates the direction of the ray.

However, we believe that animating the rays is a more natural way of showing the direction of a ray, while keeping the actual ray object simple. What we mean by animation is that we gradually extend rays from their origin towards their end point. Once a ray has reached its full length, its child rays (such as reflection rays) start their animation. We do things this way as it clearly demonstrates the recursive nature of ray tracing (see the accompanying video).

We decided to shade the rays based on the scene's lighting conditions combined with an ambient color. This provides a better understanding of their position and orientation in the 3D scene. The ambient color was added to make the rays easily visible even when there is no lighting in the scene. To better distinguish between different kinds of rays, we color rays based on their type.

The difference between the two approaches is shown in Figure 4.

3.2. Settings and Controls

One of the most important features of the application is its interactivity. We want the user to be able to experiment with different settings and to learn ray tracing and its components that way. To this end, we allow users to change a wide variety of settings regarding the ray visualization and properties of objects in the scene, and to add or remove a predefined set of objects to/from the scene (see the accompanying video).

We aim to make our application as accessible and intuitive as

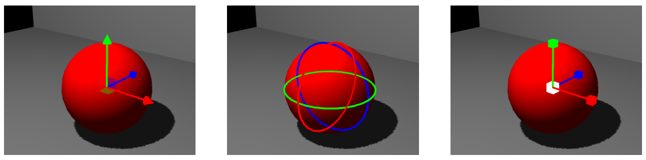


Figure 5: *Transformation gizmos. From left to right: Translation gizmo, rotation gizmo, and scale gizmo.*

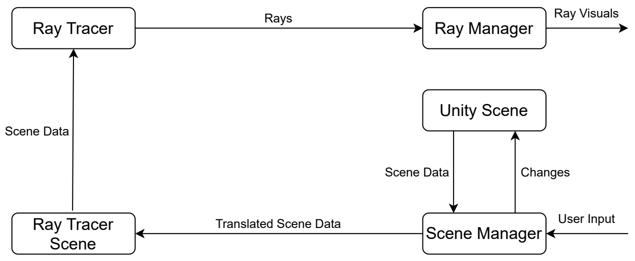


Figure 6: *The design for the general structure of the application.*

possible with our user interface (UI) design. Most of our UI components are on the right side of the screen. This means that the scene is always visible so that any changes made in the properties panel have an immediate and obvious effect. Another important concept is documentation. All elements in the properties panel have tooltips, and most aspects of the application are explained in the help panel. It is important that this information is easily accessible, but only visible on demand. Further, the properties panel's contents change based on the user's selection. For example, when the camera is selected, only its settings are displayed.

Nevertheless, the best UI is, arguably, no UI. If something can be done intuitively just through keyboard and mouse input, it is almost always better than designing UI components for it. A good example of this are camera controls as well as positioning, rotating and scaling objects in the scene. By placing shapes on a selected object that can be clicked and dragged, we can translate, rotate and scale the object in an intuitive way. This technique is commonly employed in the scene editors of game engines (such as Unity's editor). The shapes are often called transformation gizmos. Figure 5 shows what this looks like in our application. Because clicking and dragging may not always have the desired precision, we still have position, rotation and scale UI components in the properties panel, but in most cases the gizmos are a much more direct and intuitive way to transform an object.

4. Implementation

Our goal of developing an educational ray tracing tool leads to requirements which then inform the design of our implementation. Our approach needs to run fluently at all times even on older hardware, be available on many platforms, and be easily extensible. It is further required to handle scenes that are dynamically changing as a result of user interaction.

4.1. Program Structure

The application is implemented in Unity[‡]. Unity is a freely available game engine that can be used for 2D and 3D applications. It is widely used, well documented, and there are many resources online that explain and discuss its various components. This means that our application can be easily extended. Unity also provides build support for a wide range of platforms, which allows us to make the application widely available [vWV22].

Conceptually, the application consists of two core components: the ray tracer and the 3D Unity application. The ray tracer takes information about a scene and produces a list of rays traced in that scene from the camera. The Unity application visualizes the scene from a different perspective (that of the user in the interactive application), crucially also incorporating the rays generated and followed by the ray tracer.

In Unity a scene contains all information regarding the current state of the application, among others including UI elements and rays. Most of this information is irrelevant to the ray tracer: it only needs to know about the camera, the objects in the scene, and their materials. Another relevant point is that, as already mentioned, the Unity scene will change dynamically based on user input.

Our design to meet these requirements is illustrated in Figure 6. The Unity application takes input from the user and changes its internal scene. These changes then need to be sent to the ray tracer which uses its own, simpler scene representation (*Ray Tracer Scene* in Figure 6). For this, we include a translation layer that converts the Unity scene to the format of the ray tracer, which then outputs a list of rays for the Unity application to draw (via the *Ray Manager*).

Our interactivity and performance requirements mean that the translation from the Unity scene to the ray tracer needs to be fast. For this, we have implemented the ray tracer using Unity's built in ray casting utilities. These utilities work directly with Unity's internal scene representation, so the translation layer is comparably lightweight: it essentially comes down to filtering for the objects relevant to the ray tracer.

4.2. Scene and Ray Manager

The application is split into the ray tracer and the Unity application, whereas the Unity application provides the input for the ray tracer and also handles its output. To better separate these two components, we have written the Unity side of our code to contain two important manager objects: the *Scene Manager* and the *Ray Manager* (see Figure 6).

The scene manager handles the input for the ray tracer. This means that it manages any changes that are made to the scene by the user and presents that scene data to the ray tracer. As discussed in Section 4.1, the ray tracer is implemented in Unity and can directly use Unity scene data, so all the scene manager has to do is to collect this data into one convenient scene object. This scene object is simply a list of references to objects in the Unity scene, but with one important addition: whenever an object property is modified,

[‡] https://unity.com/

an event gets sent to inform listeners that the scene has changed. These events allow us to avoid unnecessary calculations when the scene has not changed.

The ray manager handles the output of the ray tracer. It requests a list of rays from the ray tracer and is responsible for visualizing those rays. At the start of the application, the ray manager obtains a reference to the scene manager and the ray tracer. It subscribes to the events the scene manager sends out whenever a change is made to the scene. The ray manager also listens for similar events sent out by the ray tracer whenever its settings are changed. When either type of event comes in, the ray manager makes the ray tracer produce a new set of rays. These new rays are then drawn in the Unity scene, as described in Section 4.4.

4.3. Ray Tracer

Our ray tracer is based on Whitted's model [Whi80] with Schlick's approximation for refraction [Sch94]. As discussed in Section 4.1, we make use of built-in Unity functions for casting rays and determining object intersections. More precisely, we employ Unity's `Physics.Raycast` function[§]. It casts a ray from a given point and returns information about the first object with a `Collider` component it intersects. From this we determine the location of the intersection, the type of object that was intersected and its material, and all other information that is needed for ray tracing.

Note that our ray tracer's main output is a set of rays, not primarily an image. Of course, our ray tracer can still produce an image, but there is an entire set of functions for generating rays that is unique to our ray tracer. First of all, the rays themselves are stored in simple ray objects. These contain the ray's origin, direction and length, but also the ray's type and color. Rays have types for the purpose of visualization. For example, we want to be able to distinguish between a ray produced by a reflection and a ray produced by a refraction. We can thus color the rays drawn based on their type to make it clear to the user what each ray does in the scene. In the code, the color of a ray is the color it contributes to its pixel in the final image. This way we can create an image from a set of rays.

The ray tracer outputs the rays in a tree structure. Because rays are traced in a recursive fashion, this tree arises naturally: each recursively called trace function just adds its ray as a child of the ray of its caller. Each pixel thus corresponds to one tree with the root ray traced from the camera through the pixel. This means that the final output of the ray tracer is a list of ray trees (one for each pixel).

4.4. Ray Visualization

As described in Section 3.1.2, we animate the rays by elongating them in a recursive fashion. One advantage of this approach is its comparably simple implementation with our organization of rays in a tree structure (see Section 4.3). For animation, we recursively traverse this tree until we find a ray that is not fully extended and increase its length by a small amount. Doing this in each frame until all rays are at their full length results in the desired animation. It is possible to reset the animation back to the start by going through the ray trees and setting each ray's length to zero. Note that this simple approach traverses each ray tree every frame, while only a few, not fully extended rays may be of interest in that frame. If needed this could be improved by maintaining a list of the currently active rays, but the induced extra cost is negligible overall.

While often a rather small number of rays is shown for the sake of visual clarity, there are situations where drawing hundreds—up to even thousands—of rays is beneficial. Most importantly, it shows how the rays, as a collective, bounce around in the scene, and it can allow to identify individual rays with interesting behavior (e.g., a ray bouncing several times before leaving the scene). It can also help to convey a better impression of the amount of work involved in ray tracing a high resolution image.

Drawing a large number of rays does come at a significant computational cost though, so the ray drawing code needs to be well designed to handle dynamically generated rays. When the rays we need to draw have changed since the last frame, for example due to the camera being moved, we cannot simply move the existing ray objects in the same direction as the camera because the structure of the ray trees and the number of rays may have changed. Unfortunately, the simplest option to destroy the old ray objects and create new ones in the right positions is not feasible, as the Unity functions corresponding to these actions, `Destroy`[¶] and `Instantiate`[‖], are not fast enough to handle hundreds of rays. This means that we need to reuse the already existing ray objects in the scene, even if the structure of the ray trees is different from what it was before.

The solution lies in noticing that there is a difference between the plain data rays produced by the ray tracer and the Unity scene ray objects used to visualize that data. We can keep the ray objects around when the ray trees change, but we need to update their positions and colors to match the new data, and we may also have to hide some objects if the total number of rays has decreased. This can be achieved through the use of a so-called object pool.

An object pool is a design pattern commonly used in Unity applications when a lot of instances of the same type of object have to frequently be created and destroyed. It works by keeping a large number of instances of the object in a "pool". When a new object needs to be created we instead activate an unused one from the pool, and when it needs to be destroyed we deactivate it. Because activating and deactivating an object is much faster than creating or destroying it, this significantly improves performance.

In our application we store the ray objects in such a pool. When a new set of rays comes in from the ray tracer, we take one ray object from the pool for each ray, activate it, and set its position and color to reflect the ray. If there are ray objects in the pool that are still active from before but are not being used for the new rays, they are deactivated. This allows us to update hundreds of rays every frame while maintaining good frame rates.

[§] `https://docs.unity3d.com/ScriptReference/Physics.Raycast.html`

[¶] `https://docs.unity3d.com/ScriptReference/Object.Destroy.html`

[‖] `https://docs.unity3d.com/ScriptReference/Object.Instantiate.html`

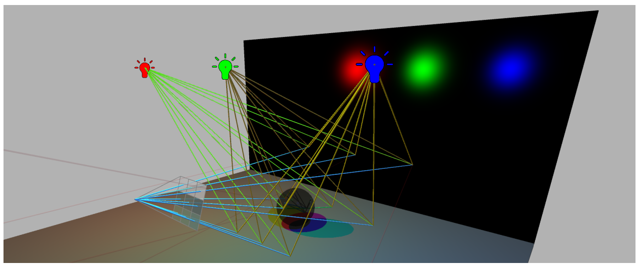


Figure 7: *The scene used to measure the performance of Virtual Ray Tracer. It contains multiple light sources, a reflective object and a transparent object.*

5. Evaluation

We first evaluate the run-time performance of Virtual Ray Tracer and then discuss our user study. Table 1 lists the timings of the tool for the scene shown in Figure 7 and an increasing resolution of the virtual screen. The data, captured on a laptop running Windows 10 with an Nvidia GTX 1050 GPU and an Intel i7-7700HQ CPU, shows that the application runs very smoothly up to a virtual screen size of 16×16 pixels on relatively modest hardware, with 32×32 still achieving interactive frame rates.

We note that the visualization already gets noticeably cluttered for 8^2 virtual screen pixels, so for most practical use cases, the frame times are well within acceptable standards. We measured the average frame time for a stationary scene and the artificial setting where the virtual camera's position is updated every frame. This forces ray paths to be recalculated continuously, simulating the worst case where the user is constantly updating the scene.

Table 1 shows that there is a modest difference between the two situations for larger virtual screen sizes. In practice, users do not tend to make changes to the scene every frame and the average frame time for the stationary scene is thus more representative.

Second, in order to determine the effectiveness of the application as an educational tool, and also to evaluate the qualitative aspects of the application, we set up a user study. We reached out to students who had previously followed the Computer Graphics course and also people from outside the Computing Science degree program at the University of Groningen. The idea behind contacting these two groups is that the Computer Graphics students should be able to provide feedback on whether the application would have been useful to them in the course, while the others can give insight into the effectiveness of the application as an introduction to ray tracing.

We recruited 17 participants, 8 of which had previously followed the Computer Graphics course. Of the 9 that had not, 6 were members of the general public with no formal education in Computer Science or a related field. We sent them a demo version of the application with a few scenes. The first scene explained the basic controls and layout of the tool. The other scenes each focused on a ray tracing concept and started with a short written explanation.

The first of these scenes showed a sphere with a diffuse material and explained diffuse reflections as well as shadows. In the next scene, specular reflections were added to the sphere. The sphere was made transparent for the scene after that. This covered all the ray tracing functionality implemented in the application, so instead of introducing new concepts, the next few scenes were more complex and focused on combining and experimenting with the concepts shown previously. Along with gradually increasing the complexity of the scenes, we also enabled more features of the application with each scene. This way the users were introduced to the application without being overwhelmed. Once the users finished the demo we asked them to fill in a short survey. The results of this survey are discussed in the following in Sections 5.1, 5.2, and 5.3.

Table 1: *Average frame timings based on internal measurements based on the Virtual Ray Tracer scene shown in Figure 7.*

Virtual screen pixels	2^2	4^2	8^2	16^2	32^2
Frame times animated scene (ms)	3.35	3.70	5.17	17.79	44.01
Frame times stationary scene (ms)	3.33	3.44	4.32	12.92	37.78

5.1. Educational Questions

The first set of questions asked the users about the educational qualities of the application. Below for each question we list the possible responses and the number of participants that selected each response. This is followed by a discussion of why we consider the question to be relevant and what we think the responses say about the application.

If you followed the course, do you think the application would be helpful to future students of the course?

1. Yes.
   Responses: 8 (100.0%).
2. No.
   Responses: 0 (0.0%).

We asked this question because the application may see use in future iterations of the Computer Graphics course. As a result, one of the main goals of the application is to be helpful to students following the course. All 8 participants that had followed the course previously answered yes to this question. We see this as a strong indication that we were successful in our goal and that the application may be of real benefit to future students.

Did the application help you understand ray tracing better?

1. Yes.
   Responses: 10 (58.8%).
2. No, but I already understood ray tracing well beforehand.
   Responses: 6 (35.3%).
3. No.
   Responses: 1 (5.9%).

This question quite directly asks whether the application succeeded in its educational goal. We can see that most participants answered yes, and of those that answered no most already had prior experience with ray tracing. This tells us that, generally speaking, the application is helpful as an introduction to ray tracing. The participant who answered no did not have a CS background, so perhaps the application was too technically complex for them. We discuss this in more detail in Section 5.2.

Do you think the application can help other people understand ray tracing better?

1. Yes, I think the application can be very helpful.
   Responses: 11 (64.7%).
2. Yes, I think the application can be moderately helpful.
   Responses: 6 (35.3%).
3. No, I don't think the application can be helpful.
   Responses: 0 (0.0%).

Since a large part of the participants had taken the Computer Graphics course, they were already familiar with ray tracing. For these participants the previous question concerning themselves becomes less expressive. This question is supposed to rectify that. Two thirds of the participants believed that the application can be very helpful to others, while one third believed it could at least be moderately helpful. This reinforces our belief that the application has real educational potential.

Which of the following things do you think were successful in helping you understand ray tracing?

1. The informative text at the start of scenes.
   Responses: 8 (47.1%).
2. The visualization of ray tracing in the scenes.
   Responses: 15 (88.2%).
3. The ability to experiment with various settings in the scenes.
   Responses: 11 (64.7%).

This final educational question attempts to pin down which parts of the application contribute the most to its educational value. We can see that almost every participant thought the visualization was helpful. We can also see that experimenting with settings is quite helpful. The informative text was the least popular option. Especially the participants who had followed the Computer Graphics course were unlikely to mark it as one of their choices, with some mentioning the lack of depth and mathematical detail in the informative text. This is an area that can be improved upon in future versions of the tool. However, adding detail to the informative text might have a negative impact on accessibility for the general public.

5.2. Technical Questions

The second set of questions asked the users about technical qualities of the application, including aspects of the application such as the visuals, the user interface, the controls and general ease of use. Again, we list below each question the possible responses and the number of participants that selected each response. This is followed by a discussion of the purpose of the question and what we think the responses say about the application.

Did you find the application easy to use?

1. Yes, the application was very easy to use and intuitive.
   Responses: 7 (43.8%)
2. Yes, but some things were confusing or difficult.
   Responses: 8 (50.0%)
3. No, but it was not very confusing or difficult either.
   Responses: 1 (6.3%)
4. No, the application was very confusing and/or difficult to use.
   Responses: 0 (0.0%)

Given its educational purpose and our aim to reach as wide an audience as possible with the application, ease of use is of high importance. From the feedback, it seems that we have largely succeeded in this with almost all participants answering yes. However, a small minority found at least some aspects of the application confusing or difficult.

What do you think of these aspects of the application?

1. The visuals.
   Very bad: 0. Bad: 0. Neutral: 3. Good: 7. Very good: 7.
2. The user interface.
   Very bad: 0. Bad: 0. Neutral: 7. Good: 9. Very good: 1.
3. The controls.
   Very bad: 0. Bad: 1. Neutral: 3. Good: 10. Very good: 3.
4. The scenes and explanations of ray tracing concepts.
   Very bad: 0. Bad: 0. Neutral: 3. Good: 11. Very good: 3.

The previous question told us that some aspects of the application could do with improvement. This question should point out which aspects those are, but it hopefully also tells us what was already good. We see that the controls and user interface are rated the lowest, but on average most users still thought they were acceptable. While the other aspects are important as well, note that the visuals were our main focus in this first iteration of our tool.

What do you think of the complexity of the application?

1. The application is too simple. More settings and controls would be an improvement.
   Responses: 0 (0.0%)
2. The complexity of the application is good.
   Responses: 16 (94.1%)
3. The application is too complex. There are too many unnecessary settings and controls.
   Responses: 1 (5.9%)

We discussed in Section 3.2 how we tried to give the user as much control as possible without making the application too complex. This question should determine whether we were successful. We see that almost all users thought the complexity of the application was good. Interestingly, the one person who disagreed answered in a previous question that they found the application very easy to use. Below, we discuss some more detailed responses we gathered.

5.3. Additional Comments

Finally, we asked our participants to state their thoughts on both the educational and technical aspects of the application. From this we can determine the general sentiment of the participants.

The educational aspects of the application were very well received. There were numerous positive comments about the scenes and the visualization. For example, one participant commented: "*I really liked how you could see the rays being traced, how they bounced off of different objects, and how they refracted. It was a great visual showcase of how ray-tracing works, and I think students should be able to better learn the concepts using it.*"

Some of the participants who had followed the Computer Graphics course commented on a lack of more detailed, mathematical explanations of the ray tracing concepts covered by the application. One such participant said: "*In context of the CG course I think this* *would be nice to see as an introduction to the corresponding topics (shadows, reflections etc) however it does lack a bit in the mathematics part of ray-tracing.*"

The technical aspects, however, elicited some negative comments. Most of these pointed out parts of the UI or controls that were perceived complicated or confusing. From the responses to the multiple choice questions (Section 5.2) we can gather that the general opinion of the technical aspects of the application is positive. However, the comments pointed out small aspects of the application that can be improved upon in future versions.

6. Conclusion

We have developed Virtual Ray Tracer, an interactive application that visualizes ray tracing. It was designed to help with teaching ray tracing and to be used in the Computer Graphics course taught at the University of Groningen. We have evaluated the application through a user study. Regarding the educational potential of the application, the results are positive. It is clear that visualizing ray tracing can be of great help in understanding it better. It is also useful to be able to change ray tracer settings and the properties of objects in a scene with the visualization updating based on every change made. This interactivity allows users to experiment and see how various settings affect the rays being traced.

Some aspects of the application were less well received, however. Mainly the user interface and controls were deemed partly confusing. Especially users without a CS background had some difficulty with the application. This can partially be remedied by more carefully designing these aspects of the application, but at some point the only reasonable way to reduce complexity is to start cutting features. Doing this would give users less room to experiment and play with the application, reducing its educational effectiveness. Depending on the target demographic, this may or may not be a worthwhile trade-off. In conclusion, we are pleased with the application in its current state and the largely positive feedback we received from users. We are looking forward to using it in the next edition of our Computer Graphics course.

We see several directions for future work. The responses to the survey, especially the comments discussed in Section 5.3, clearly indicate there is potential for improving user-friendliness. Beyond UI improvements, we plan to add a tutorial to make the application more accessible. Another option would be customize the application to different target groups, e.g., to CS students, school children, or the audience of an exposition. For students of the Computer Graphics course, it could be useful to extend the application in a way that allows them to adjust the ray tracing process itself. We also aim to conduct an extended study with a larger group of participants and evaluate various learning goals.

The capabilities of our Virtual Ray Tracer could be extended by acceleration structures (such as bounding volume hierarchies) as well as more advanced ray tracing approaches (distribution-based and stochastic methods). Additionally, we plan to incorporate ray tracing also for rendering the scene visuals. The challenge here is to maintain high responsiveness of our tool on commodity hardware.

In our implementation, we generally chose simplicity over complexity, but modifications could be able to improve performance further, and will be considered in future work. For instance, by implementing the ray tracer using Unity functions in C# we were able to use the same copy of the scene representation for different purposes, but were unable to multithread the code. Unity also supports C/C++ libraries in the form of native plug-ins[**], which could be used to further improve the performance of our application.

Virtual Ray Tracer is fully available on GitHub [vWV22], which is also where all related material and future updates can be found.

Acknowledgements This paper is based on the BSc theses of the first two authors at the University of Groningen.

References

[DNL*17] DENG Y., NI Y., LI Z., MU S., ZHANG W.: Toward Real-Time Ray Tracing: A Survey on Hardware Acceleration and Microarchitecture Techniques. *ACM Comp. Surveys 50*, 4 (2017), 58:1–41. 1

[GFE*12] GRIBBLE C., FISHER J., EBY D., QUIGLEY E., LUDWIG G.: Ray tracing visualization toolkit. In *Proceedings of ACM SIGGRAPH* (New York, USA, 2012), I3D '12, ACM, pp. 71–78. 2

[HAM19] HAINES E., AKENINE-MÖLLER T. (Eds.): *Ray Tracing Gems*. Apress, 2019. http://raytracinggems.com. 1

[MS21] MARSCHNER S., SHIRLEY P. (Eds.): *Fundamentals of Computer Graphics*. CRC Press, 2021. 1

[MSW21] MARRS A., SHIRLEY P., WALD I. (Eds.): *Ray Tracing Gems II*. Apress, 2021. http://raytracinggems.com/rtg2. 1, 3

[Pho75] PHONG B.: Illumination for computer generated pictures. *Communications of the ACM 18*, 6 (1975), 311–317. 3

[Rus99] RUSSELL J.: An interactive web-based ray tracing visualization tool. *Undergraduate Honors Program Senior Thesis. Department of Computer Science, University of Washington* (1999). 2

[SAH*16] SIMONS G., AMENT M., HERHOLZ S., DACHSBACHER C., EISEMANN M., EISEMANN E.: An Interactive Information Visualization Approach to Physically-Based Rendering. In *Vision, Modeling & Visualization* (2016), The Eurographics Association. 2

[Sch94] SCHLICK C.: An inexpensive BRDF model for physically-based rendering. *Computer Graphics Forum 13*, 3 (1994), 233–246. 5

[SHP*19] SIMONS G., HERHOLZ S., PETITJEAN V., RAPP T., AMENT M., LENSCH H., DACHSBACHER C., EISEMANN M., EISEMANN E.: Applying visual analytics to physically based rendering. *Computer Graphics Forum 38*, 1 (2019), 197–208. 2

[SJL15] SPENCER B., JONES M., LIM I.: A visualization tool used to develop new photon mapping techniques. *Computer Graphics Forum 34*, 1 (2015), 127–140. 2

[SSM02] SMYK M., SZABER M., MANTIUK R.: *JaTrac — an exercise in designing educational raytracer*. Springer, 2002, pp. 303–311. 2

[VGV*20] VITSAS N., GKARAVELIS A., VASILAKIS A., VARDIS K., PAPAIOANNOU G.: Rayground: An Online Educational Tool for Ray Tracing. In *Eurographics 2020 - Education Papers* (2020), Romero M., Sousa Santos B., (Eds.), The Eurographics Association. 2

[vWV22] VAN WEZEL C., VERSCHOORE W.: Virtual Ray Tracer, 2022. URL: https://github.com/wezel/Virtual-Ray-Tracer. 4, 8

[Whi80] WHITTED T.: An improved illumination model for shaded display. *Commun. ACM 23*, 6 (June 1980), 343–349. 5

[ZAD15] ZIRR T., AMENT M., DACHSBACHER C.: Visualization of coherent structures of light transport. *Computer Graphics Forum 34*, 3 (2015), 491–500. 2

[**] https://docs.unity3d.com/Manual/NativePlugins.html