university of groningen

faculty of science and engineering

A Virtual Ray Tracer

Bachelor thesis

June 30, 2021

Student: W.A. Verschoore de la Houssaije

Primary supervisor: Prof. Dr. J. Kosinka

Secondary supervisor: Dr. S.D. Frey

Advisor: A. Tiwary

Abstract

Ray tracing is one of the more complicated techniques commonly taught in Computer Graphics. Visualizations can help in understanding difficult ray paths and interactions, but none of the currently existing applications that perform this kind of visualization are targeted at education.

In this thesis we discuss the development of an application that allows users to view and interact with the ray tracing process in real-time. The application shows a scene containing a camera casting rays that interact with objects in the scene. Users are be able to change ray properties such as their speed, the amount of rays as well as the material properties of the objects in the scene. The goal of the application is to help the users understand ray tracing better.

We conducted a user study to determine the effectiveness of the application. From it we can conclude that the application can be of help in ray tracing education although it may be too complicated for users without a CS background. We discuss the reasons for this and propose potential solutions.

Contents

Chapter 1

Introduction

Ray tracing 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 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.

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 such as the one in Figure 1.1 are frequently employed, but they are 2-dimensional and static, which limits their effectiveness as an educational tool.

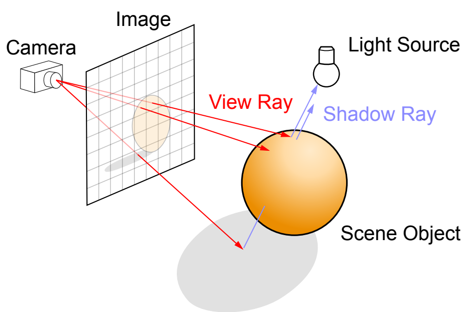


Figure 1.1: A simple illustration of ray tracing. Credit: Henrik/Wikipedia. Accessed from `https://commons.wikimedia.org/wiki/File:Ray_trace_diagram.svg#file`

To address this issue, this thesis discusses the development of an interactive application that visualizes the ray tracing process. In doing this we aim to determine the best way to visualize ray tracing and to evaluate the effect the application has on the learning process. The application can then help users understand ray tracing faster and better than they would without interactive visualizations.

The main demographic for the application is students following the Computer Graphics course at the University of Groningen. The application may be used in future iterations of the course to help teach students about ray tracing. However, the application is built to be accessible to as wide an audience as possible, so anyone interested in ray tracing may get something out of it.

In the next chapter we look at related work. In Chapter 3 we discuss the general design of the application while Chapter 4 covers the implementation details. In Chapter 5 we evaluate the results of the user study we conducted. We state our conclusions in Chapter 6 and discuss potential future work in Chapter 7.

Chapter 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 [5, 4, 7, 9], they are generally not 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 normally do not visualize the ray tracing process [6, 8].

The idea for an educational application that visualizes ray tracing is not entirely unique however. One of the first implementations comes in the form of a set of Java Applets developed in 1999 [3], see Figure 2.1. Unsurprisingly, its age means that the application was very simple, and it is 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.

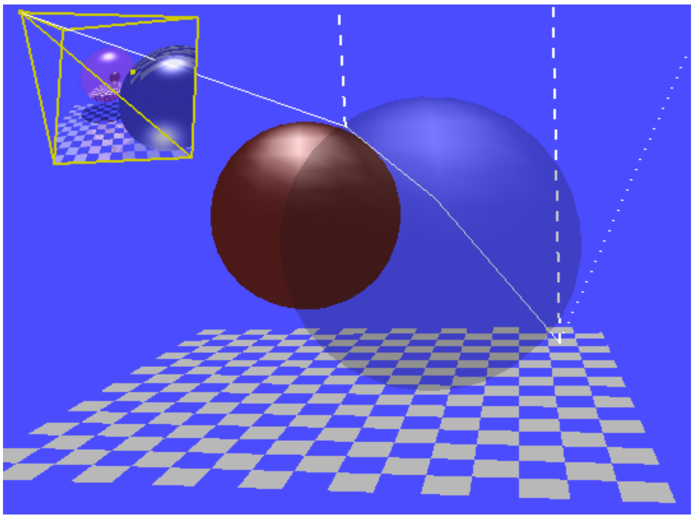


Figure 2.1: An interactive web-based ray tracing visualization tool from 1999 (image adopted from [3]).

A more recent application similar to ours is the Ray Tracing Visualization Toolkit (rtVTK) [1], see Figure 2.2. As the name suggests, the rtVTK is a toolkit for the visualization of ray tracing. The main goals of the 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 the 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 will of course not do. It seems there is room for a dedicated and user friendly educational application that visualizes the ray tracing process.

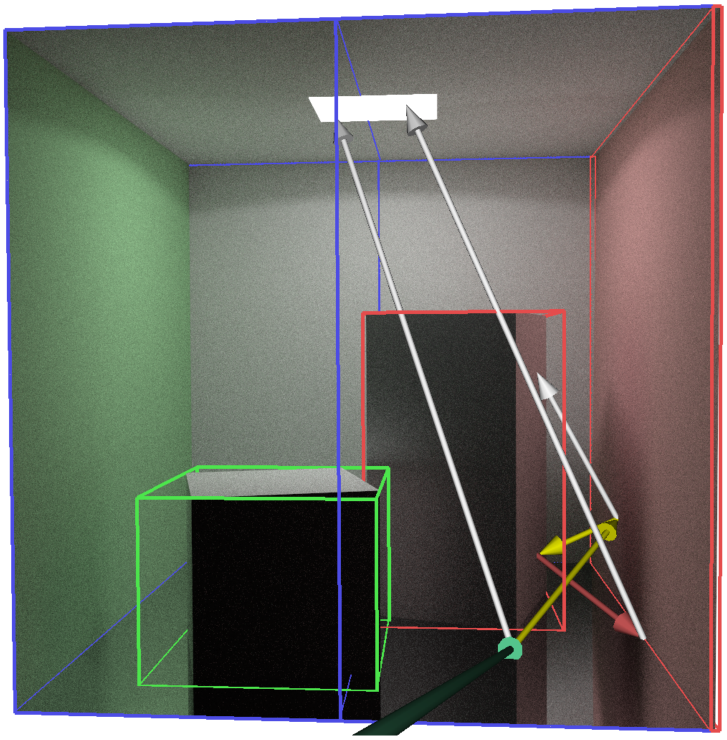


Figure 2.2: The Ray Tracing Visualization Toolkit (image adopted from [1]).

Chapter 3

The Application

The default setup for the application is a scene with some objects, lights and always one camera. 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 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. Figure 3.1 shows a simple scene.

The user can interact with the objects, lights and camera in the scene by clicking on one to select it. This will open a properties panel. When the user changes a property of the selected object, the changes 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 in a render 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 bring up a high resolution ray traced render of the scene. Figure 3.2 shows the same simple scene as Figure 3.1, but with the UI elements enabled.

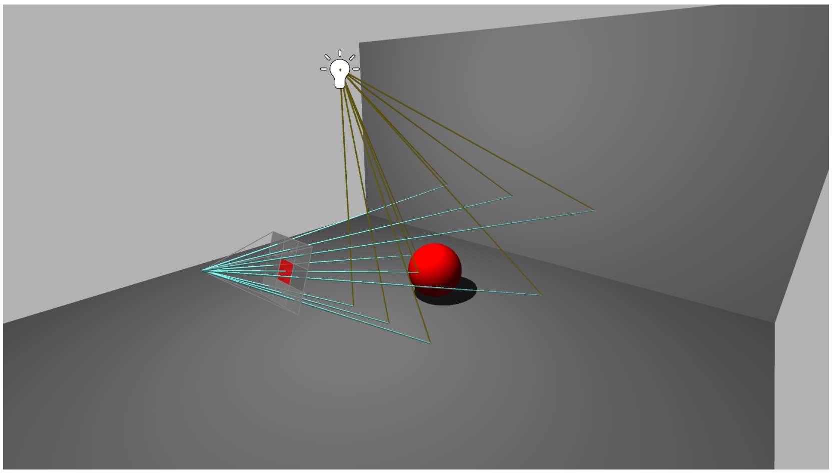


Figure 3.1: A screenshot of the application showing a simple scene with all UI elements disabled.

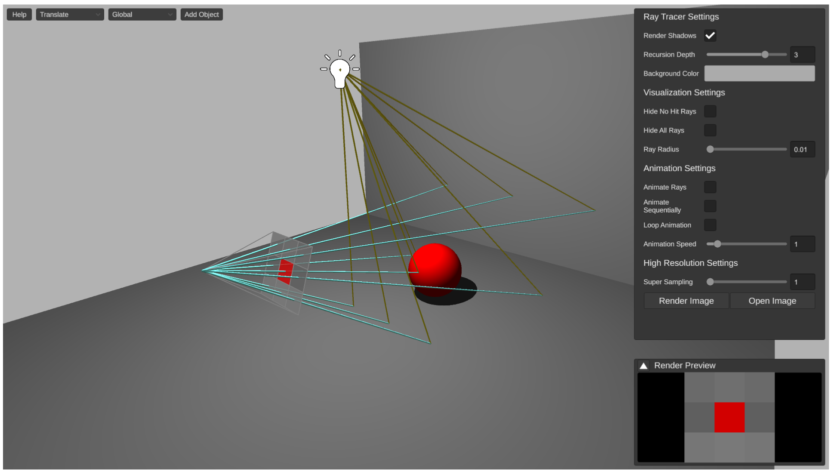


Figure 3.2: A screenshot of the application showing a simple scene with all UI elements enabled.

3.1 Unity

Since writing the application from scratch is infeasible, we decided at the start that we needed to use a game engine, or some similar type of software, to develop the application. We eventually settled on Unity[1] as the best option.

Unity is a freely available game engine that can be used for 2D and 3D applications. One of the main reasons we chose Unity is because it is well documented and there are many resources online that explain and discuss various parts of the game engine. Unity also has build support for a wide range of platforms, which is important because we want the application to be available to as many people as possible.

Other game engines such as Unreal and Godot match Unity in terms of documentation and build support. What made us choose Unity over these other options is that we already had some prior experience with Unity. Also, scripting in Unity is done in C#, which is (in our opinion) easier than lower level languages such as C++ as is used for scripting in Unreal.

3.2 Visuals

The core of the application is the visualization of ray tracing. This can be split into two components: the visualization of the scene, and the visualization of the rays traced in the scene.

3.2.1 Scene Visuals

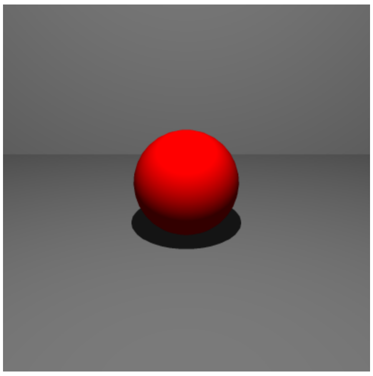
Ideally, the scene would look identical to the final ray traced image. This way users immediately see what the result looks like without having view it in a separate window. However, the application also needs to be interactive, allowing

[1] `https://unity.com/`, `https://en.wikipedia.org/wiki/Unity_(game_engine)`

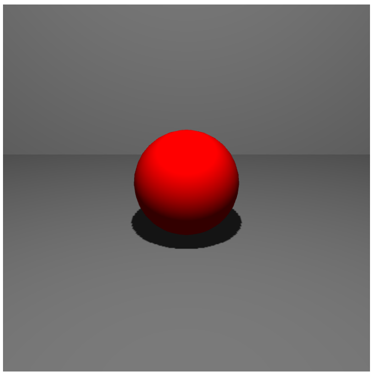
users to change the scene in real-time. Since ray tracing is still too slow for this, the scene cannot be rendered to exactly match the ray traced result.

The best we can do is to match the ray traced scene as closely as possible using rasterization techniques. By using a custom shader based on the same Phong illumination model [2] 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. The only exception is shadows, since Unity provides built in support for that.

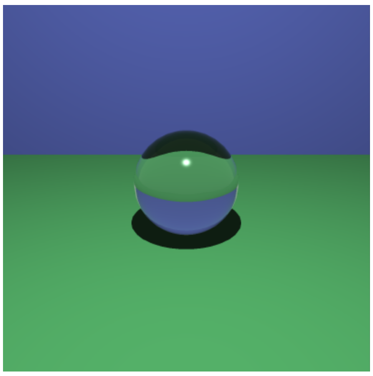
The major downside of this approach is that it can get rather confusing when the scene visuals do not match the ray traced result. This is especially true for transparent materials as shown in Figure 3.3. The alternative would be to put no effort into matching the ray tracer at all, for example by rendering 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.



(a) Ray traced diffuse material.



(b) Diffuse material in Unity.



(c) Ray traced refractive material.

(d) Refractive material in Unity.

Figure 3.3: A comparison of different materials in the Unity scene and in the ray traced render. Diffuse materials look nearly identical, but transparent materials are not properly visualized in Unity.

3.2.2 Ray Visuals

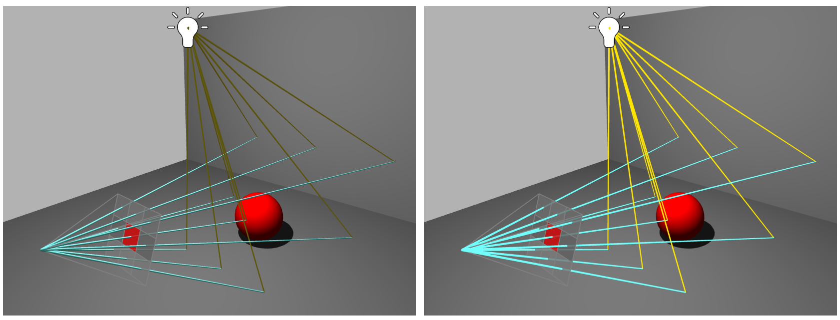
In order to visualize ray tracing in an intuitive and appealing way, we need to carefully consider how we draw the rays. It needs to be clear where a ray is coming from, where it is going, and what kind of ray it is. All this combined with the fact that it should be possible to draw many rays at once 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 rest of the scene hard to see. This becomes especially true when the number of rays visualized is large. Therefore, the rays should have a simple shape and material.

The simplest way to shade a ray is to give it a solid color. The main question is whether the rays should be affected by lights in the scene or not. Ignoring lighting is simpler and, since we color rays based on their type, makes it easier to distinguish between different kinds of rays. If the rays are affected by the lights in the scene, a variation in lighting conditions might make rays of the same type appear to have different colors. This could be confusing to the user. There is one major downside to the no lighting approach though, it means that there are very few context clues about the position of a ray. In practice, it makes rays not affected by lights look as if they are two dimensional, even if the actual ray object is three dimensional. The difference between the two approaches is shown in Figure 3.4. In our estimation this is a big enough problem that we decided to color the rays based on the scene's lighting conditions.

The simplest shape for a ray, and the shape that we have decided to use, is a cylinder. It can be argued that a cylindrical arrow (see Figure 3.5) 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 better 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, any child rays (e.g. a reflection ray) will start their animation. We do things this way because it clearly demonstrates the recursive nature of ray tracing.



(a) Rays affected by scene lights. The rays look 3D and their position are easily determined.

(b) Rays unaffected by scene lights. The rays look 2D and their position are hard to determine.

Figure 3.4: A comparison of lit and unlit ray materials.

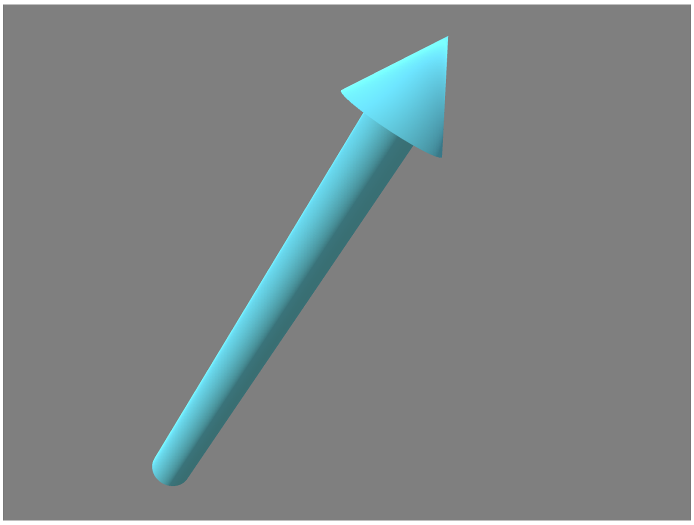


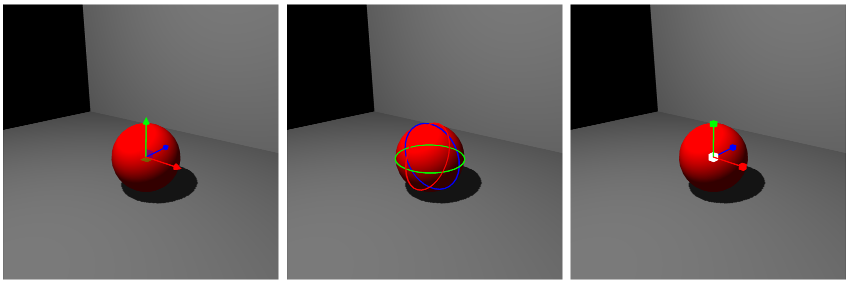
Figure 3.5: A cylindrical arrow.

3.3 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 because we believe this is a great way to learn. What that means is that we allow users to change a wide variety of settings regarding the ray visualization and also properties of objects in the scene. This comes at the cost of increased complexity though. With more settings available the application also risks becoming more confusing and difficult to use.

We can partially avoid this by designing a good user interface. Of course, this is somewhat subjective, but we can discuss some choices we made and why we think they help decrease complexity. For example, 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 immediately 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 when the user actively requests it. This idea of showing only what is necessary is also reflected in the fact that the properties panel's contents change based on the user's selection. For example, when the camera is selected, only its settings are displayed.

In the end, the best UI is 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, but positioning, rotating and scaling objects in the scene can also be done without adding more settings to the properties panel. 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, including Unity's editor. The shapes are often called transformation gizmos. See Figure 3.6 to see 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 nicer way to transform an object.



(a) Translation gizmo. (b) Rotation gizmo. (c) Scale gizmo.

Figure 3.6: Transformation gizmos.

Chapter 4

Implementation

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. The Unity application visualizes the scene, but more importantly, it takes the rays from the ray tracer and visualizes those. This concept is illustrated in Figure 4.1.

In practice, the design needs to be slightly more complex. The main issue is that Figure 4.1 implies that the ray tracer and the Unity application use the same scene representation. In reality, the Unity application has its own internal scene that is a lot more complex than what is needed for the ray tracer. The main reason for this additional complexity is that the Unity scene also contains all information for visualization and the UI. For example, consider the rays drawn in the Unity scene. These should of course be ignored by the ray tracer when it is producing the next set of rays to be drawn. However, it is important to note that the Unity and ray tracer scenes are not fundamentally different; the ray tracer scene is effectively a subset of the Unity scene.

Another problem with Figure 4.1 is that it represents the scene as a static object. However, the scene changes constantly as users move objects, change materials and experiment with settings. These changes are all handled by the Unity application and represented in its internal scene. The Unity application then needs to make sure that the ray tracer also receives the updated scene information.

With this in mind, we can create a more practical design. The Unity application takes input from the user and changes its internal scene. These changes then need to be sent to the ray tracer, but the ray tracer uses its own scene representation different from the Unity scene. Because of this, we include a translation layer that converts the Unity scene to a format that the ray tracer understands. The ray tracer then outputs a list of rays for the Unity application to draw. This concept is illustrated in Figure 4.2.

4.1 Original Design

The Computer Graphics course taught at the University of Groningen uses a ray tracer written in C++ for the practical assignments on ray tracing. Originally, the plan was to adapt this ray tracer for the application. This is possible

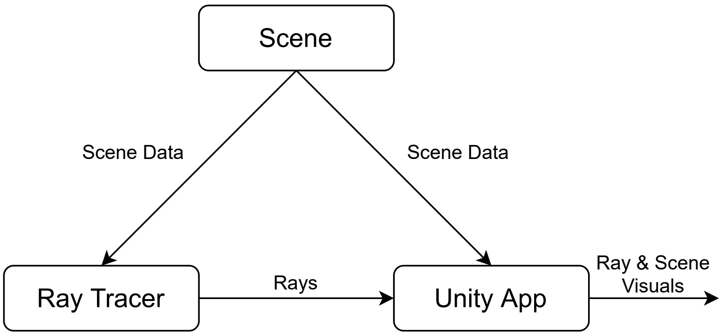


Figure 4.1: A simple design for the general structure of the application.

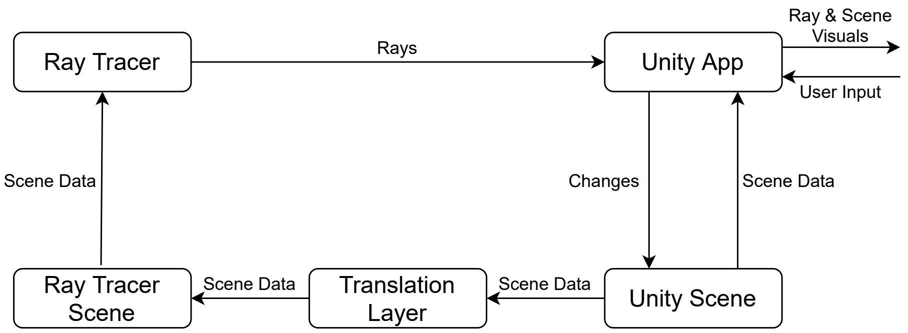


Figure 4.2: An improved design for the general structure of the application.

because, while scripting in Unity is done in C#, Unity also supports C/C++ libraries in the form of native plug-ins[1].

Using the ray tracer from the course has a number of advantages. The obvious one is that it saves us from having to write our own ray tracer. However, an arguably more important point is that by using the ray tracer from the course we guarantee that the application's behavior matches what students have come to expect from ray tracing. In theory, a ray tracer written in C++ will also be faster than an equivalent one written in C#. This is especially true because the ray tracer from the course can very easily be parallelized using OpenMP. Parallelizing C# code is more difficult. Interestingly, we moved away from using the C++ ray tracer because of poor performance achieved in practice.

While the actual ray tracing in C++ is indeed very fast, the communication between C++ and C# is not. Because some properties of objects like their position and color can be changed in a continuous fashion, we need the translating, sending and ray tracing of the scene to happen within a fraction of a second, otherwise the application starts feeling unresponsive. However, C++ plugins run separately from the Unity application and therefore a significant amount of work had to be done to translate the Unity scene and send it to the C++ ray tracer. In the end, we were not able to make this process fast enough for our performance requirements, so we had to look for another solution.

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

4.2 Current Design

The design we settled on has the ray tracer implemented in Unity. Unity has built in utilities for casting rays and determining object intersections that are fast enough for (simple) ray tracing. Implementing the ray tracer in Unity has several advantages and some disadvantages.

The main advantage is that little to no time has to be spent on translating the scene and sending it to the ray tracer. This reduction in communication time means that, while the actual ray tracing takes longer, the time between updating the scene and visualizing the rays is orders of magnitude smaller than with the C++ ray tracer. The reason no translation has to take place is that the ray casting functions in Unity work directly with the internal scene.

Another advantage is that this approach decreases the overall complexity of the application. There is no longer a need to store two separate versions of the scene, because the ray tracer uses the same scene as the Unity application. This means that we do not have to worry anymore about propagating changes made to the Unity scene to the ray tracer, we can simply apply any changes in Unity and the ray tracer will access the same, updated data. A side effect of this is that the ray tracer and the rest of the application are more tightly coupled than before, which means that changes made in the one component are more likely to also inadvertently affect the other component.

Finally, it is important to note that code interacting with a Unity scene cannot be multithreaded. Unity allows only the main thread to make calls to the Unity API, as it is not thread safe. This means that any ray tracer making use of Unity's ray casting functionality is forced to run on one thread. Given the time constraints of the thesis project we judged this to be a reasonable sacrifice. It is most likely possible to write a ray tracer in C# that includes its own ray casting and intersection code while keeping scene translation time reasonable. Such a ray tracer could be multithreaded. This possibility is discussed in further detail in Chapter 7.

4.3 Detailed Design

At the start of this chapter we discussed how the application can be split into the ray tracer and the Unity application. However, as can be seen in Figure 4.2, 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.

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 4.2, 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, an event gets sent to inform listeners that the scene has changed. These events prevent us from doing unnecessary calculations when the scene has not changed.

The ray manager handles the output of the ray tracer. This means that 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. The details of ray visualization are discussed in Section 4.5

Putting everything together, we have a scene manager for the input, a ray manager for the output, and in between them sits the ray tracer. The input for the ray tracer is scene data and its output is rays to be visualized. This architecture is illustrated in Figure 4.3.

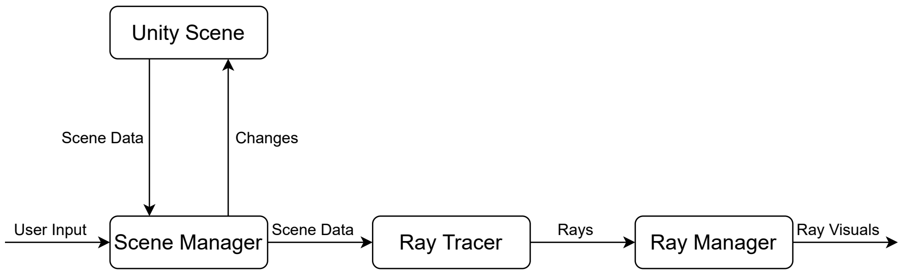


Figure 4.3: A more detailed design for the general structure of the application.

4.4 Ray Tracer

Up to now, we have mentioned the ray tracer a lot, but we have yet to discuss the details of its implementation. Largely, the ray tracer is a C# translation of the C++ ray tracer from the Computer Graphics course. Luckily C++ and C# are very similar languages, which makes the translation relatively straightforward. There are, however, a couple of important deviations.

The first is that, as explained in Section 4.2, we are using built-in Unity functions for casting rays and determining object intersections. More precisely, we make use of Unity's `Physics.Raycast` function[2]. This function casts a ray from a given point and returns information about the first object with a `Collider` component it intersects. From this we can 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.

The second difference with the ray tracer from the course is that our ray tracer's main output is a set of rays, not an image. Of course, our ray tracer can still produce an image, and the code for that matches the C++ ray tracer very closely, but there is an entire set of functions for producing rays that is unique to our ray tracer.

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

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

between a ray produced by a reflection and a ray produced by a refraction. This means we can color the rays drawn based on their type to make it clear to the user what each ray is doing 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 comes about naturally; each recursively called trace function just adds its ray as a child of the ray of its caller. When rendering an image with multiple pixels this way, each pixel corresponds to one tree of rays with the root ray traced from the camera origin through the pixel. This means that the final output of the ray tracer is a list of ray trees, one tree for each pixel.

4.5 Ray Visualization

We discussed the design aspects of the ray visualization in Section 3.2.2, but there is also significant programming work behind the ray visuals.

4.5.1 Ray Animation

As described in Section 3.2.2, we animate the rays by elongating them in a recursive fashion. One advantage with this approach is that very little code is required to make it work. As explained in Section 4.4, the rays are stored in a tree structure, so all we have to do is to recursively traverse the tree until we find a ray that is not fully extended and increase its length by a small amount. If we do this each frame until all rays are at their full length we get 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.

An issue with this method is that it traverses each ray tree every frame, while only a few, not fully extended rays may be of interest that frame. This could be solved by maintaining a list of the currently relevant rays, but that causes problems when a new set of rays is produced by the ray tracer. If the camera or an object has been moved, the ray trees might have radically different structures. It would be complicated to maintain the list in those situations, so traversing each ray tree from the start becomes necessary anyway. Because the performance cost of the animation code is not very significant anyway, making the code more complicated by maintaining such a list for an optimization that only works is some situations is unnecessary.

4.5.2 Performance

Unlike the animation logic, optimizations are needed to get satisfactory performance in the general ray drawing code. While it is often best to limit the number rays drawn in order to keep thing clear, there are situations where drawing hundreds, perhaps even thousands, of rays is a good idea. It gives better insight into the amount of work involved in ray tracing a high resolution image, and it shows how the rays, as a collective, bounce around in the scene. This does come at a significant computational cost though, so the ray drawing code needs to be well designed.

When the rays are static, there is no real problem. Unity is perfectly capable of drawing a couple thousand ray objects at a decent frame rate, even on older hardware. This is does require the mesh used by the ray objects to be relatively simple, but that is not an issue for us because, as we discussed in Section 3.2.2, we use basic cylinders for our rays.

When the rays we need to draw have changed since the last frame, for example due to the camera being moved, we do need to be careful about how we proceed. 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. It may then seem easiest to simply destroy the old ray objects and create new ones in the right positions. However, the Unity functions corresponding to these actions, `Destroy`[3] and `Instantiate`[4], are not fast enough to handle hundreds of rays. This means that we do need to find a way to reuse the already existing ray objects in the scene, even if the structure of the ray trees is radically 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 an object pool.

An object pool is a design pattern commonly used in Unity applications where 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 can 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 a decent frame rate.

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

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

Chapter 5

Evaluation and Results

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. 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 better insight into the effectiveness of the application as an introduction to ray tracing.

We ended up finding 17 participants, 8 of which had previously followed the Computer Graphics course. We sent them a demo version of the application with a few scenes. The first scene explained the basic controls and layout of the application. The other scenes each focused on a 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 previously shown. 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 next.

5.1 Educational Questions

The first set of questions asked the users about the educational qualities of the application. In this section we discuss these questions. Below 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 asked the question 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: 9 (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. Curiously, one participant who had not followed the course also answered yes to this question.

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 idea 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%).

We anticipated that a large part of the participants may have already been familiar with ray tracing. The previous question becomes less useful then. 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 where 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. This was expected; the entire application was built on the belief that visualizing ray tracing is a good idea after all, but it is good to see it confirmed in this way. We can also see that experimenting with settings is quite helpful.

5.2 Technical Questions

The second set of questions asked the users about technical qualities of the application. Think of aspects of the application such as the visuals, the user interface, the controls and general ease of use. In this section we discuss these questions. Below 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 asked 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 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 confusion or difficult either.
   Responses: 1 (6.3%)
4. No, the application was very confusing and/or difficult to use.
   Responses: 0 (0.0%)

Because we want to reach as wide an audience as possible with the application, it is important that it is easy to use. It seems that we have largely succeeded in this with almost all participants answering yes. However, a small majority found at least some aspects of the application confusing or difficult.

What do 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. The visuals are arguably the most important part of the application considering the focus is on visualizing ray tracing. It is good to see then that they are rated the highest.

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 3.3 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. This indicates the limitations of multiple choice questions. In the next section we discuss some more detailed responses we gathered.

5.3 Additional Responses

Because participants of the survey may have additional comments that do not fit within the constraints of multiple choice questions, we added an option for everyone to leave 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. The technical aspects, however, elicited more negative comments. Most of these pointed out parts of the user interface or controls that were complicated or confusing. Interestingly, this does not really match the answers given to the multiple choice questions, which highlights the importance of allowing participants to leave additional comments.

Chapter 6

Conclusion

The goal of this thesis was to develop an interactive application that visualizes ray tracing. This application should help with teaching ray tracing and could be used in the Computer Graphics course taught at the University of Groningen. We have developed this application and evaluated it through a user study.

With regards to 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 confusing. Especially users without a CS background had 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.

Chapter 7

Future Work

No development project is ever entirely finished. This also applies to the application. There are a number things that could be improved or added, all of which would make the application even better at helping people understand ray tracing.

The responses to the survey, especially the comments discussed in Section 5.3, make it clear that the application can be more user friendly. Beyond general UI improvements, which is a slightly nebulous concept, a tutorial is a concrete addition to the application that could help people understand some of the more confusing settings and controls. It would walk users through various aspects of the application and force them to explicitly confirm that they understand one part before allowing them to continue to the next. The problem with this is that it may become frustrating to users who pick up the basics quickly.

An additional solution to the complexity problem is to create two different versions of the application. Because people with a CS background are less likely to get confused by a large number of settings and controls, we leave the application unchanged for them. Other users might actually benefit from a stripped down version of the application with only the most important settings and controls enabled. This would give them less room to experiment, but would hopefully prevent them from becoming too confused or overwhelmed.

On the subject of more experienced users, students of the Computer Graphics course may find it useful to visualize custom written ray tracer code. In the practical assignments of the course students extend a preexisting ray tracer, for example by adding reflections. The only output they get is the final rendered image. Especially when there is a bug in their code, it would valuable to students if they could have the application visualize the rays traced by their code. This would allow them to more easily spot bugs and get a better understanding of what their code is actually doing. However, it is important to note that this would take a large amount of work to implement, most likely requiring a significant rewrite of the existing ray tracer.

Rewriting the ray tracer may be good idea regardless of whether user written ray tracer code functionality is added. As described in Section 4.2, by implementing the ray tracer using Unity functions we are unable to multithread the code. It also needlessly couples the ray tracer to the Unity scene. To separate the ray tracer from Unity we would need to write a pure C# representation of the scene. The ray tracer could then also be implemented in plain C# and it would do its calculations using this scene. Any changes made to the scene by the user would then affect the C# scene first, after which they have to be propagated to the Unity scene. In a way this increases complexity by introducing two distinct representations of the scene that have to be kept up to date, but it would make for cleaner code and a faster ray tracer after implementing multithreading.

These are not the only improvements that could be made to the application, but they some of the most interesting and impactful ones. As mentioned at the start of this chapter, development is never done, so even if all these ideas are implemented there would still be more work to do. In the end, we are pleased with the application as it is now and the largely positive feedback we received from users.

Acknowledgements

I would like to thank my supervisors, Jiri Kosinka and Steffen Frey, for their help with the project. They were always available for questions and gave great feedback on the application. I would also like to thank Ankur Tiwary for his help in setting up the user study. Finally, I would like to thank Chris van Wezel for co-developing the application with me. I could not have done it on my own.

Bibliography

[1] 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*, I3D '12, page 71–78, New York, NY, USA, 2012. Association for Computing Machinery.

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

[3] J.A. Russell. An interactive web-based ray tracing visualization tool. *Undergraduate Honors Program Senior Thesis. Department of Computer Science, University of Washington*, 1999.

[4] G. Simons, M. Ament, S. Herholz, C. Dachsbacher, M. Eisemann, and E. Eisemann. An Interactive Information Visualization Approach to Physically-Based Rendering. In *Vision, Modeling & Visualization*. The Eurographics Association, 2016.

[5] G. Simons, S. Herholz, V. Petitjean, T. Rapp, M. Ament, H. Lensch, C. Dachsbacher, M. Eisemann, and E. Eisemann. Applying visual analytics to physically based rendering. *Computer Graphics Forum*, 38(1):197–208, 2019.

[6] M. Smyk, M. Szaber, and R. Mantiuk. *JaTrac — an exercise in designing educational raytracer*, pages 303–311. Springer US, Boston, MA, 2002.

[7] B. Spencer, M.W. Jones, and I.S. Lim. A visualization tool used to develop new photon mapping techniques. *Computer Graphics Forum*, 34(1):127–140, 2015.

[8] N. Vitsas, A. Gkaravelis, A. Vasilakis, K. Vardis, and G. Papaioannou. Rayground: An Online Educational Tool for Ray Tracing. In Mario Romero and Beatrice Sousa Santos, editors, *Eurographics 2020 - Education Papers*. The Eurographics Association, 2020.

[9] T. Zirr, M. Ament, and C. Dachsbacher. Visualization of coherent structures of light transport. *Computer Graphics Forum*, 34(3):491–500, 2015.