Real-time Visualisation of Volume Raycasting in Virtual Ray Tracer

Philip Blesinger

Bachelor's Thesis

University of Groningen

October 16, 2024

Bachelor's Thesis

Real-time Visualisation of Volume Raycasting in Virtual Ray Tracer

Philip Blesinger

October 16, 2024

Contents

1 Introduction **3**

2 Background **4**
2.1 Ray Tracing . . . 4
2.2 Volume Ray Casting . . . 5
2.2.1 Data Representation . . . 5
2.2.2 Casting a Ray & Collection of Samples . . . 5
2.2.3 Trilinear Interpolation . . . 5
2.2.4 Compositing Method . . . 6
2.2.5 Transfer Function . . . 7
2.3 Lighting . . . 7
2.3.1 Phong Illumination . . . 7

3 Method **8**
3.1 Initial Considerations . . . 8
3.2 GPU Ray Casting . . . 9
3.3 3D Texture Sampling . . . 9
3.4 Improving Depth Perception . . . 9
3.5 Lighting for Isosurface Visualisation . . . 9
3.5.1 Central Differences for Surface Normal Estimation . . . 9
3.6 Failed Solutions . . . 10

4 Implementation **10**
4.1 3D Texture Generation . . . 10
4.2 Enabling Camera Depth Texture . . . 10
4.3 Ray Casting Shader . . . 10
4.3.1 Shader Properties . . . 10
4.3.2 Blending Operations . . . 11
4.3.3 Vertex Shader . . . 12
4.3.4 Central Differences . . . 12
4.3.5 Fragment Shader . . . 12
4.3.6 Interaction with other Geometry . . . 12
4.4 Extensions in Unity Environment . . . 13
4.4.1 Synchronization of Settings with Unity Ray Caster . . . 13
4.4.2 Performance Considerations . . . 14

5 User Study **14**
5.1 Design . . . 14
5.2 Results . . . 14
5.3 Analysis of Results . . . 14
5.3.1 Effectiveness of the Ray Caster Application . . . 14
5.3.2 Impact of the Visual and Interactive Features . . . 15
5.3.3 User Satisfaction with Specific Features . . . 15
5.3.4 Hardware and Responsiveness . . . 15
5.3.5 General Feedback and Improvements . . . 15

6 Conclusion **15**

7 Future work **16**
7.1 Acceleration Structures . . . 16
7.2 Efficiency-Related Optimisations . . . 16
7.3 Configurable Lighting Conditions for Isosurface . . . 16

A User Study Questionnaire **16**

B User Study Results **19**

Abstract

Ray tracing is a fundamental technique in computer graphics, used to generate realistic images by simulating the behavior of light as it interacts with objects in a scene. The Virtual Ray Tracer (VRT) was developed as an educational tool to provide users with an interactive environment for understanding ray tracing concepts. While VRT has successfully visualized surface-based ray tracing, recent efforts have expanded its capabilities to include volume ray casting, a technique used for rendering the internal structures of volumetric data.

This thesis extends the functionality of VRT by improving the visualization of volume ray casting. A new accelerated ray casting pipeline was implemented to allow users to interact with and visualize voxel-based datasets in real time. Key enhancements include real-time visualization of the voxel grid and the ability to dynamically adjust settings such as sampling distance, transfer functions, and compositing methods.

Through this work, VRT now supports an interactive and intuitive exploration of volume rendering concepts, with a focus on educational effectiveness and usability. The results of this project contribute to a deeper understanding of ray casting for both novice and advanced users, making VRT a more powerful tool for learning computer graphics principles.

A user study revealed that the Ray Caster application significantly improved users' understanding of ray casting, particularly through its interactive features and live preview, with most participants reporting enhanced comprehension after using the tool.

1 Introduction

In the field of Computer Graphics, ray tracing is one of the core concepts in computer visualisation techniques. It, along with other core concepts to this Thesis, is briefly summarised in section 3. In order to give students of the Rijksuniversiteit Groningen and any other interested party a more intuitive understanding of Ray Tracing, Virtual Ray Tracer, henceforth referred to as VRT, was created [9]. It is an interactive tool made using the game engine Unity [4], which visualizes the process of ray tracing while performing it, letting the user manipulate components on-the-fly in order to gain an understanding of their role within the system. A plethora of visual media explaining the process of ray tracing and volume ray casting in the form of various animations and short films already exists. However, none of them are interactive. The only such tool we could find was a Java Applet from 1999 [6], but presumably due to its age it can no longer be accessed on the web.

While the idea of visualizing ray paths in and of itself is not original, Virtual Ray Tracer [8] forms the only interactive environment where the user can freely tinker with various aspects of ray tracing and casting in order to discover their purpose and impact upon the final image.

Following the VRT project's initial success, it was updated to improve its functionality and educational effectiveness [8]. Volume Ray casting support was added later [7], with the volume being displayed to the user as a transparent cube in 3d space. Rays are visualized as traveling through the volume, with each point where a sample is taken being highlighted. Virtual Ray Caster allows the user to freely choose the transfer function, compositing method and other parameters such as the distance between samples taken [7]. Given the success of the previous three projects on VRT, we further expanded upon its ray casting capabilities.

The previous voxel-grid-based ray casting simulation was limited in transparency by the lack of a visualisation of the voxel grid before an image is rendered using the ray casting pipeline. We endeavoured to implement a separate accelerated ray casting pipeline in order to grant a volumetric view of the grid during manipulation of the 3d preview.

The appearance of the voxel grid as a mere grey cube, as can be seen in Figure 1, did not facilitate an intuitive understanding of the sampling process. Therefore, the volume needed to be visualized to the user in the preview as well, not just when they apply their settings and fully render a ray-cast image, preferably by way of ray-casting as well, albeit with a faster method in order to maintain responsiveness of the application.

This thesis is structured as follows: In the **Background** section, we provide an overview of ray tracing and volume ray casting, focusing on key concepts such as data representation, sample collection, and lighting models. The **Method** section outlines the initial considerations for improving depth perception and enhancing visualization of the voxel grid, followed by a discussion of GPU ray casting, 3D texture sampling, and custom lighting techniques. In the **Implementation** section, we describe the technical details of generating

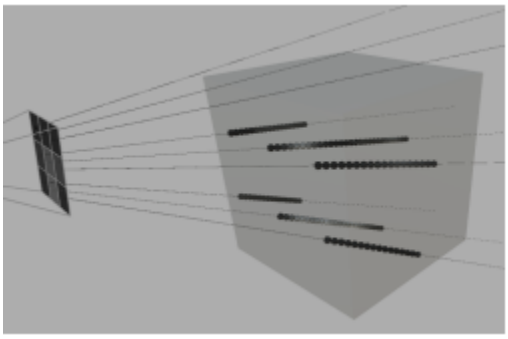


Figure 1: The previous preview for Virtual Ray Caster [7]. A ray is cast through each pixel of the camera, taking sample (highlighted in black) at regular intervals. The voxel grid is shown as a transparent grey cube.

3D textures, configuring shaders, and integrating the system within the Unity environment. The **User Study** presents the design and results of an experiment conducted to evaluate the effectiveness of the Ray Caster application, focusing on user experience and performance. Finally, the **Conclusion** and **Future Work** sections summarize the findings and suggest potential improvements and extensions for the project, such as acceleration structures and further optimization of rendering techniques.

2 Background

2.1 Ray Tracing

Ray tracing is a process where rays are cast through each pixel in a virtual camera, determining the color of the pixel upon contact with surfaces, taking into account material colour, light sources etc., as can be seen in Figure 2.

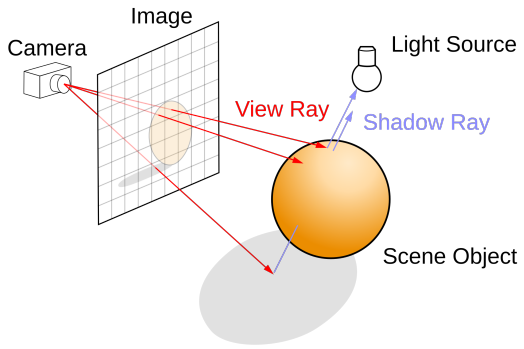


Figure 2: Representation of surface ray tracing, image adopted from [5]

Mathematical Model

Mathematically, the path of a ray can be described by a parametric equation:

$$\mathbf{P}(t) = \mathbf{O} + t\mathbf{D}$$

where:

- $\mathbf{P}(t)$ is the position of the ray at time t,
- $\mathbf{O}$ is the origin of the ray (the camera or a reflection point),

- **D** is the direction of the ray, and
- t is a scalar representing the distance traveled along the ray.

In the context of this thesis, the term ray *tracing* refers to processes where the surface of an object is visualized, whereas (volume) ray *casting* refers to rays being cast through a volume and showing its internal structure.

2.2 Volume Ray Casting

Volume ray casting is a process where rays are cast through a volume, taking samples of the density of the material along the way, in order to visualize the internal structure of a heterogeneous volume. As an example, this technology is often used to visualize medical scans.

2.2.1 Data Representation

When working with volume rendering, one of the simplest and most common types of volumetric data is a 3D voxel (volumetric pixel) grid of density values. This grid represents a spatial distribution of density throughout a volume, and each voxel (a small cubic unit in the grid) stores a scalar density value. These density values describe how much material is present in a given voxel, and this information is critical for simulating how light interacts with the volume during rendering.

2.2.2 Casting a Ray & Collection of Samples

For each pixel in the final image, a ray is cast from the camera through the voxel grid. The direction of the ray is determined by the camera's orientation and field of view, just like in surface-based ray tracing.
Once a ray is cast, it begins to march through the 3D voxel grid, moving step by step through the volume. In each step, the ray samples the density value at the voxel it is currently passing through. The ray then moves forward by a fixed step size. Smaller step sizes improve accuracy but require more computation, while larger steps reduce computational cost but may miss fine details in the volume.

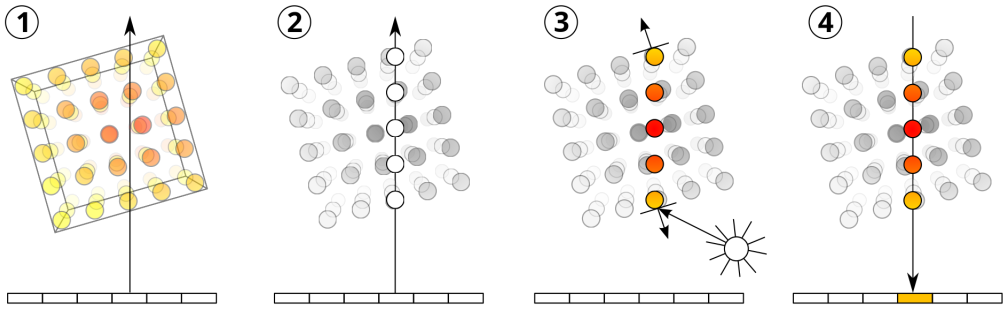


Figure 3: The four basic stages of ray casting: Ray casting, sampling, applying the transfer function and compositing, shown respectively. Image adopted from [10]

2.2.3 Trilinear Interpolation

Since rays are cast through the volume continuously, they often pass between the centers of voxels, requiring a method to estimate the scalar values at non-voxel locations. Trilinear interpolation is one such method that allows us to compute an approximate value at any arbitrary point inside the voxel grid by interpolating between the values at the surrounding voxels.

Trilinear interpolation is a three-dimensional extension of bilinear interpolation. It interpolates scalar values within a cube formed by eight adjacent voxels. These eight voxels surround the point at which the

interpolation is performed. The process is divided into three linear interpolation steps, corresponding to the three axes (x, y, and z) of the voxel grid.

Trilinear interpolation provides a continuous approximation of the scalar field across the voxel grid, even though the underlying data is stored at discrete locations. This method ensures that the transitions between voxel values are gradual, which is particularly important for accurately visualizing smooth structures within the volume. It also prevents visual artifacts that might arise from directly sampling at discrete voxel locations (nearest-neighbor interpolation), which would result in a blocky or pixelated appearance.

While trilinear interpolation improves visual quality by smoothing out the sampled data, it is computationally more expensive than simpler methods like nearest-neighbor interpolation because it requires multiple interpolation steps. However, it strikes a balance between computational efficiency and accuracy, making it one of the most widely used techniques for volume rendering.

In Virtual Ray Caster, trilinear interpolation allows for smoother and more detailed rendering of the volume, enhancing the accuracy of sample collection along rays and improving the overall visual fidelity of the rendered images.

2.2.4 Compositing Method

A compositing method is a technique used in volume rendering to combine information from multiple samples along the path of a ray to produce a final pixel color in an image. In volume ray casting, as rays pass through the voxel grid, they collect values at multiple points inside the volume. The compositing method defines how these sampled values are aggregated into a single output that determines, along with the transfer function2.2.5, the final color and opacity of the pixel.
Some compositing methods are designed to produce and image that informs the user about the composition of the volume as a whole, whereas others isolate more specific aspects, such as the occurrence of a specific density or the shape of the object's surface.
VRT includes the following compositing methods (with the formulae being given by the previous Ray Caster version [7]):
In each formula, let:

$$
\begin{aligned}
C_n &= \text{Composited colour at sample number } n \\
C_0 &= (0,0,0) \\
O_n &= \text{Composited opacity at sample number } n \\
O_0 &= 0 \\
c_n &= \text{colour of sample number } n \\
o_n &= \text{opacity of sample number } n \\
d_n &= \text{density of sample number } n \\
f(d) &= \text{transfer function of density } d \\
T &= \text{Termination point for early ray termination} \\
X &= \text{Opacity cutoff value} \\
Y &= \text{Density matching value}
\end{aligned}
$$

1. **Accumulate:** This function integrates the color and opacity along the ray by progressively adding contributions from each voxel, weighted by transmittance. Ideal for visualizing effects where light or color builds up as it travels through the volume, such as in smoke, fog, or clouds. This method is useful for capturing gradual transitions and effects that depend on the accumulation of light, and gives the best overall understanding of the composition of a volume at first glance.
   The formula for accumulation is given by:

$$
\begin{aligned}
C_n &= C_{n-1} + (1 - O_{n-1}) \cdot c_n \\
O_n &= O_{n-1} + (1 - O_{n-1}) \cdot o_n \\
T &= O_n >= X
\end{aligned}
$$

2. **First:** The first voxel encountered with a desired density defines the final color and opacity; Commonly used for applications where the initial intersection with the volume is the most significant, like in medical imaging scenarios where one wants to highlight the first layer or boundary. This method can be described by:

$$C_n, O_n = \begin{cases} f(D), \exists n \in [1, n], d_n = D \\ f(0) \end{cases}$$
$$T : C_{n-1} \neq C_0$$

3. **Average:** This function computes the mean of the color or opacity values along the ray's path. The average is computed using:

$$C_n, O_n = f(\frac{\sum_{i=1}^{n} d_i}{n})$$

4. **Maximum:** The final color or opacity is determined by the maximum value encountered along the ray. This can be expressed as:

$$C_n, O_n = f(\max(d_1, ..., d_n))$$

2.2.5 Transfer Function

The transfer function maps density values to color values, such that layers with different densities can be visually distinguished. It is a key component in volume ray casting as it defines how the scalar values (e.g., densities) within our volumetric dataset are mapped to visual properties such as color and opacity. This mapping allows different features within the volume to be distinguished and rendered effectively.
VRT's transfer function maps scalar density values to RGBA values, where:

- **R, G, B:** represent the color associated with a specific density value.
- **A:** (alpha channel) controls the opacity or transparency of the corresponding region within the volume.

The transfer function is designed interactively, with users adjusting it based on the characteristics of the data they wish to highlight. For example, a high-density region in medical imaging might be mapped to a specific color to represent bone, while lower densities might correspond to soft tissue or air, and be assigned different colors or degrees of transparency.
VRT includes a histogram for each voxel grid, which shows the distribution of densities to the user in order to inform their decision about what values the transfer function should be set to. At the same time, a recommended Transfer function loads up upon selection of each voxel grid.

2.3 Lighting

2.3.1 Phong Illumination

Phong illumination is a model used to simulate the way light interacts with surfaces to create illumination effects. It involves calculating three types of light reflection: ambient, diffuse, and specular. Below is a brief breakdown of each component:

- **Ambient Reflection:**
  - Represents the constant, non-directional light present in the scene, ensuring that objects are not completely dark even if they are not directly illuminated.
  - Calculated as:

$$I_{\text{ambient}} = k_{\text{ambient}} \cdot I_{\text{ambient}}$$

    where k_{ambient} is the ambient reflection of the surface, and I_{ambient} is the intensity of ambient light.

- **Diffuse Reflection:**
  - Models the scattering of light in all directions when it hits a rough surface. The intensity varies with the angle between the light vector and the surface normal.
  - Calculated using Lambert's cosine law:

$$I_{\text{diffuse}} = k_{\text{diffuse}} \cdot I_{\text{light}} \cdot \max(0, \mathbf{L} \cdot \mathbf{N})$$

    where k_{diffuse} is the diffuse reflection, I_{light} is the luminosity of the light, $\mathbf{L}$ is the light vector, and $\mathbf{N}$ is the normal vector of the surface, both normalised.

- **Specular Reflection:**
  - Simulates the bright spots of light that appear on shiny surfaces where the light is reflected directly to the viewer. It depends on the viewer's angle and the light reflection direction.
  - Calculated using the Phong reflection model:

$$I_{\text{specular}} = k_{\text{specular}} \cdot I_{\text{light}} \cdot (\max(0, \mathbf{R} \cdot \mathbf{V}))^{\alpha}$$

    where k_{specular} is the specular reflectance, $\mathbf{R}$ is the reflection vector of the light, $\mathbf{V}$ is the view direction, and α is the shininess exponent controlling the size of the specular highlight.

- **Final Color Calculation:**
  - The final color of a point is the sum of ambient, diffuse, and specular contributions:

$$I_{\text{final}} = I_{\text{ambient}} + I_{\text{diffuse}} + I_{\text{specular}}$$

3 Method

3.1 Initial Considerations

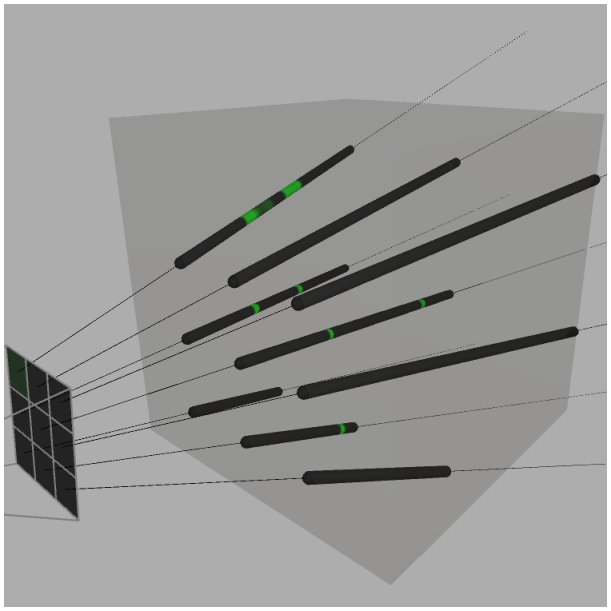


Figure 4: Bunny voxel grid without preview

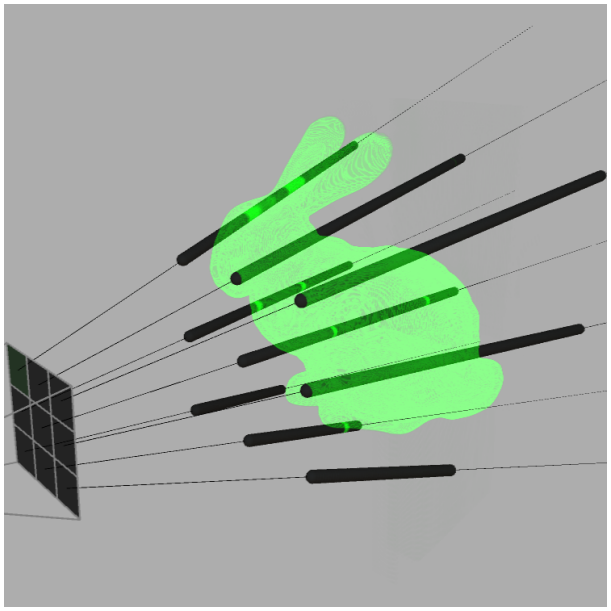


Figure 5: Bunny voxel grid with preview

In order to provide the user with more active feedback as to the impact of the settings that they are changing in the VRC, the voxel grid needs to display meaningful visual changes that correspond to those

settings, such as in Figure 5. The closer these changes are to what will happen when the user presses 'render image', the more easily the user will understand the impact of their choices, therefore gaining a better understanding of the core concepts of volume ray casting, which is our ultimate goal.

3.2 GPU Ray Casting

Following the logic of our initial considerations, it makes sense to have the voxel grid display an image generated in the same way as is done by the Unity Ray Caster. However, the calculations performed when the user presses the "render image" button are quite extensive and even modern and fast CPUs need close to 30 seconds to complete one render.
Luckily GPUs are designed to perform thousands of such tasks very quickly. We therefore need to adapt the ray casting process such that it can be handled by one. The task will entail writing a custom shader for the voxel grid object in Unity, as well as passing relevant data on to the GPU in an appropriate format.

3.3 3D Texture Sampling

In order to store our density voxel grid in such a way that it can be accessed by the GPU, we converted it to a 3D texture.
A 3D texture is a type of texture map that extends the traditional 2D texture into three dimensions. It stores data in a three-dimensional grid, where each voxel (volume element) corresponds to a specific texel (texture element) in the texture map. This setup very closely resembles a voxel grid, meaning that conversion of voxel grid data from Virtual Ray Caster can be done very easily, by storing densities in the alpha channel of each texel's RGBA field.

When the GPU samples a 3D texture, trilinear interpolation is automatically handled within the shader and requires no custom implementation from the developer.

3.4 Improving Depth Perception

In order to visualize both the preview of the voxel grid and the sample objects inside the volume which are generated by the Unity Ray Caster, whilst preserving the user's depth perception, we needed a way of blending the two correctly. To achieve this, we implemented custom depth testing, detailed in Section 4.3.6.

3.5 Lighting for Isosurface Visualisation

In order to make using the 'First' compositing method more intuitive to the user, we implemented basic Phong (see Section 2.3.1) illumination.
We defined a point light source at an arbitrary location in order to calculate lighting. We estimated surface normals by way of central differences:

3.5.1 Central Differences for Surface Normal Estimation

The surface normal at a point is a vector perpendicular to the surface at that point. In a voxel grid, the surface normal can be estimated by approximating the gradient of the scalar field values at the voxel.
The central differences method estimates the gradient by calculating the difference in scalar values between neighboring voxels. This method provides a better approximation of the gradient compared to forward or backward differences because it considers both directions around a point.
The gradient of the scalar field can be approximated using the following central differences formula for each axis:

$$\mathbf{N} = \begin{bmatrix} \rho(x+\Delta, y, z) - \rho(x-\Delta, y, z) \\ \rho(x, y+\Delta, z) - \rho(x, y-\Delta, z) \\ \rho(x, y, z+\Delta) - \rho(x, y, z-\Delta) \end{bmatrix} \tag{1}$$

Where:

- ρ is a function representing the sampling of density from the voxel grid

- Δ is a given offset

Once the normal vector is calculated, it needs to be normalized (to a length of 1) before it can be used for illumination.

3.6 Failed Solutions

One of our first attempts at visualising the voxel grid was to place an invisible second virtual camera within the unity environment, which generated a high density of sample objects within the grid in order to create a dynamic 3D model of the grid. This method had both low visual fidelity and terrible efficiency, as generating large amounts of geometry was not only inefficient in terms of computational complexity and memory, but also caused a lot of blending issues between the objects.
Another attempt was to subdivide the grid into smaller cubes and visualise their mean density by applying the transfer function to them. We discarded this solution because we had enough success with our GPU-based approach early on.

4 Implementation

The new version of VRT with live raycasting can be found here:
github.com/Glusis/Virtual-Ray-Tracer-Ray-Casting-Support-2

4.1 3D Texture Generation

To create a 3D texture for each voxel grid, we created a utility script for developers of VRT to turn any new voxel grid file into a corresponding 3D texture. It can be run from within the Unity development environment without running play mode [2].
There is also a script in place to store pre-computed normals for voxel grids, facilitating more efficient lighting calculations in the future, but it is not yet in use.
First, the developer needs to manually adjust the dimensions of the voxel grid that is to be processed. Next, they input the file path of the ".raw" file in which the voxel grid for Virtual Ray Caster is stored.
Upon startup, the script then creates a new 3D texture and loops through each texel, saving the density from the ".raw" file in the alpha channel of the texel's color. Changes to the texture are then applied and the new Asset is saved in the Resources folder with a file name specified by the developer.

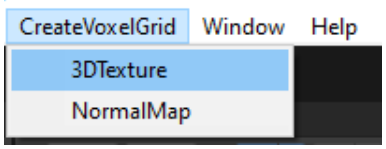


Figure 6: The script can be executed from the Editor

4.2 Enabling Camera Depth Texture

An auxiliary script is attached to the voxel grid object, which enables the generation of the main camera's depth texture, needed for depth testing in Section 4.3.6.

4.3 Ray Casting Shader

4.3.1 Shader Properties

The shader's properties are its variables which are exposed to the Unity environment and can be edited by other scripts, providing an endpoint where synchronized variables can be passed along.

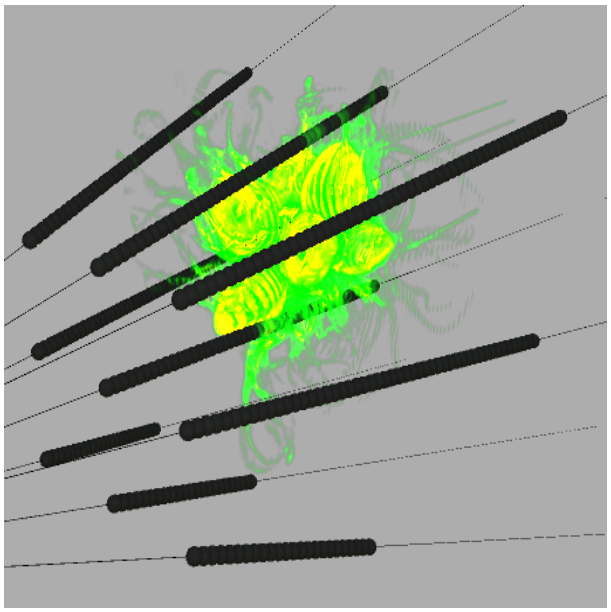


Figure 7: Hazelnut with Accumulate compositing method

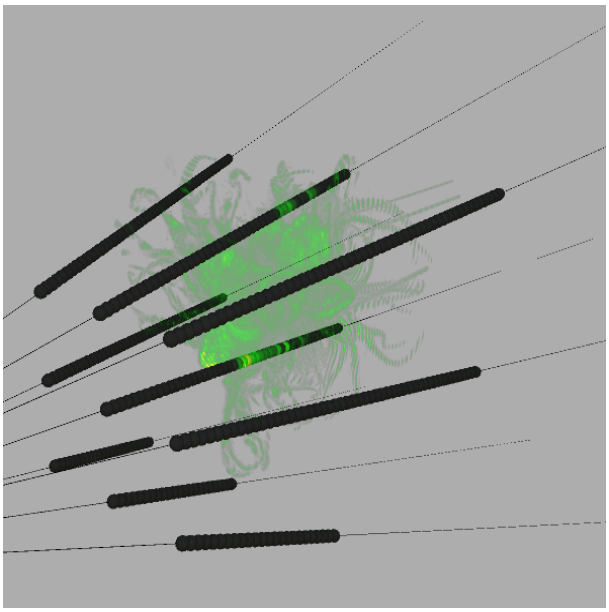


Figure 8: Hazelnut with Average compositing method

- `_MainTex`: The 3D texture containing the density values for raycasting.
- `_MinDensity`: A threshold below which density values are ignored.
- `_StepSize`: The distance between each sample along the ray.
- `_AlphaCutoff`: Maximum allowed opacity for compositing.
- `_CompositingFunction`: Determines which compositing method is used (0 for Accumulate, 1 for Maximum, 2 for Average, 3 for First Hit). Examples are shown in Figures 4.3 and 4.3.
- `_TargetDens`: Target density for the first compositing method.
- `_Transfer1` to `_Transfer5`: Lookup table parameters for color transfer.
- `_Transfer1c` to `_Transfer5c`: Color values used for mapping densities to colors.

4.3.2 Blending Operations

The shader includes some settings to ensure that the voxel grid blends correctly with other geometry:

- `"Queue" = "AlphaTest"` puts the object at position 2450 in Unity's render pipeline, after all opaque objects, to ensure that they are in the depth buffer before the shader executes.
- `ZWrite Off` disables writing to the depth buffer, to treat it as a transparent object.
- `ZTest LEqual` sets built-in depth testing conditions. (This is separate from the custom depth test described in Section 4.3.6.) More information can be found in [3]
- `Blend SrcAlpha OneMinusSrcAlpha` Sets the combination of the output of the fragment shader with the render target to premultiplied transparency. More information be found in [1]

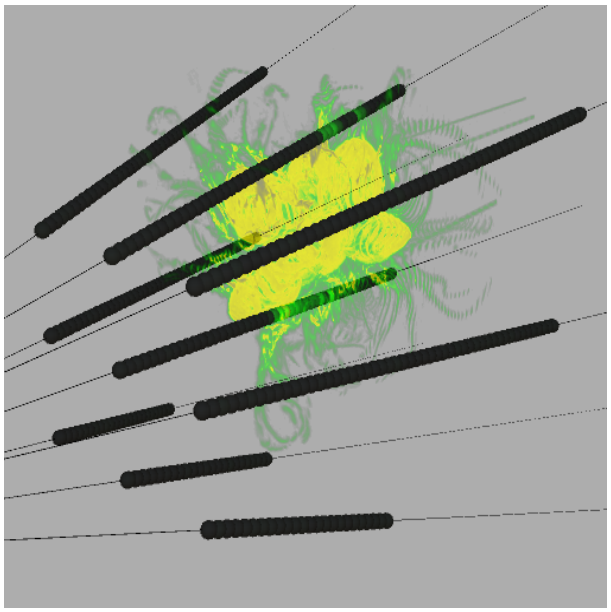


Figure 9: Maximum compositing method

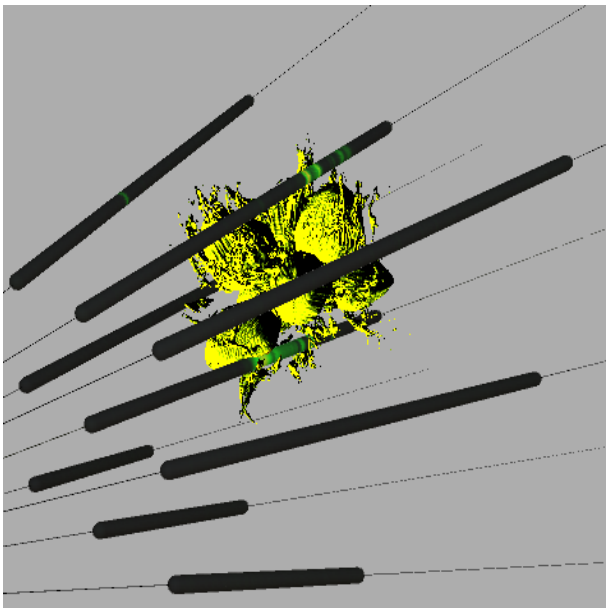


Figure 10: First compositing method

4.3.3 Vertex Shader

The vertex shader performs three tasks:

- Prepares the object vertex by converting it to object space, in order to set a starting point for our ray marching ray.
- Calculates the unit vector from the camera to the object, which will determine the direction of the ray in the fragment shader.
- Calculates the vertex's clip position, which is needed to get the screen UV coordinates to sample the depth texture.

4.3.4 Central Differences

To compute surface normals as outlined in Section 3.5.1, the `centralDiff` method computes a vector composed of the differences in density between voxels along all axes, given a delta value which determines the distance from the point for which the normal is to be estimated. At this distance, in both directions, samples are taken and the two samples are subtracted from each other. As for each axis such a difference is computed, the three differences together create a vector which will serve as a surface normal.

4.3.5 Fragment Shader

The first meaningful calculation the fragment shader performs is determining the number of samples taken per ray, by taking the step size (the distance between each sample, given by the user in terms of voxels, then converted to Unity units) and dividing the diagonal of (and therefore longest distance across) the voxel grid by it. After this step count is established, a loop runs through each sample position, samples the 3D texture to get a corresponding density value and applies the selected compositing function.
Afterwards, the Transfer function is applied to output the desired color.

4.3.6 Interaction with other Geometry

In order for the user to see the Unity ray caster's samples without being occluded by the voxel grid's preview, the voxel grid shader needs to be aware of other geometry inside of it. This is achieved by depth testing,

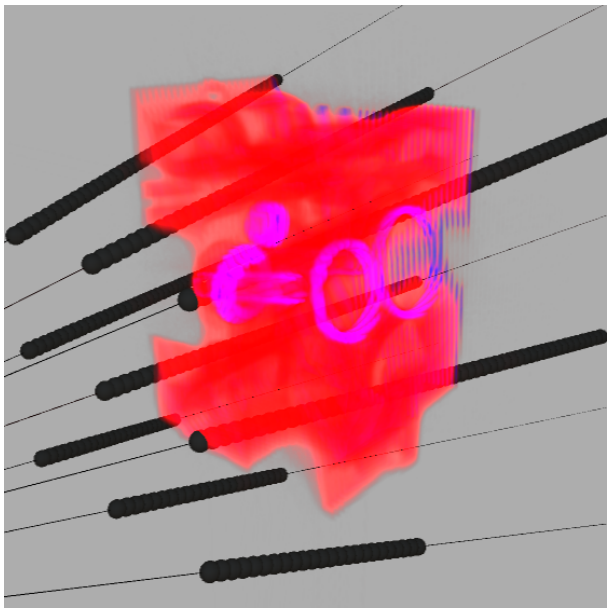


Figure 11: Accumulate *without* depth testing

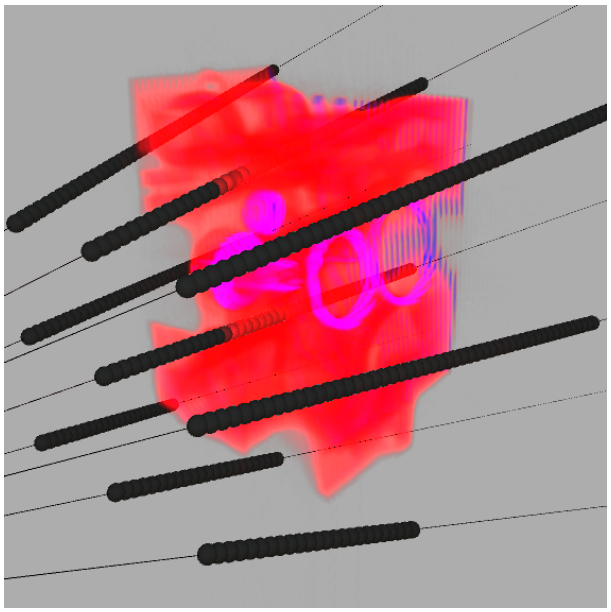


Figure 12: Accumulate *with* depth testing

shown in Figure 12.
Unity's built-in depth testing, however, is not able to interfere with the ray casting process itself and therefore produces incorrect results when objects are placed inside of the voxel grid. We thus supplement the process by accessing the main camera's depth texture and computing a vector between the depth of our ray's starting point and the depth of any geometry in the depth buffer. The length of this vector gives us the distance the ray can travel until it hits other geometry. (Excluding, of course, any transparent objects, which are placed behind the voxel grid in the pipeline and do not write to the depth buffer.) We then place a condition upon the sampling loop which terminates the ray if it exceeds the length of this depth vector.
This results in behaviour which closely resembles a transparent object, letting the user see both the ray and sample objects generated by the virtual camera of Virtual Ray Caster and the volume preview, with accurate depth perception.

4.4 Extensions in Unity Environment

4.4.1 Synchronization of Settings with Unity Ray Caster

The synchronization of settings between the custom ray casting shader and the Unity Ray Caster ensures that changes made by the user in the graphical interface are immediately reflected in the ray casting preview. This synchronization is achieved through an event-driven system that updates shader parameters in real-time as users adjust settings like sampling distance, transfer functions, and compositing methods. This approach allows users to interactively tweak these parameters and see the impact on the rendered preview without needing to manually restart the rendering process.

Key parameters that are synchronized include:

- **Step Size**: Adjusting this setting updates the distance between samples along each ray, allowing users to balance between rendering precision and performance.
- **Transfer Function Parameters**: Any changes to the mapping of densities to colors are directly applied to the shader, enabling users to experiment with different visualizations of the volumetric data in real time.
- **Compositing Method**: Switching between different compositing approaches, such as accumulation or maximum intensity projection, is reflected instantly, giving users an immediate sense of how different methods affect the final visualization.

This real-time synchronization not only improves the user experience by making the application more responsive but also serves as an educational tool by allowing users to understand the effects of various settings dynamically. For example, users can see how adjusting the step size impacts the level of detail in the rendered volume or how different transfer functions highlight specific features of the data.

4.4.2 Performance Considerations

While the Ray Caster's preview proved to be sufficiently efficient to not hamper user experience in terms of responsiveness in the user study, we decided to give the user the ability to toggle the preview off, in case they had performance issues anyway. For example, they could be running the program on a Computer which does not have a dedicated GPU, which may have application responsiveness implications which we did not test.

5 User Study

5.1 Design

To test the efficacy of our additions, we gave students access to the new version of VRT with the following instructions:

> Open **Virtual Ray Tracer.exe**, select **Levels**, then **15. Ray Casting**.
> Follow the Tutorial on the bottom left of the screen twice:
>
> 1. On the first playthrough, **disable** the "Show Preview" function (at the very bottom of the right menu, as explained in the tutorial).
> 2. On the second, leave it enabled and see how it changes your experience.
>
> Next, fill out this questionnaire:
> https://forms.gle/Neb1H7dzjXyU7bQe9
> Thank you so much for participating!

The questionnaire was composed as described in Appendix A.

5.2 Results

A visualization of our results can be found in Appendix B.

5.3 Analysis of Results

From analyzing the questionnaire data, we can draw several conclusions regarding how the Ray Caster application is perceived by its users:

5.3.1 Effectiveness of the Ray Caster Application

- Across various users, the understanding of ray casting improved after using the application. Here's the average improvement at each stage:
  - Before using the application: ~ 4.87 (median around 4).
  - After using the application without the live preview: ~ 6.5.
  - After using the application with the live preview: ~ 7.78.

  The clear trend shows that users benefit from the live preview, and their understanding of ray casting increases more when this feature is used.

5.3.2 Impact of the Visual and Interactive Features

- The majority of respondents noted that the **visualization of the virtual camera** and the ability to **experiment within the application** were helpful in understanding ray casting. Specifically, these two elements were highlighted by almost all participants.
- The **live preview** was also mentioned frequently as a significant aid in comprehension, particularly by those with a lower initial understanding of ray casting.

5.3.3 User Satisfaction with Specific Features

- Ratings for helpful features such as visuals, user interface, tutorial, and breakdowns are generally high, with most features rated between 6 and 10 by the users.
- The breakdown of ray calculations received slightly lower ratings compared to visuals and responsiveness, indicating potential areas for improvement in clarity or user interaction.
- The **text tutorial** was frequently mentioned, and the feedback suggests it was a valuable part of the learning experience.

5.3.4 Hardware and Responsiveness

- Several users mentioned the performance of the application in relation to their hardware. High-end hardware (e.g., RTX 4090, Ryzen 7) was generally associated with better responsiveness, while users with older systems (e.g., GeForce 9300M) noted performance issues.

5.3.5 General Feedback and Improvements

- Positive remarks included the utility of the live preview feature and the professional appearance of the application.
- Some areas for improvement were mentioned, such as:
  - Improving the clarity of opacity computations.
  - Adding a Linux build to expand accessibility.

6 Conclusion

The Ray Caster application is generally seen as a helpful tool, particularly due to its interactive features and visual feedback. Users experienced significant improvements in their understanding of ray casting, especially when utilizing the live preview. However, some performance optimizations and content clarity improvements could enhance the experience further, especially for users with particularly old hardware.
The exploration of failed solutions, such as the use of a secondary virtual camera and geometric subdivision, demonstrated the importance of efficient GPU-based rendering. These initial attempts provided key insights into the limitations of CPU-bound approaches, ultimately guiding the development of more performant solutions, such as the use of 3D textures and custom shaders for real-time visual feedback.
The successful implementation of live volume ray casting in Virtual Ray Tracer demonstrated the effectiveness of GPU-accelerated techniques for real-time visualization of voxel grids. By leveraging 3D textures and custom shaders, the final solution provided enhanced depth perception and accurate lighting, ensuring that users could intuitively explore volumetric data. The integration of these features highlights the power of real-time ray casting for complex data sets and showcases the potential for future developments in interactive visualization and GPU-optimized rendering.

7 Future work

7.1 Acceleration Structures

To further enhance the capabilities of Virtual Ray Caster, a valuable extension would be the visualization of acceleration structures, such as octrees. Visualizing these structures within the ray caster would offer deeper insights into the voxel grid's organization, demonstrating how spatial partitioning divides the volume into hierarchical regions. The existing work on GPU-based ray casting and 3D texture sampling directly supports the creation of these interactive visualizations. With the current framework extended, users could see not only the rendered volumetric data but also how an octree recursively subdivides the voxel space into smaller sub-volumes. This feature would enable users to intuitively understand the role of acceleration structures in managing volumetric data, providing a visual and interactive way to explore how these structures affect ray traversal within complex datasets. Such an extension would make Virtual Ray Tracer an even more powerful educational tool for understanding both volume rendering and the underlying data structures that optimize it.

7.2 Efficiency-Related Optimisations

There are already some tools prepared to pre-compute surface normals for iso-surface lighting, but they are currently disabled due to lack of polish. In a future project they could be leveraged for greater efficiency of the shader.
Furthermore, empty space culling could be both implemented and visualized to the user, providing further educational value to the VRC.

7.3 Configurable Lighting Conditions for Isosurface

The user could benefit from being able to change the position of the light source and its intensity, as well as other lighting-specific variables such as the specularity and diffuse coefficient, in order to better view a volume while using the "first" compositing method. Exposing these variables to the user may therefore be of value.

Acknowledgements

I would like to thank both my girlfriend and my dog for staying by my side throughout this project, and motivating me during the times when it seemed impossible to find solutions to the problems I encountered. I would also like to thank my supervisors Steffen Frey and Jiri Kosinka for consistently providing me with invaluable feedback and guidance. I would have gotten lost at each step of the project without them.
Parts of this thesis were drafted with the assistance of Natural Language AI. However, all intellectual work, critical thinking, and final decisions presented in this thesis are entirely my own.

A User Study Questionnaire

1. Did you follow the RUG Scientific Visualization Course?
   (a) I am or was a student of the RUG Scientific Visualization course
   (b) I am or was a student in a computing science-related field
   (c) I have not been a student in computing science or a related field
2. If you followed the RUG Scientific Visualization course, do you think the Ray Caster application would be helpful to future students of the course?
   (a) Yes

(b) No

3. Rate your understanding of ray casting before using the Ray Caster application.

   (a) 1 (Never heard of it)
   (b) 2
   (c) 3
   (d) 4
   (e) 5
   (f) 6
   (g) 7
   (h) 8
   (i) 9
   (j) 10 (Could pass an exam on it)

4. Rate your understanding of ray casting after using the Ray Caster application, without the live preview.

   (a) 1 (Never heard of it)
   (b) 2
   (c) 3
   (d) 4
   (e) 5
   (f) 6
   (g) 7
   (h) 8
   (i) 9
   (j) 10 (Could pass an exam on it)

5. Rate your understanding of ray casting after using the Ray Caster application, with the live preview.

   (a) 1 (Never heard of it)
   (b) 2
   (c) 3
   (d) 4
   (e) 5
   (f) 6
   (g) 7
   (h) 8
   (i) 9
   (j) 10 (Could pass an exam on it)

6. Which of the following helped you understand ray casting?

   (a) The text tutorial that you can follow at the bottom left

(b) The visualization of the virtual camera shooting rays through the voxel grid
(c) The live preview
(d) Being able to experiment and try things out in the application
(e) The ray calculation window with the breakdown of a single ray

7. Rate the following aspects of the application:
   (a) Visuals
      i. 1 (Unusable)
      ii. 2
      iii. 3
      iv. 4
      v. 5
      vi. 6
      vii. 7
      viii. 8
      ix. 9
      x. 10 (Excellent)
   (b) User Interface
      i. 1 (Unusable)
      ii. 2
      iii. 3
      iv. 4
      v. 5
      vi. 6
      vii. 7
      viii. 8
      ix. 9
      x. 10 (Excellent)
   (c) Tutorial
      i.
      ii. 1 (Unusable)
      iii. 2
      iv. 3
      v. 4
      vi. 5
      vii. 6
      viii. 7
      ix. 8
      x. 9
      xi. 10 (Excellent)
   (d) Ray Calculation breakdown
      i. 1 (Unusable)
      ii. 2
      iii. 3

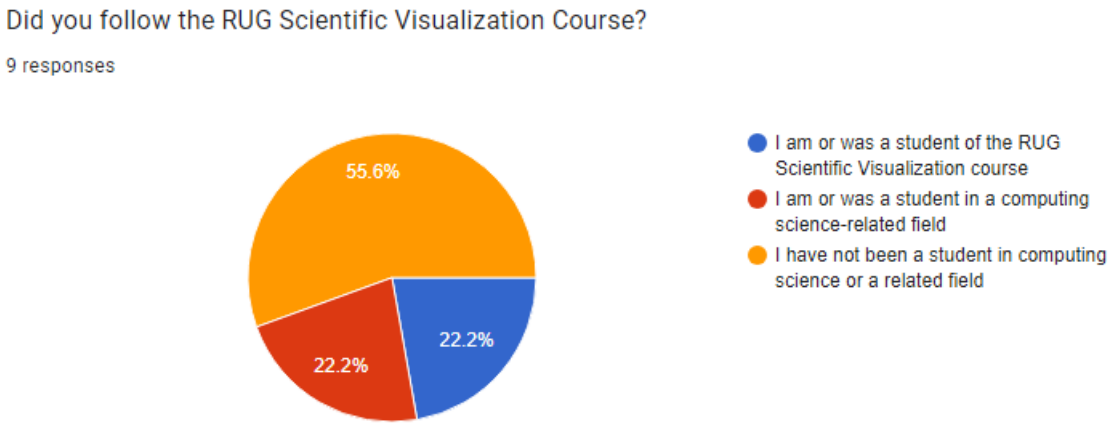
iv. 4
v. 5
vi. 6
vii. 7
viii. 8
ix. 9
x. 10 (Excellent)

(e) Responsiveness

i. 1 (Freezing/Crashing)
ii. 2
iii. 3
iv. 4
v. 5
vi. 6
vii. 7
viii. 8
ix. 9
x. 10 (Completely Smooth)

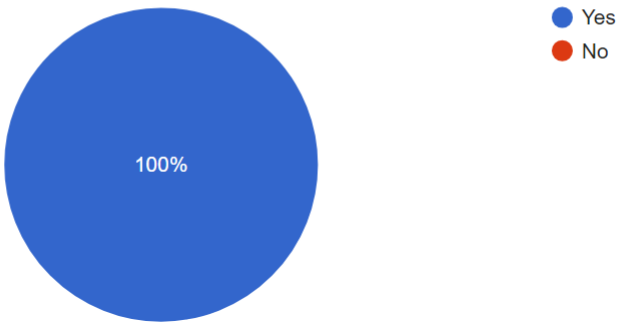
8. If you know, please note down what hardware you used to run the Ray Casting application.
9. Additional remarks.

B User Study Results



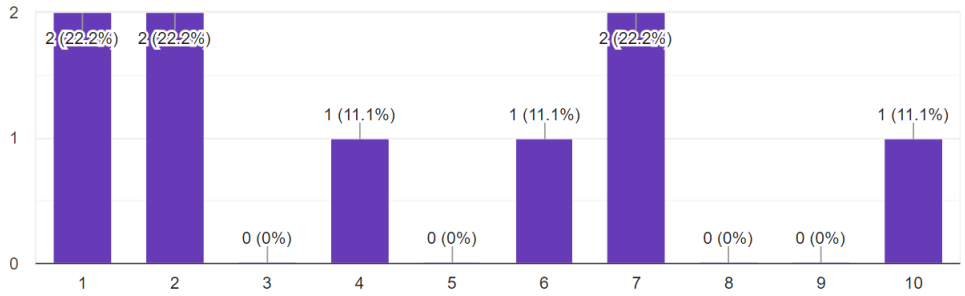
If you followed the RUG Scientific Visualization course, do you think the Ray Caster application would be helpful to future students of the course?

6 responses



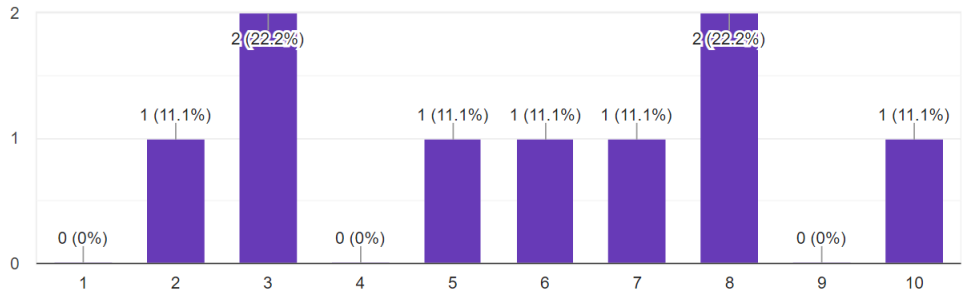
Rate your understanding of ray casting **before** using the Ray Caster application

9 responses



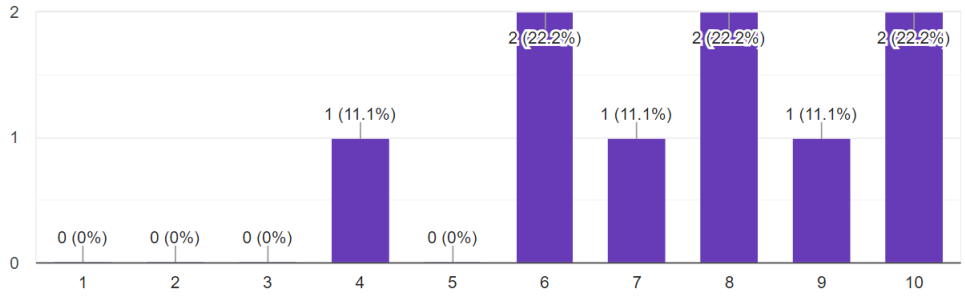
Rate your understanding of ray casting **after** using the Ray Caster application, **without** the live preview

9 responses



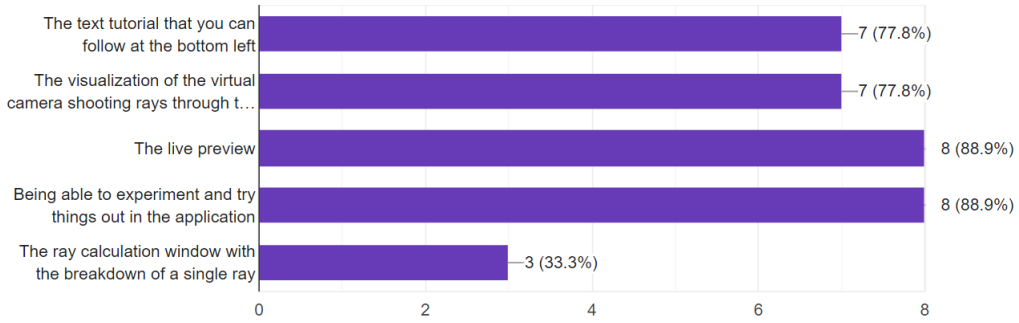
Rate your understanding of ray casting **after** using the Ray Caster application, **with** the live preview

9 responses



Which of the following helped you understand ray casting?

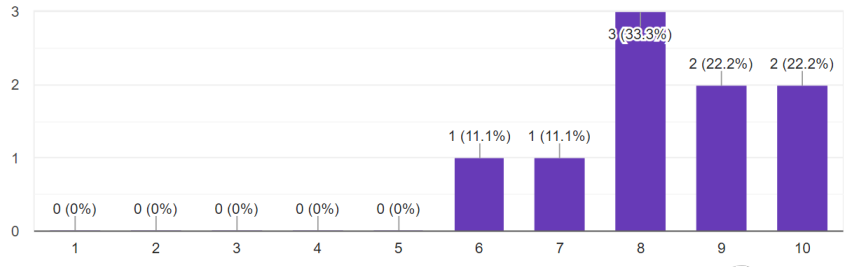
9 responses



Rate the following aspects of the application

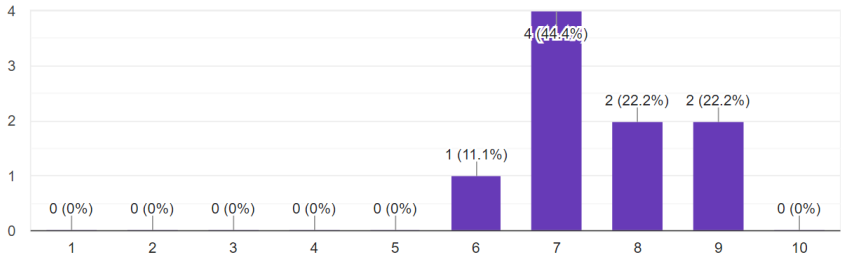
Visuals

9 responses



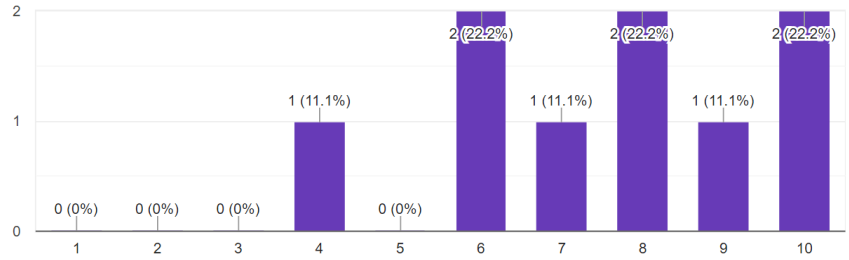
User Interface

9 responses



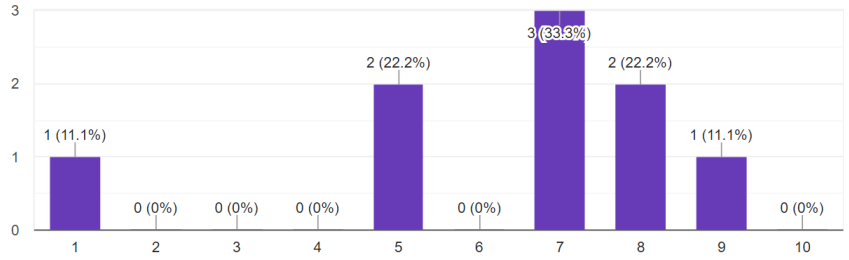
Tutorial

9 responses



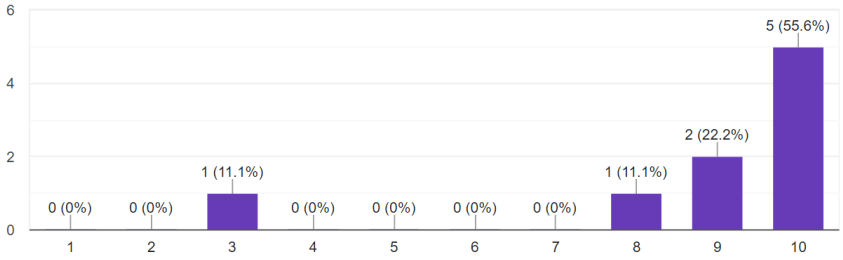
Ray Calculation breakdown

9 responses



Responsiveness

9 responses



References

[1] Unity documentation on blend function. Available at `https://docs.unity3d.com/Manual/SL-Blend.html`.

[2] Unity documentation on play mode function. Available at `https://docs.unity3d.com/Manual/ConfigurableEnterPlayMode.htmll`.

[3] Unity documentation on ztest function. Available at `https://docs.unity3d.com/Manual/SL-ZTest.html`.

[4] Unity, 2024. Available at `http://www.unity.com`.

[5] Wikipedia, Sep 2024. Available at `https://en.wikipedia.org/wiki/Ray_tracing_(graphics)#/media/File:Ray_trace_diagram.svg`.

[6] Jake A Russell. An interactive web-based ray tracing visualization tool. *Undergraduate Honors Program Senior Thesis. Department of Computer Science, University of Washington*, 2, 1999.

[7] Lukke Van der Wal. Virtual ray tracer: Ray casting support. *Bachelors' Thesis, Rijksuniversiteit Groningen*, 2024.

[8] Chris S. van Wezel, Willard A. Verschoore de la Houssaije, Steffen Frey, and Jiří Kosinka. Virtual ray tracer 2.0. *Computers & Graphics*, 111:89–102, 2023.

[9] Willard A. Verschoore de la Houssaije, Chris S. van Wezel, Steffen Frey, and Jiri Kosinka. Virtual Ray Tracer. In Jean-Jacques Bourdin and Eric Paquette, editors, *Eurographics 2022 - Education Papers*. The Eurographics Association, 2022.

[10] Wikipedia, Mar 2023. Available at `https://en.wikipedia.org/wiki/Volume_ray_casting`.