Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Fakultät für Mathematik und Informatik
Institut für Informatik
Mensch-Computer-Interaktion

Bachelorarbeit

zur Erlangung des Grades
Bachelor of Science (B.Sc.)
im Studiengang Games Engineering
an der Universität Würzburg

Visualization of Raytracing

---

vorgelegt von
Pirmin Pfeifer
Matrikelnummer: XXXXXXX

am September 10, 2023

Prüfer: Prof. Dr. Sebastian von Mammen
Betreuer: Dr. Matthäus G. Chajdas
Lehrstuhl für Informatik IX
Julius-Maximilians-Universität Würzburg

Zusammenfassung

deutsche Zusammenfassung - Raytracing erfreut sich in der Spieleentwicklung zunehmender Beliebtheit. Bei der Fehlersuche in Raytracing-Algorithmen stellt sich die Frage "Welche Teile der Szene werden von wie vielen Strahlen durchquert?". In dieser Arbeit wird *RayVis* vorgestellt, ein Werkzeug zur Visualisierung von großen Strahlendatensätzen in einer 3D-Szene. *RayVis* bietet drei verschiedene Visualisierungen zur Beantwortung von Fragen über die Dichte und Position von Strahlen. Die erste Visualisierung zeichnet die Strahlen als 3D-Mesh in die Szene. Die zweite visualisiert die Strahldichte und -richtung in einem Vektorfeld. Und die letzte Visualisierung stellt die Strahldichte als Volumen dar.

Abstract

english abstract - Raytracing is rapidly gaining popularity in game development. When debugging raytracing algorithms, one question to be looked at is "Which parts of the scene are traversed by how many rays?". This thesis presents *RayVis*, a tool for visualizing large rays data sets in a 3D scene. *RayVis* provides three different visualizations for answering questions about the density and location of rays. The first visualization draws the rays as 3D meshes into the scene. The second one visualizes the ray density and direction in a vector field. And the last fone visualization the ray density as volume.

Contents

List of Figures

1 Introduction

This thesis presents *RayVis*, a tool for visualizing large amounts of rays in a 3D scene. *RayVis* provides three different visualizations for answering questions about the density and location of rays. The effectiveness of the visualizations was validated with an expert user study.

1.1 Background

Raytracing is the method of calculating intersection points between a ray and all objects in a scene. It can be used to solve global questions in 3D rendering. For example, shadows can be calculated by checking if a ray cast from a surface to a light source intersects other objects before hitting the light source. This approach can be used to calculate all kinds of physical light effects, by tracing the path of the light through the scene and calculating the interactions of light rays with objects based on physical principles.

These global light effects, like illumination, shadows and reflection, are very complex to implement and quite performance-intensive on traditional rasterization render pipelines. Many modern video games, that aim for realistic real-time graphics, utilize the global approach to rendering, provided by raytracing, for realistic shadows or reflections (Avalanche Software, 2023; CAPCOM Co., Ltd., 2023; CD Projekt RED, 2020; NVIDIA, 2023). This is made possible through recently released real-time rendering hardware, like Microsoft Xbox Series X, Sony PlayStation 5, Nvidia RTX 40 series and AMD RX 7000 series, which support hardware acceleration for raytracing.

With more developers working on real-time raytracing applications the demand for raytracing debug tools is rising. When debugging raytracing, two questions are particularly interesting. Which rays are cast to create the color of a single pixel? And, Which parts of the scene are traversed by how many rays? The first question,

which is already solved by a visualization tool kit developed by Gribble et al., 2012, is encountered when individual pixels produce unexpected results. The second is primarily asked when the performance of the scene is investigated.

1.2 Goals

Since the inspection of a single ray is already solved, the focus of this thesis lies in answering spacial questions about large amounts of rays. When looking at the performance of ray calculations, it can be helpful to locate areas with high or low ray densities. The explicit questions this thesis aims to answer, with visualizations, are "Where is the ray density particularly high?", "Where is the ray density particularly low?" and "Which effect was calculated by the visualized rays?". Therefore, the goal is to create visualizations of many rays and their densities in 3D space.

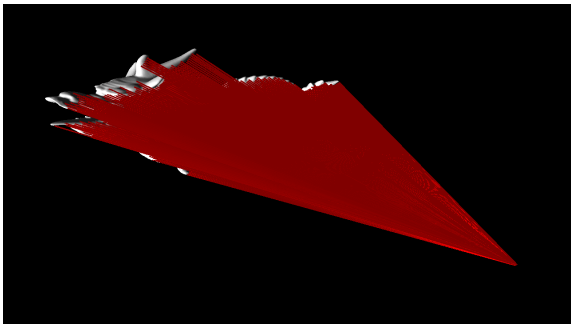
1.3 Methodology

To achieve this, first different visualization approaches are investigated and chosen for implementation. Afterward, ray and scene data is acquired, and a loader is written. Then a renderer for the scene geometry is set up. For each chosen visualization approach, necessary rendering data is determined and an algorithm to calculate that data from the given ray data is developed and implemented. Then either a shader is written to render the data structure or the data structure is already a mesh that can be rendered by the rendering pipeline used for scene rendering. An extensive user interface is added to give control over parameters used in data generation and real-time visualization rendering. Finally, the tool is tested by raytracing experts in an expert-user study, to validate if the research questions can be answered with the visualizations.

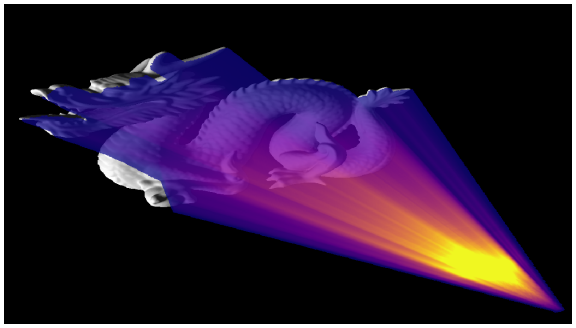
1.4 RayVis - Overview

RayVis provides three different visualizations for large ray data sets. The first visualization is called *Ray Mesh*. A line is drawn from the start point to the hit- or end-point for each ray (See Fig. 1.1). For large ray data sets, only a subset

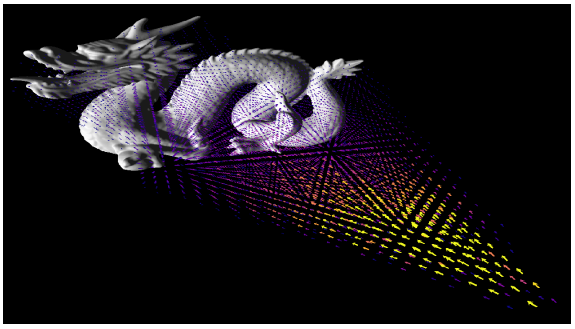
of rays can be rendered as mesh, due to the performance of Direct3D raytracing renderer. The other two visualizations both visualize the ray density of the scene. The density volume visualization samples the volume data with raymarching, along the primary render view ray, for each pixel (See Fig. 1.1). The accumulated ray count is then layered on top of the scene and mapped into the inferno color map, using a user-specified value range. The last visualization is a vector field, that samples the density and direction of this volume in a regular grid. For each sample cell, an arrow, pointing in the average ray direction of rays in the cell, is drawn and scaled by density value (See Fig. 1.1).



(a) *Ray Mesh* visualization for primary ray calculation of the Stanford dragon.



(b) Density volume visualization for primary ray calculation of the Stanford dragon.



(c) Vectorfield visualization for primary ray calculation of the Stanford dragon.

Figure 1.1. Different visualization modes of *RayVis*. The details of each visualization are described in Section 3.3.

2 Related Work

This chapter lays out the fundamentals of Raytracing and its current advancements in video games. It also gives an overview of existing visualizations of the raytracing algorithm. Additionally, the basics of the algorithms and methods used for the visualizations are presented.

2.1 What is Raytracing?

Raytracing is the method of calculating intersection points between a ray and all objects in a scene (Glassner, 1989; Marrs et al., 2021). A ray is defined by its origin point, the direction vector it is extending in and, optionally, a maximum ray length. The raytracing algorithm can be used to solve global questions in the rendering of three-dimensional geometry. For example, global shadows can be calculated by checking if a ray cast from a surface to a light source intersects other objects before hitting the light source. This approach can be used to calculate almost all physical light effects, by tracing the path of the light through the scene and calculating the interactions of light rays with objects based on physical principles. Raytracing finds also application in physics- and game engines to find objects in a 3D scene (Epic Games, 2023; Unity Technologies, 2023).

Recently released video games, like "Cyberpunk 2077" by CD Projekt RED, 2020, "A Plague Tale: Requiem" by Asobo Studio, 2022 or "Resident Evil 4" by CAPCOM Co., Ltd., 2023, utilize raytracing for reflections, shadows or global illumination. This is made possible by the latest generation of real-time rendering hardware, like Microsoft Xbox Series X, Sony PlayStation 5, Nvidia RTX 40 series and AMD RX 7000 series, which provides hardware acceleration for raytracing. This raytracing hardware is not yet powerful enough to render the complex game scenes completely at interactive frame rates. Therefore a combination of traditional rasterization pipelines and raytracing effects is used in most games, that currently support raytracing.

2.2 Existing Visualizations for Raytracing

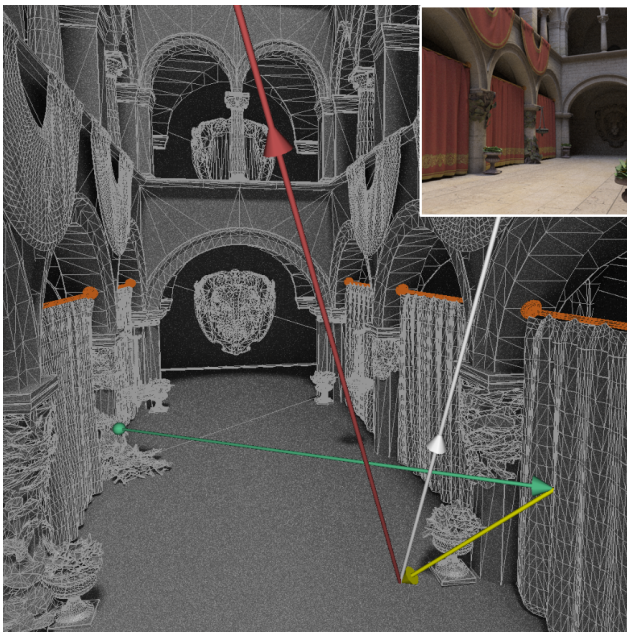


Figure 2.1. Ray state visualization in rtVTK, showing the rays calculated for a single pixel in the computation of a high-quality image of the Crytek Sponza scene. (Source: Gribble et al., 2012)

The question "Which rays are cast to create the color of a single pixel?" is already solved by the Ray Tracing Visualization Toolkit (rtVTK), developed by Gribble et al., 2012. This Toolkit can be integrated into raytracing algorithms, to interactively visualize all rays cast to render a single pixel. Each ray is rendered as an arrow into the 3D scene, making it easier to understand how the scene is traversed for the rendering of the investigated pixel (See Fig. 2.1). It helps with spotting errors in the raytracing algorithm, like rays missing objects they should hit or rays cast in the wrong direction. The Ray Tracing Visualization Toolkit only handles the inspection of rays used for a single pixel but is not designed for the visualization of all rays used to calculate an image. Therefore, it can not be used to answer the research questions of this thesis.

2.3 What is Volume Data?

Kaufman, 1996, describes volume data as 3D entities which contain information. These do not consist of surfaces and edges. And may be too voluminous to be represented geometrically. For volume data a sampling function can be defined, that computes the volume information for any given partition of the space the volume spans. With the sampling function, a grid of cells (voxels) of uniform or non-uniform

size can be generated. Each voxel then holds discrete data describing the volume information in it. The volume data can have many areas where all sampled data is zero or null. Such volumes are called sparse volumes. For these volumes, different algorithms can be used to cut down on memory usage, compute or lookup time of the data set (Museth, 2013; Zhang et al., 2011; Zhu et al., 2007). The volume data used in *RayVis* is mostly sparse. Therefore a simple two-level hierarchy data structure is used to reduce the memory usage of the volume data.

2.4 What is Raymarching?

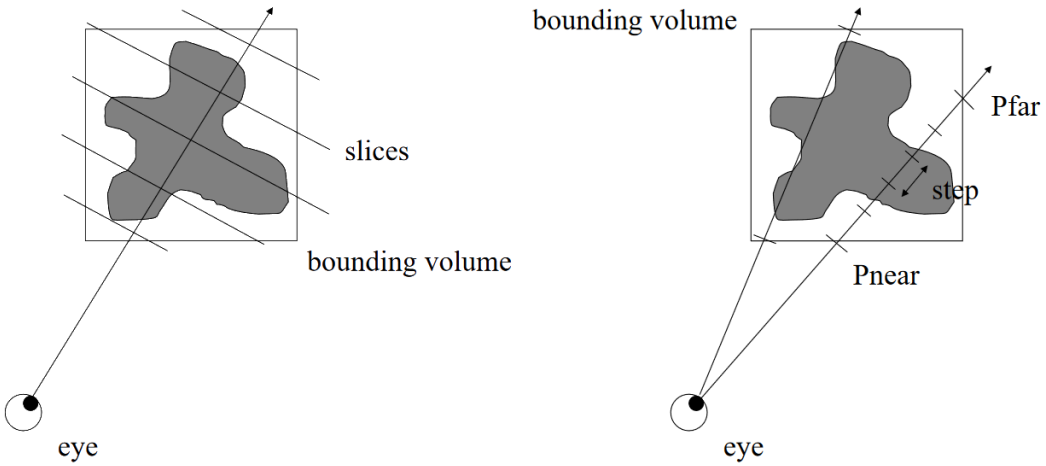


Figure 2.2. Volume is sampled along a ray. Left: Sample steps are at fixed intervals. Right: Fixed number of sample steps. (Adapted from: Green, 2005)

Raymarching is the technique of sampling a volume along a ray (See Fig. 2.2). For each pixel, a ray is marched through the volume, sampling it at different points along the ray and accumulating the volume values. Either the step size is a fixed distance or the distance is calculated by dividing the intersection distance of the volume and the ray by a fixed number of steps. In computer graphics, raymarching is used to render volumetric effects, like flames, explosions, clouds or fog (Bittner, 2020; Green, 2005; Wronski, 2018). *RayVis* uses raymarching to render ray density in the volume visualization mode.

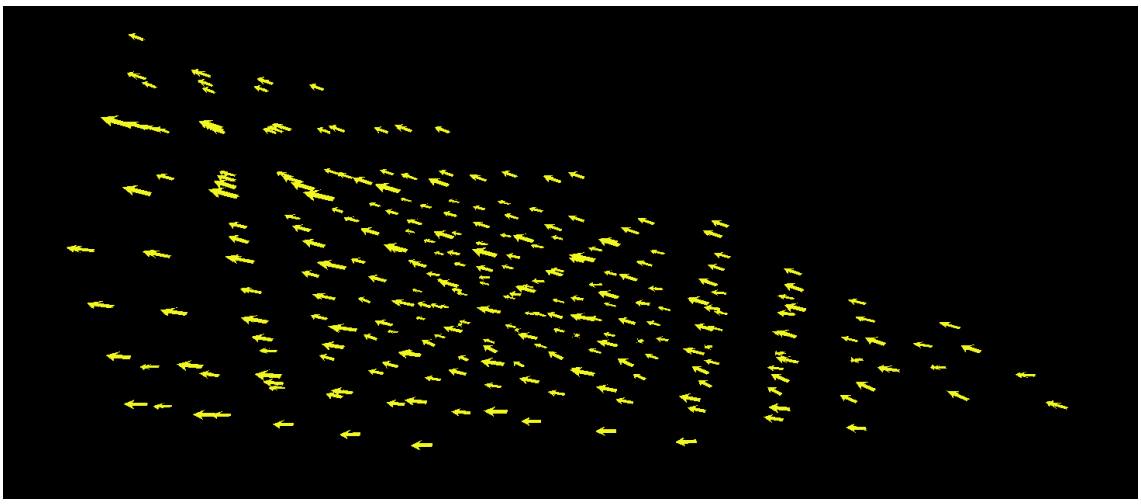


Figure 2.3. Small 3D vector field rendered in *RayVis*. The directional data of each voxel is visualized through the orientation of the arrows.

2.5 What is a Vector Field?

The term vector field originates from vector mathematics. It describes a mapping of each point in space to a direction and optionally a strength (Galbis & Maestre, 2012). This can be formulated as a function. Therefore, vector fields can, in the context of this thesis, be considered as volume data that hold directional information. A vector field can be rendered by drawing an arrow mesh in regular intervals. And then setting the rotation of each arrow to represent the sampled direction of the vector field (See Fig. 2.3).

3 Methodology

This chapter specifies the data format used as input. Presents the scenes and effects used in the examples. Lays out the details of the data structures and shader implementations. Furthermore, the user interface of *RayVis* is explained.

3.1 Dataformats & Datastructures

RayVis uses its own data format for saving and loading. Each file contains one or multiple ray data sets and scene geometry. The original ray data is provided by Advanced Micro Devices (AMD) in an internal format and is converted into the *RayVis* data format.

3.1.1 Sourcedata

The source data is provided by AMD in an internal format. Because this data is in a format available to the research community, all data is converted into a format native to *RayVis*. These *RayVis* files can also be created from any arbitrary ray and scene data sets, produced by any raytracing algorithm.

The ray data sets originate from different graphics effects, calculated on these scenes. The primary ray data contains the rays cast from a camera to render the scene into a 2D image buffer. The shadow rays are cast from the hit points of the primary rays in the direction of a directional light source. All rays that hit geometry on their way to the light source are in the shadow and shaded accordingly. For the reflection calculation, the original ray direction of the primary rays is reflected on the normal vector of the hit primitive and cast from the hit point. Ambient occlusion is sampled with a set amount of rays, cast in random directions on the hit normal hemisphere with a short maximum ray length (Marrs et al., 2021).

The model of the Stanford dragon, which is used as an example model in the images of this thesis, was taken from the Stanford 3D Scanning Repository and originates from Curless and Levoy, 1996. Additionally, the UE4 Sun Temple scene (created by Epic Games, 2017) is used as a sample scene for this thesis.

3.1.2 RayVis Dataformat

RayVis loads and saves its file in a chunk format utilizing the radeon data file (RDF) library by AMD, 2023. Each *RayVis* file contains one scene chunk and at least one ray data chunk. The ray data chunk contains all rays for one step of an effect calculation. It starts with the `RayTraceHeader struct`, followed by the binary dump of all `Ray structs` belonging to this ray data object. The details of the header and ray data structures can be seen in Listing A.1. The scene data chunk provides the scene geometry in a custom format. The `SceneChunkHeader struct` is used as the header data and provides the mesh and root node count of the scene (See Listing A.2). The data of the chunk contains as many `MeshPrefix structs`, and their follow-up data, as specified by `meshCount`. This mesh data then is followed by as many `RootNodePrefix structs`, and their follow-up data, as specified by `rootNodeCount`. To convert a given scene and ray data set, a chunk file has to be created with the RDF library. Each ray data set has to be saved in its chunk. The chunk data has to comply with the format specified in the appendix (A.1 & A.2). The file finally has to be saved with the `.rayvis` extension, to be recognized by *RayVis*.

3.1.3 Ray Density Volume Data

The ray density volume data is used for two of the visualizations provided by *RayVis*. The data is sampled into a voxel grid. The voxel size is uniform and can be specified in the user interface. This voxel grid is organized into cubic non-overlapping voxel groups, which are called chunks (See Fig. 3.1). Only chunks that contain any non-zero data have to be stored, greatly reducing the memory consumption of the data structure. Which enables to use of smaller voxels to increase the detail of the visualization. Each voxel stores the number of rays that intersect with the voxel, which is referred to as ray density, and the averaged direction of the intersecting rays.

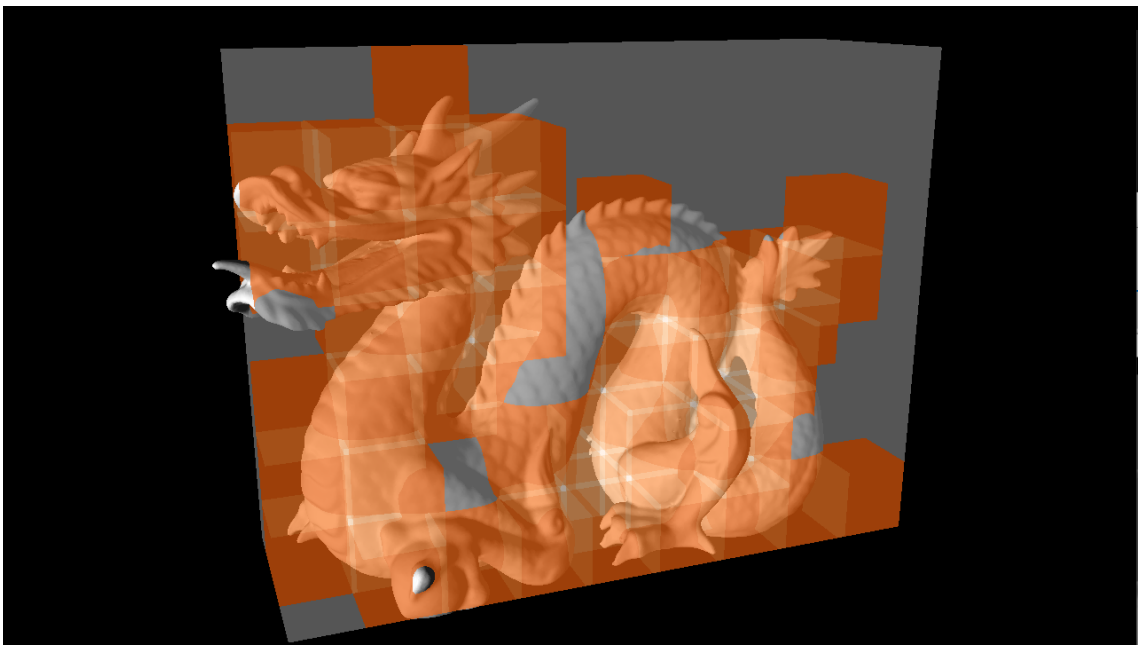


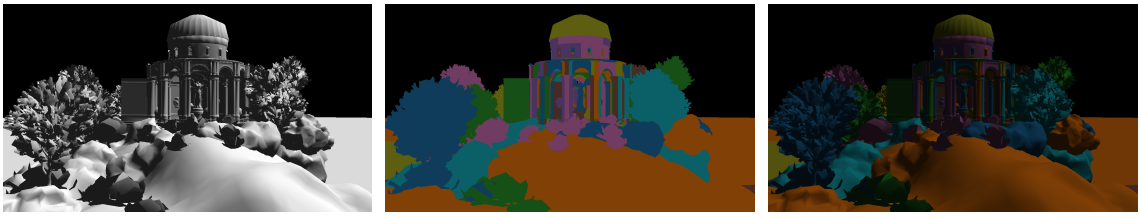
Figure 3.1. Visualization view of voxel grid chunks in *RayVis*. The grey cube represents the first level bounding box of the data structure. The orange cubes visualize the second-level chunks hit by the pixel's primary ray. The brightness shows the number of chunks that are hit. Each chunk is rendered with one voxel more in all directions, to better visualize edges and increase clarity.

To calculate the voxel grid for a given set of rays, multiple steps are needed. First, an axis-aligned bounding box, spanning around all rays, is created. This bounding box then is expanded to be dividable into whole chunks. In the next step a voxel traversal algorithm, developed by Amanatides and Woo, 1987, is used to find all chunks in the bounding box that are hit by any rays. Afterward, the voxel data in each chunk is calculated. This is done by intersecting each ray with the chunk. For each hitting ray, the intersected voxels are calculated, using the same algorithm as in the previous step. The voxels ray count is incremented for each ray passing through and the ray directions are accumulated. Finally, the ray direction of each voxel is normalized. These last two steps are parallelized, to cut down computation time significantly.

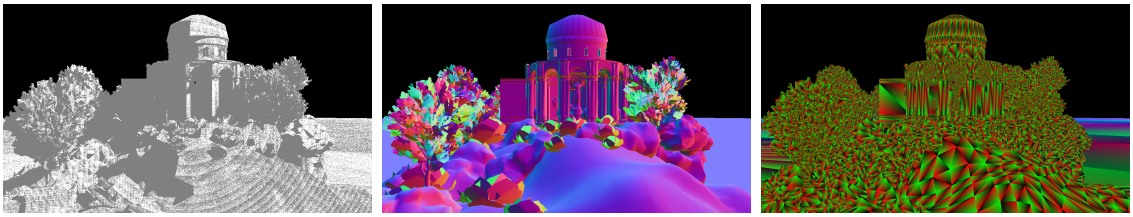
(a) Smooth grayscale **(b)** Instance color **(c)** Smooth instance color

(d) Raytraced Shadows **(e)** Normals **(f)** Barycentrics

Figure 3.2. Shading modes on the Stanford dragon. Each shading mode either visualizes a property of the mesh or shades the scene to contrast the visualizations.



(a) Smooth grayscale **(b)** Instance color **(c)** Smooth instance color



(d) Raytraced Shadows **(e)** Normals **(f)** Barycentrics

Figure 3.3. Shading modes on the UE4 Sun Temple scene. Each shading mode either visualizes a property of the mesh or shades the scene to contrast the visualizations.

3.2 Rendering

RayVis uses Direct3D Raytracing (DXR) Tier 1.1, developed by Microsoft, 2020, for rendering the scene geometry and the mesh components of the visualizations. For the volume visualization a separate compute-shader is used. *RayVis* can render the scene model in six different shading modes or exclude the scene geometry completely from rendering (See Fig. 3.2 & 3.3). The default shading mode renders the model with the defuse part of the phong shading model (Bui-Tuong, 1975) and is called

smooth grayscale. Its main focus is providing a clean and contrast-rich image for the visualizations, without obscuring the 3D properties of the scene. In instance color mode all instances are assigned a color from a table. The rendered color of a hit mesh is then determined through the color assigned to its parent instance node. This enables the viewer to differentiate the instances of the DXR acceleration structure. The smooth instance color mode displays the multiplication of the two previously described shader models. This mode creates a compromise between the clarity of the smooth shading mode and the additional information from the instance rendering mode. To utilize the DXR renderer global directional shadows are also implemented with the shadow mode. For each primary ray, that hits any primitive, a ray is cast in the light direction. The image shows if the shadow ray calculation resulted in a hit. Additionally, *RayVis* can also visualize the mesh normals in world space or the barycentric coordinates of the mesh, as shown. For the primary ray generation of all modes, a simple perspective pinhole camera approach is used.

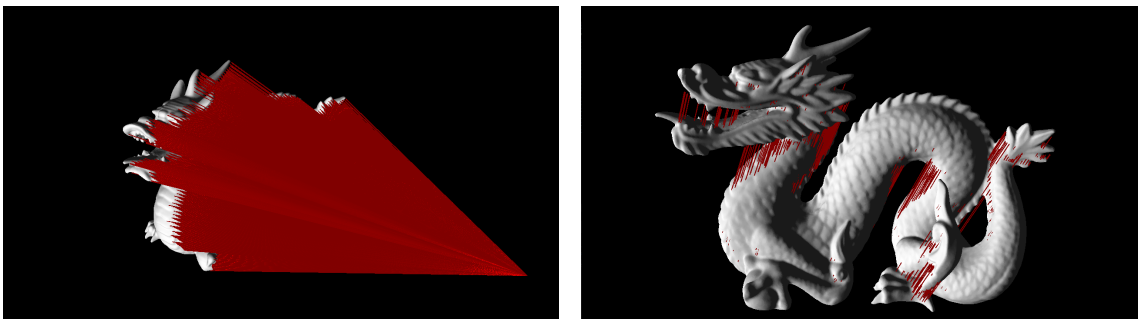
3.3 Visualizations

RayVis provides three visualizations, a *RayMesh*, a Vectorfield and a Volume visualization. The rays visualized in all three can be filtered to only include hitting or missing rays. Each mode provides a different view of the ray data.

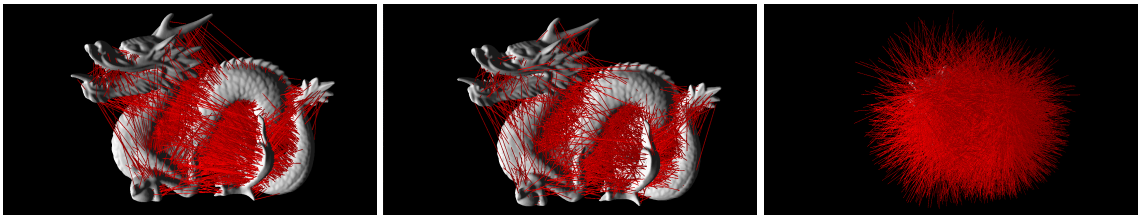
3.3.1 Ray Mesh

The *Ray Mesh* visualization is a naive approach for the visualizations of many rays. For each ray, a line mesh is generated by scaling and rotating two orthogonal planes to span from the ray-start- to the ray-end-point. These meshes are given an instance color, chosen by the user, and are rendered in the smooth instance color mode (See Fig. 3.4 & 3.5). For some effects, like hitting primary rays or missing ambient occlusion rays, the visualization can get very cluttered, reducing its readability.

When the data set contains a hundred thousand to a million rays and their meshes are overlapping, an interactive frame rate can no longer be guaranteed. Especially because Direct3D raytracing doesn't handle overlapping triangles, with every large extends' in one dimension, very well. In these cases, only a fraction of all rays can

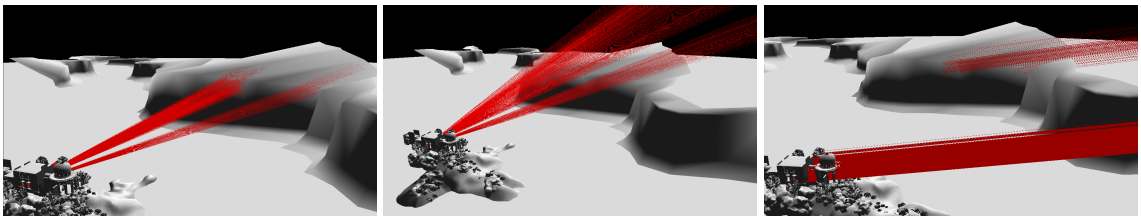


(a) Primary rays (only hit) **(b)** Shadow rays (only hit)

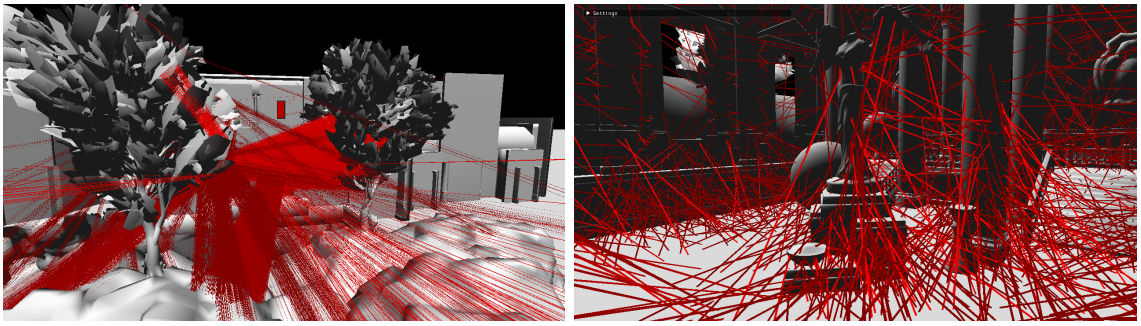


(c) Reflection rays (only hit) **(d)** Ambient occlusion rays (only hit) **(e)** Ambient occlusion rays (only miss)

Figure 3.4. *Ray Mesh* visualization for different effects on the Stanford dragon. For each ray, a line is rendered. The origin of the primary rays can be found where the rays converge. The parallel nature of the shadow rays is also seen. For some effects, the visualization can get very cluttered.



(a) Primary rays (only hit) **(b)** Primary rays (only miss) **(c)** Shadow rays



(d) Reflection rays (only hit) **(e)** Ambient occlusion rays (only hit)

Figure 3.5. *Ray Mesh* visualization for different effects on the UE4 Sun Temple scene. For each ray, a line is rendered. The hitting reflection rays have not been filtered to only contain a fraction of the rays.

be rendered by this visualization. The stride of the rendered rays can be customized to give more control over the performance and ensure interactiveness.

3.3.2 Vector Field

(a) Primary rays (only hit) **(b)** Shadow rays (only hit) **(c)** Shadow rays (only miss)

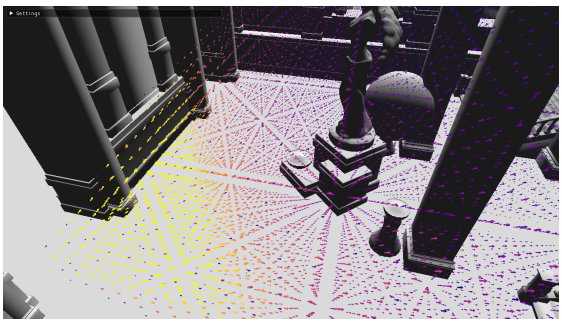
(d) Reflection rays (only hit) **(e)** Ambient occlusion rays (only hit) **(f)** Ambient occlusion rays (only miss)

Figure 3.6. Vector field visualization for different effects on the Stanford dragon.

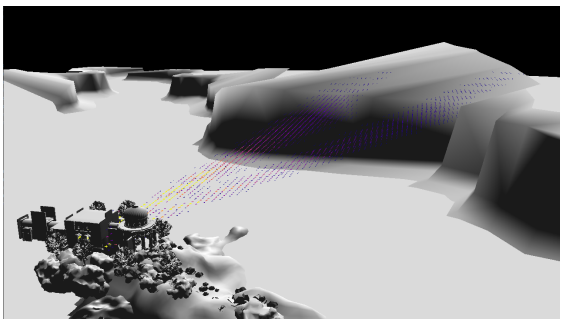
The vector field visualizes the volume data through arrows drawn in the scene. The arrow's direction represents the average ray direction in its area and the ray density through color and scale (See Fig.3.6 & 3.7). This is the only visualization of *RayVis* that preserves the directional data of the original ray data set. For the calculation, a sample size, which needs to divide the chunk size without a remainder, is selected. Then the chunks of the volume data structure are divided into small cubes of voxel, with side lengths equal to the sample size. The ray count and ray direction of all voxels in each cube is accumulated and averaged. For each sampling cube, an arrow is instantiated and rotated accordingly, so it points in the average ray direction. The color of the arrow is determined by the ray count and represented by the inferno color pattern. Additionally, the arrows can either be scaled by the ray count or by the inverse of the ray count.

3.3.3 Density Volume

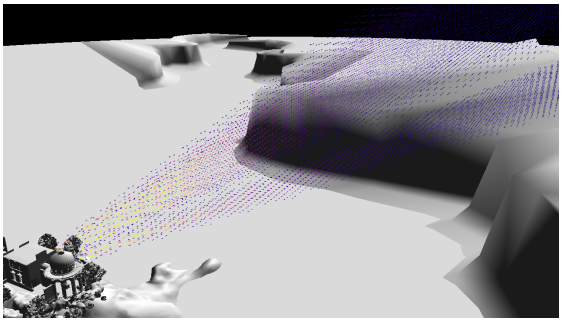
The density volume visualizes the ray count of the volume data structure with ray-marching. Data has to be transformed and uploaded to the GPU, before rendering



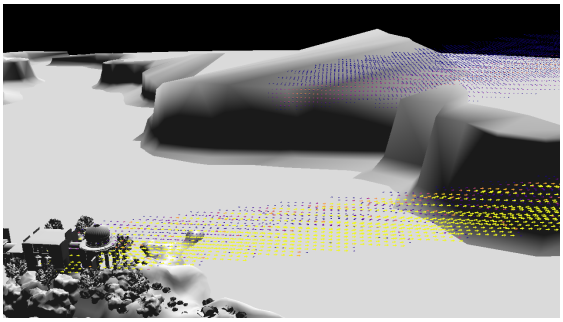
(a) Primary rays (inside)



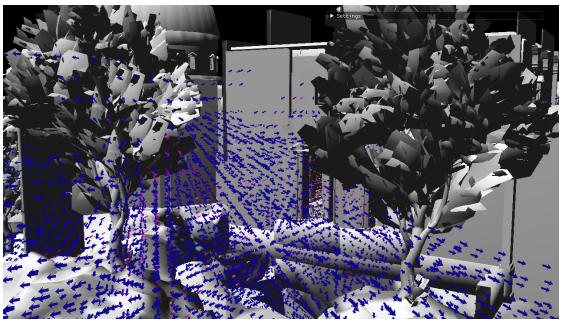
(b) Primary rays (only hit - outside)



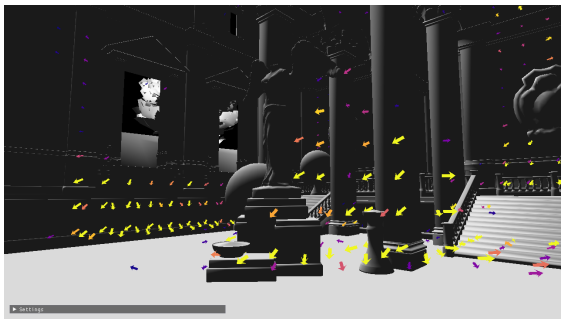
(c) Primary rays (only miss - outside)



(d) Shadow rays (outside)



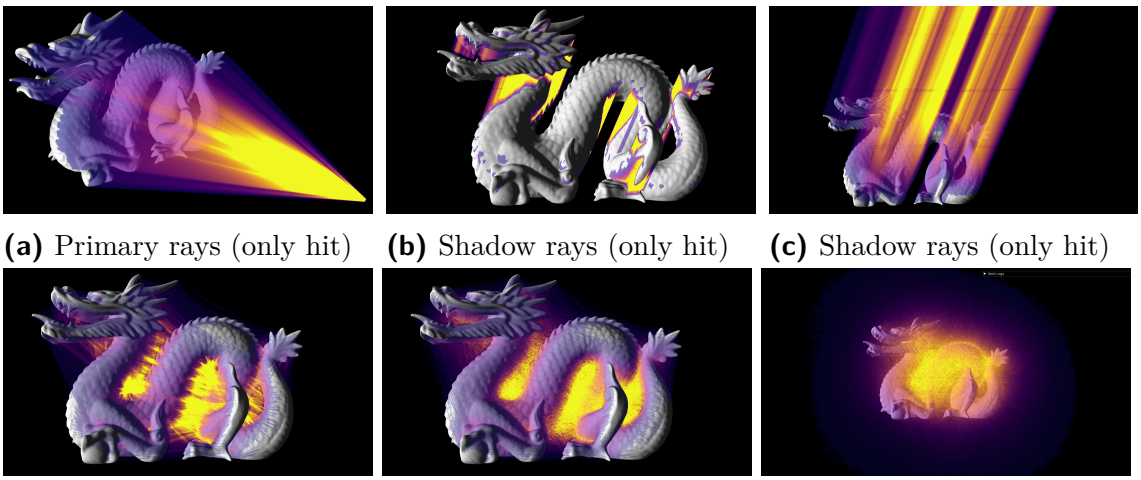
(e) Reflection rays (only hit - outside)



(f) Ambient occlusion rays (only hit - inside)

Figure 3.7. Vector field visualization for different effects on the UE4 Sun Temple scene. In Figure 3.7a a spot with high ray density can be seen where the origin of the rays was in the calculation. In Figure 3.7b, 3.7c and the overall location of the rays can be seen. The shadow rays, in Figure 3.7d, cast from the sun temple are denser than the rays cast for the terrain behind it. In Figure 3.7e the arrow scale is relative to the inverted density. In the last Figure 3.7f arrows have been filtered to only show above a threshold.

the volume. First, the highest ray count in the data has to be found. Then all ray counts are converted into a floating point value between zero and one, where one is representing the highest value in the volume. Also, the chunks are expanded by one voxel in each direction. These new voxels are filed with the downscaled average ray count of their neighbors in the original data. This is done to smooth out the edges between chunks in the volume renderer. Finally, the transformed chunks are

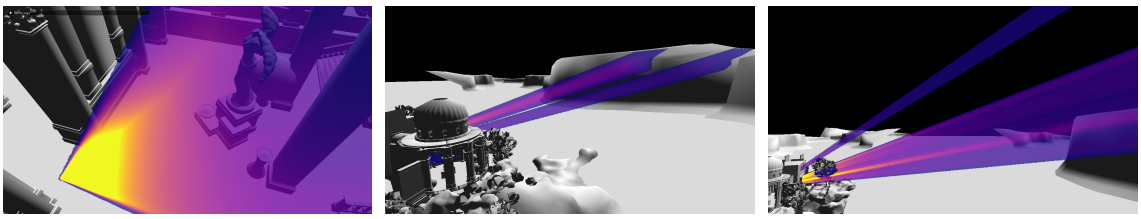


(a) Primary rays (only hit) **(b)** Shadow rays (only hit) **(c)** Shadow rays (only hit)

(d) Reflection rays (only hit) **(e)** Ambient occlusion rays (only hit) **(f)** Ambient occlusion rays (only miss)

Figure 3.8. Density volume visualization for different effects on the Stanford dragon. The bright spots of the volume show areas with high ray density. The shape of the ray geometry also can be made out in Figures.

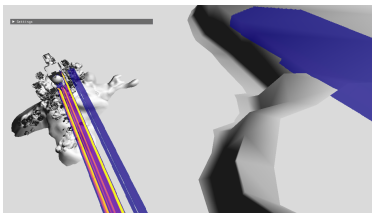
uploaded to the GPU as three-dimensional textures in a Direct3D `UNORM` Format. The chunk's minimum and maximum bounds are also uploaded to the GPU. In the compute shader used to render the volume visualization, each ray is checked against the bounding box of the chunks. The maximum ray length, in this step, is reduced based on the hit distance from the scene rendering. Then for all chunks the `TraceVolume` function is called. The code for the function can be found in the appendix A.3. This function first checks, if the ray hits the given chunk bounds. If it does, the data in the 3D chunk texture is accumulated using, an implementation of the raymarching algorithm. The step size of the algorithm can be specified by the user to fractions of the `cellSize`. After all the chunks are checked with the function, the results are accumulated and clamped. Finally, the result is mapped to a color, using the inferno color map, which is layered over the scene render. The final render can have small artifacts at the edges of adjacent chunks. This is caused by calculating the raymarching algorithm independently for each chunk (See Fig. 3.8 & 3.9).



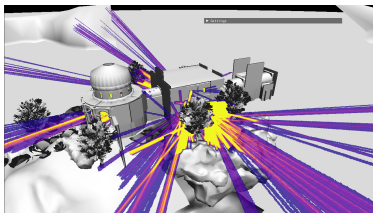
(a) All primary rays (inside)

(b) Primary rays (only hit - outside)

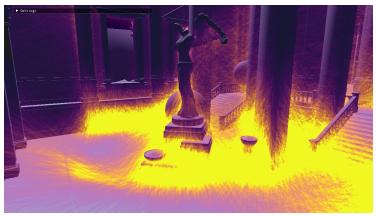
(c) Primary rays (only miss - outside)



(d) Shadow rays (outside)



(e) Reflection rays (only miss - outside)



(f) Ambient occlusion rays (only hit - inside)

Figure 3.9. Density volume visualization for different effects on the UE4 Sun Temple scene. The bright spots of the volume show areas with high ray density. The diverging of the primary and reflection rays can be seen through shape of the volume.

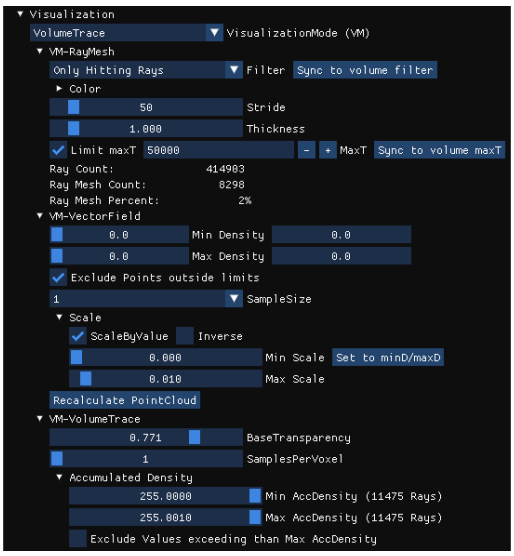
3.4 User Interface

RayVis provides an extensive graphical user interface (GUI) for controlling the visualizations and rendering. The state of the GUI and *RayVis* can be saved and loaded as JSON files. A command line interface (CLI) for specifying the source location and the initial configuration file is also present.

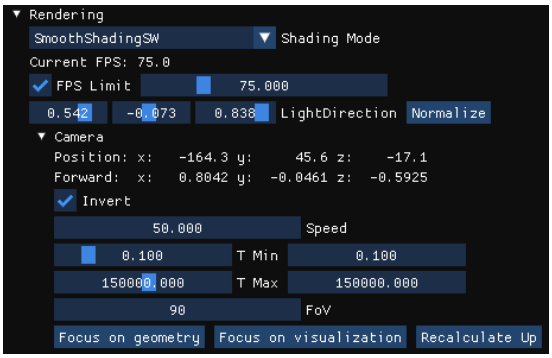
3.4.1 Graphical User Interface

The GUI of *RayVis* is made with DearImGui (Cornut & contributers, 2023) and provides various controls over the source file, renderer, visualizations and data generation. The toolbar can be used to load and save source and configuration files. The GUI is composed of four sections. In the first section, the current ray trace can be selected and the scene can be scaled up, to reduce floating point errors in the calculation of the visualizations (See Fig. 3.10a). The current source file location is also shown. The next section gives control over the volume data calculation. A ray filter can be selected and the chunk and voxel size can be changed. In some ray data sets the missing rays contain hit distance data, which is minimally smaller

(a) Setting UI with toolbar. The statistics show how much memory is currently used by the volume of data. Parameters of the volume data can be changed and a recalculation can be started.



(b) Visualization section of the GUI. It exposes the rendering parameters of all three visualizations.



(c) Rendering section of the GUI. Shading mode and parameters can be changed. Also, the camera settings can be modified.

Figure 3.10. Graphical User Interface (GUI) Overview. The user can view different statistics, switch and control the visualizations and set parameters for the rendering.

than the maximum distance of the ray. To accommodate for that error a miss detection tolerance can be set. Because the calculation of the volume data can take multiple seconds, it is only done if the user requests a recalculation. The user is also informed about the current and estimated memory consumption of the volume data structure and its generation. The main section of the GUI controls the visualizations of *RayVis*. All visualizations can be edited in this GUI, even if not currently selected for rendering. The visualization selection is either done with the combo

menu item or with hotkeys. In the *RayMesh* and Volume subsections, all settings are immediately applied to the visualization. The Vectorfield needs to be recalculated the same way as the volume data. Multiple parameters can be controlled for each visualization, as shown in Figure 3.10b. The last section exposes the settings of the renderer, which includes the camera (See Fig. 3.10c).

3.4.2 Command Line Interface

RayVis has no mandatory command line arguments, but it provides a CLI to set different parameters while starting the application. The initially loaded `.rayvis` file can be specified with the `-input` (`-i`) parameter. The default `config.json` location can also be overridden, using the `-config` (`-c`) parameter. The export functionality, of `.rayvis` files, is not needed for analyzing the ray data sets. Therefore, it has to be enabled with the `-enableExport` flag.

4 Evaluation

After the different visualizations are implemented, raytracing experts at AMD evaluate which visualizations are most valuable for solving the research question. The reception of the tool was mixed, because of usability issues.

4.1 Methodology

The users are tasked with answering questions for ray data of the Cornell Box[1] (Goral et al., 1984). Each user is provided with a `readme` file, explaining everything that should be done, a questionnaire and the `RayVis.exe`[2] and the `.rayvis` file, containing the Cornell Box scene and ray data for different effects. At the beginning of the study, each user has to fill out the first part of the questionnaire. Containing questions about the raytracing background of the user. After that, the scene is loaded and the user takes some time to get familiar with the software. Then the user is tasked with answering questions about the ray location and density of the scene, including the research questions. Finally, the user is asked to evaluate his experience with the software, in regards to usability and ability to solve the research questions with the visualizations.

4.2 Results

Five experts, which are proficient with raytracing both through their work at AMD and their scientific background, have participated in the user study, resulting in the following results. The research questions could be mostly answered, by the participants, for the Cornell Box data set (See Table 4.1). All users were able to

[1] Created artificially with a raytracing software.
[2] This version of *RayVis* differs slightly from the version used to generate the image in this thesis.

Question ↓	Successfully answered →	yes	no	n/a
Trace 0:	Where is the ray density of the hitting rays particularly high?	4	0	1
	Where was the camera in the trace?	5	0	0
Trace 4:	Where are the hitting rays located in the scene?	4	1	0
	Where is the ray density of the missing rays particularly low?	3	1	1

Table 4.1. Cornell Box trace question results. Trace 0 contains primary rays and trace 4 contains shadow rays.

Effect ↓ Correctly classified →	yes	no
Trace 0 - primary rays (1)	5	0
Trace 1 - primary rays (2)	5	0
Trace 2 - reflection rays	3	1
Trace 3 - ambient occlusion rays	2	2
Trace 4 - shadow rays	5	0

Table 4.2. Cornell Box effect classification results. The users were tasked with finding out which effect was calculated by the different ray data traces contained in the Cornell Box `.rayvis` file. Between reflection, ambient occlusion and global illumination ray calculations rays are distributed similarly, making it more difficult to differentiate them from each other.

identify the traces containing primary and shadow rays for the Cornell Box. Ambient occlusion and reflection rays were correctly classified by most users (See Table 4.2).

In the final part of the questionnaire, each visualization was named as the most useful, for answering the questions of the study, at least once. Regarding the question, of whether *RayVis* would already improve current workflows with ray data sets, two participants said it would improve it. The other participants indicated an improvement in their workflow, only when the software gets more developed and some major changes would be made to improve usability and clarity.

All participants struggled with the unintuitive user interface and were missing tooltips, explaining the function of each parameter. The performance of the *RayMesh* visualization is quite poor even on computers with the latest flagship CPU and GPU. This problem did not limit the users in answering the questions.

5 Future Work

The user study concludes that *RayVis* is a useful tool, which still has minor and major grounds for improvement. Both usability and visualization clarity need work, before *RayVis* could be adapted into the core workflow of analyzing ray data sets. There are also some areas, such as the *RayMesh* visualization, where the performance of *RayVis* is lacking. Additional visualizations could be added to show more and different information from the ray data sets and to widen the application areas of *RayVis*.

The performance of *RayVis* is not optimal in some cases. The rendering of all rays in the *RayMesh* visualization is, in the current implementation, not efficient enough to be considered interactive. This could be improved by switching to a rasterization pipeline for this visualization. The performance of the volume visualization, for high resolutions, can be improved by using a better spares volume data structure, than the two-level chunk data structure currently used in *RayVis*.

An acceleration structure heat map visualization was considered for this thesis but was left out due to not fitting into the final scope. The concept for this visualization is to calculate the amount of rays traversing each bounding volume. Then the bounding volumes could be visualized by drawing their bounds and coloring them accordingly to the number of rays. This could be used to identify parts of the acceleration structure, which are traversed the most. Another variant of this approach is to differentiate between rays traversing through a bounding volume and rays hitting something in the bounding volume. Rays that traverse a bounding volume, without hitting anything inside it, are heavy on the performance and therefore interesting to identify. A visualization for ray-start- and end-points was also considered and brought up in the expert user feedback. The approach is similar to the *RayMesh* visualization but only draws short arrows at the start of each ray. Additionally, at the hit- and end-points of the rays glyphs are drawn and colored depending on the ray state. It contains additional information, compared to the *RayMesh*, and would also be less cluttered.

REFERENCES

PAPPER & THESIS REFERENCES

Amanatides, J., & Woo, A. (1987). A fast voxel traversal algorithm for ray tracing. *Proceedings of EuroGraphics*, *87* (cit. on p. 11).

Bittner, K. (2020). The current state of the art in real-time cloud rendering with raymarching (cit. on p. 7).

Bui-Tuong, P. (1975). Illumination for computer generated pictures. *CACM* (cit. on p. 12).

Curless, B., & Levoy, M. (1996). A volumetric method for building complex models from range images. *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, 303–312 (cit. on p. 10).

Goral, C. M., Torrance, K. E., Greenberg, D. P., & Battaile, B. (1984). Modeling the interaction of light between diffuse surfaces. *ACM SIGGRAPH computer graphics*, *18*(3), 213–222 (cit. on p. 21).

Green, S. (2005). Volume rendering for games. *Game Developers Conference*, *2005* (cit. on p. 7).

Gribble, C. P., Fisher, J., Eby, D., Quigley, E., & Ludwig, G. (2012). Ray tracing visualization toolkit (cit. on pp. 2, 6).

Kaufman, A. E. (1996). Volume visualization. *ACM Computing Surveys (CSUR)*, *28*(1), 165–167 (cit. on p. 6).

Museth, K. (2013). Vdb: High-resolution sparse volumes with dynamic topology. *ACM transactions on graphics (TOG)*, *32*(3), 1–22 (cit. on p. 7).

Wronski, B. (2018). Volumetric fog and lighting. *GPU Pro*, *360*, 321 (cit. on p. 7).

Zhang, Q., Eagleson, R., & Peters, T. M. (2011). Volume visualization: A technical overview with a focus on medical applications. *Journal of digital imaging*, *24*, 640–664 (cit. on p. 7).

Zhu, Y., Williams, S., & Zwiggelaar, R. (2007). A hybrid asm approach for sparse volumetric data segmentation. *Pattern recognition and image analysis*, *17*, 252–258 (cit. on p. 7).

BOOK REFERENCES

Galbis, A., & Maestre, M. (2012). *Vector analysis versus vector calculus*. Springer Science & Business Media. (Cit. on p. 8).

Glassner, A. S. (1989). *An introduction to ray tracing*. Morgan Kaufmann. (Cit. on p. 5).

Marrs, A., Shirley, P., & Wald, I. (2021). *Ray tracing gems ii: Next generation real-time rendering with dxr, vulkan, and optix*. Springer Nature. (Cit. on pp. 5, 9).

ONLINE REFERENCES

AMD. (2023, March). *Radeon data file library* [Accessed on 10.07.2023]. Advanced Micro Devices, Inc. https://github.com/GPUOpen-Drivers/libamdrdf (cit. on p. 10).

Cornut, O., & contributers. (2023). *Dear imgui* [Accessed on 12.07.2023]. https://github.com/ocornut/imgui (cit. on p. 18).

Epic Games. (2017, October). *Unreal engine sun temple, open research content archive (orca)* [Accessed on 10.07.2023]. https://developer.nvidia.com/ue4-sun-temple (cit. on p. 10).

Epic Games. (2023). *Unreal engine 5 documentation - traces overview* [Accessed on 08.07.2023]. https://docs.unrealengine.com/5.0/en-US/traces-in-unreal-engine---overview/ (cit. on p. 5).

Microsoft. (2020). *Directx raytracing (dxr) functional spec* [Accessed on 10.07.2023]. https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html (cit. on p. 12).

NVIDIA. (2023). *Ces 2023 rtx dlss game updates* [Accessed on 05.04.2023]. https://www.nvidia.com/en-us/geforce/news/gfecnt/20231/ces-2023-rtx-dlss-game-updates/ (cit. on p. 1).

Unity Technologies. (2023). *Unity documentation - physics.raycast* [Accessed on 08.07.2023]. https://docs.unity3d.com/ScriptReference/Physics.Raycast.html (cit. on p. 5).

LUDOGRAPHY

Asobo Studio. (2022, October). A plague tale: Requiem. *Focus Entertainment*. (Cit. on p. 5).

Avalanche Software. (2023, February). Hogwarts legacy. *Warner Bros. Games*. (Cit. on p. 1).

CAPCOM Co., Ltd. (2023, March). Resident evil 4. *CAPCOM Co., Ltd.* (Cit. on pp. 1, 5).

CD Projekt RED. (2020, December). Cyberpunk 2077. *CD Projekt RED*. (Cit. on pp. 1, 5).

Appendix A. Code Listings

A.1 RayVis File Format Specification

Listing A.1. Ray Data Chunk Specification

```
static constexpr std::uint32_t RAY_TRACE_VERSION    = 1;
static constexpr const char*   RAY_TRACE_CHUNK_ID  = "RAYVIS_RAYTRACE";

struct RayTraceHeader final {
        std::uint32_t traceId; //This is usually identical to chunk id
};
struct Ray final {
        std::uint32_t rayId;

        Float3 origin;      /// Ray origin
        float  tMin;        /// Ray time minimum bounds
        Float3 direction;   /// Ray direction
        float  tMax;        /// Ray time maximum bounds

        float tHit = -1;

        struct {
                std::uint32_t instanceIndex;
                std::uint32_t primitiveIndex;
                std::uint32_t geometryIndex;
        } hitInfo;
};
```

Listing A.2. Scene Data Chunk Specification

```
constexpr std::uint32_t SCENE_CHUNK_VERSION = 1;
constexpr const char*   SCENE_CHUNK_ID      = "RAYVIS_SCENE";

// The scene chunk contains as many MeshPrefix's
//              (and their follow-up data as specified by meshCount)
// The mesh data is followed by as many RootNodePrefix's
//              (and their follow-up data as specified by rootNodeCount)
struct SceneChunkHeader final {
        size_t meshCount;
        size_t rootNodeCount;
};

// Followed by as many PrimitivePrefix's and
// their follow-up data as specified by primitvieCount
struct MeshPrefix {
        size_t primitvieCount;
};

// Followed by a data chunk of size vertexByteSize*vertexCount
//      containing primitive vertices in float DXGI_FORMAT_R32G32B32_FLOAT
// Vertex data is followed by a data chunk of size indexByteSize*indexCount
//      containing primitive indices in
//      either DXGI_FORMAT_R32_UINT or DXGI_FORMAT_R16_UINT
//      depending on indexByteSize
struct PrimitivePrefix {
```

```
        size_t vertexByteSize;
        size_t vertexCount;
        size_t indexByteSize;
        size_t indexCount;
};

// Followed by as many NodePrefix's and their children as specified by nodeCount
struct RootNodePrefix {
        // Describes a scene graph

        size_t nodeCount;
        size_t rootNodeId; // has to be zero
};

// Followed by as many child ids as uint32_t as specified by childCount
struct NodePrefix {
        size_t childCount;

        size_t meshId    = std::numeric_limits<size_t>::max();  // max == no mesh
        Float3 meshColor = Float3(0.f);

        int32_t   id;
        uint8_t   instanceMask;
        linalg::Matrix4x4 matrix;
};
```

A.2 Trace Volume Function

Listing A.3. Trace Volume

```
// Volume data struct
// contains metadata for the volume
// and is present as a constant buffer
struct VolumeData
{
        float3 min;
        float chunkCount;
        float3 max;
        float cellSize;
        float chunkSize;
};

float2 TraceVolume(
        float3 origin,
        float3 direction,
        float maxT,
        float3 minBounds,
        float3 maxBounds,
        Texture3D<float> volume)
{
        float2 minMaxT = IntersectAABB(origin, direction, minBounds, maxBounds);
        minMaxT = float2(max(0, min(minMaxT.x, maxT)), min(minMaxT.y, maxT));

        if ((minMaxT.y < minMaxT.x) || (maxT <= minMaxT.x) || (minMaxT.y < 0))
        {
                // Early return if the volume is not hit
                return float2(0.0, 0.0);
        }


        int maxSteps = (volumeData.chunkSize * SQR3) * samplesPerCell;

        float stepSize = volumeData.cellSize / samplesPerCell;
        float stepT = stepSize / length(direction);

        float startT = ceil(minMaxT.x / stepT) * stepT;
        int steps = min(ceil((minMaxT.y - startT) / stepT) + samplesPerCell, maxSteps);
```

```
        float weight = 1.f / samplesPerCell;

        float3 stepVector = direction * stepT;
        float3 samplePoint = (origin + direction * startT) - minBounds;
        float3 extends = maxBounds - minBounds;

        float volumeValue = 0;
        for (int i = 0; i < steps; i++)
        {
                volumeValue += volume.SampleLevel(
                        vSampler,
                        NonUniformResourceIndex((samplePoint / extends)),
                        0) * weight;
                samplePoint = samplePoint + stepVector;
        }
        return float2(volumeValue, 1);
}
```