Implementing a hardware-accelerated path tracing renderer in 'Quake'

Abschlussarbeit

zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

an der

'Hochschule für Technik und Wirtschaft (HTW) Berlin'
Fachbereich 4: Informatik, Kommunikation und Wirtschaft
Studiengang *Internationale Medieninformatik*

1. Gutachter_in: Prof. Dr-Ing. David Strippgen
2. Gutachter_in: Prof. Dr. Klaus Jung

Eingereicht von Ruslan Novikov [566919]

04.03.2022

Acknowledgment

Abstract (EN)

The increased computing power of modern graphics cards allows computationally expensive ray tracing operations to be performed in real time. However, the number of games that fully support ray tracing is very small. The goal of this thesis is to analyze the data structures and code base in 'Quake' to support hardware-accelerated ray tracing operations in order to create a path tracing render. For this purpose, previous rasterization based rendering methods are repurposed or replaced. An analysis of the state-of-the-art APIs and games is done to determine the commonly used methods and techniques in path tracing applications.

Abstract (DE)

Dank der erhöhten Rechenleistung moderner Grafikkarten können rechenintensive Raytracing-Operationen in Echtzeit durchgeführt werden. Allerdings ist die Zahl der Spiele, die Raytracing vollständig unterstützen, sehr gering. Das Ziel dieser Arbeit ist es, die Datenstrukturen und die Codebasis in 'Quake' zu analysieren, um hardwarebeschleunigte Raytracing-Operationen zu unterstützen um somit ein Path-Tracing Renderer zu erstellen. Zu diesem Zweck werden bisherige, auf Rasterisierung basierende Rendering-Methoden umgeschrieben oder durch Ray Tracing Lösungen ersetzt. Es wird eine Analyse der aktuellen APIs und Spiele durchgeführt, um die am häufigsten verwendeten Methoden und Techniken in Path-Tracing-Anwendungen zu ermitteln, die eine Umstellung des Renderes ermöglichen.

Contents

List of Figures

1. Introduction

I. Motivation

Due to the ever-increasing performance of GPUs each year, ray tracing and similar algorithms, such as path tracing, have seen increased use in recent years. Especially in 3D design and video production, light transport algorithms are used to simulate realistic lighting. The required computational effort reduced the feasibility in video games and other real-time applications.

With the release of Nvidia's graphic cards series 'GeForce RTX' in 2018, the chip designer brought major changes in GPU chip design and processor architecture. The most notable change was the introduction of additional tensor and 'RT' units that provided faster computational capabilities for machine learning tasks and ray tracing operations. The boost in computing speed for ray tracing operations had the potential to make ray tracing in games more viable.

Since release of the 'GeForce RTX' series and subsequent introduction of other GPUs with ray tracing by AMD, an increasing number of game developers were taking advantage of the new ray tracing capabilities and over 150 games have been released that see use of these capabilities for realistic graphics rendering [1].

The use case is often limited to graphical effects like shadows and reflections. The majority of the rendering is still accomplished with traditional rasterization based techniques, to meet the target performance preference of modern video games.

However, hardware-accelerated ray tracing offers the opportunity for new games and applications to take full advantage of the new ray tracing features. Already, some research and commercial products have been released and published that render games using only a ray traced graphics pipeline [2]. Although the amount of such titles is small, there is potential for a new game market and new research to be conducted in this area.

II. Thesis objectives

The focus of this thesis is the analysis of the underlying data structure and render pipeline of an open-source game to enable the implementation of a path tracing rendering pipeline. The analysis will identify key problems that occur when transitioning from a rasterization based renderer to a path tracing renderer.

Select sample projects and the state-of-the-art technology are analyzed for patterns and solutions for frequently occurring problems. An attempt is then made to implement these solutions by adapting them to the system architecture and data structures present in the unmodified game project.

The visual result of the modification is compared to a existing sample project to measure the visual quality of the current modification

III. Chapter overview

Introduction: In the introduction chapter, an overview of the motivation and research objectives is given.

Theoretical background: For the further understanding of the subject matter in this thesis, important concepts and terms are explained.

State-of-the-art: To better evaluate the feasibility of the modification effort, an analysis of the current state-of-the-art is performed that includes available graphic APIs and game projects.

Concept: The unmodified game project and its current rendering methods are analyzed. Based on the results of the analysis, solutions for new render pipelines and data structures are designed and proposed. A list of functional and non-functional requirements is identified and used to evaluate the quality of the modification in further chapters.

Implementation: This chapter deals with the implementation and realization of the proposed solutions. It details concrete implementation details unique to the project, as well as general implementation details that are part of the selected graphics API.

Evaluation: The modified application is evaluated by comparing the visual results with a selected sample project to evaluate the visual quality of the modification.

Conclusion: At last, the objectives defined in the first chapters are revisited and reflected. Furthermore, an outlook is given regarding possible improvement and expansion in the future.

2. Theoretical background

This chapter gives an overview of light transport algorithms and data structures relevant to this thesis. These act as a introduction or refresher on some techniques that are used often through the project.

I. Ray Tracing

Ray tracing is a light transport algorithm, where rays or vectors are used to simulate the physical behavior of light. These rays traverse a scene of geometric elements, to determine if an object intersects with a ray. In case a ray intersects an element, the attributes of the element are retrieved and a new path is calculated. The ray continues its path until it loses its potential energy or does not hit any geometry in the scene. The exact requirements for termination can be defined by the program to adjust the quality of the simulation or to improve performance of the calculation.

The ray simulation can start from the light sources, where the rays traverse the scene until the ray hits the eye or camera. This method is referred to as 'forward ray tracing. This approach, although realistic, is computationally expensive, as it does not gaurantee that the emitted light will hit the eye or camera. This results in a majority of rays that never contribute to the output image.

The dispatched ray is traced until a certain condition, like a predefined iteration depth, terminates further ray bounces. This approach is referred to as "backwards ray tracing" and ensures that each pixel has a guaranteed output value, which reduces the aforementioned calculation overhead significantly.

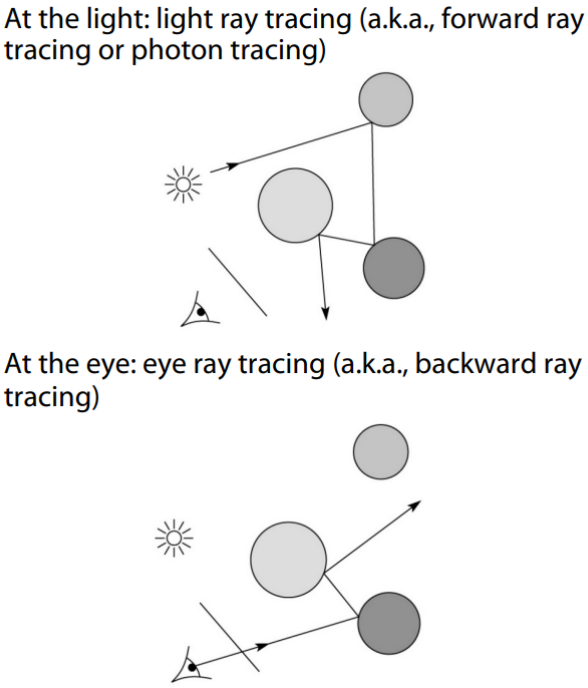


Figure 2.1.: *Ray origin from light source and from camera; Image source: [3]*

The first computer generated ray traced image was created by Arthur Appel in 1968 that was mentioned in his paper 'Some techniques for shading machine renderings of solids' [4]. Appel's approach uses primary rays to calculate visibility of the object and secondary rays in direction to the light source to determine whether the point was in shadow.

In 1979, Turner Whitted published his paper 'An improved illumination model for shaded display' where he proposed an improvement on the ray-tracing algorithm [5]. This algorithm could shoot primary visibility rays, shadow rays, reflection rays and refracted rays and used recursive function calls to take advantage of the recursive nature of ray operations. To simulate different materials, Whitted used the 'Phong' shading method to imitate diffuse and specular shading for the objects in the scene.

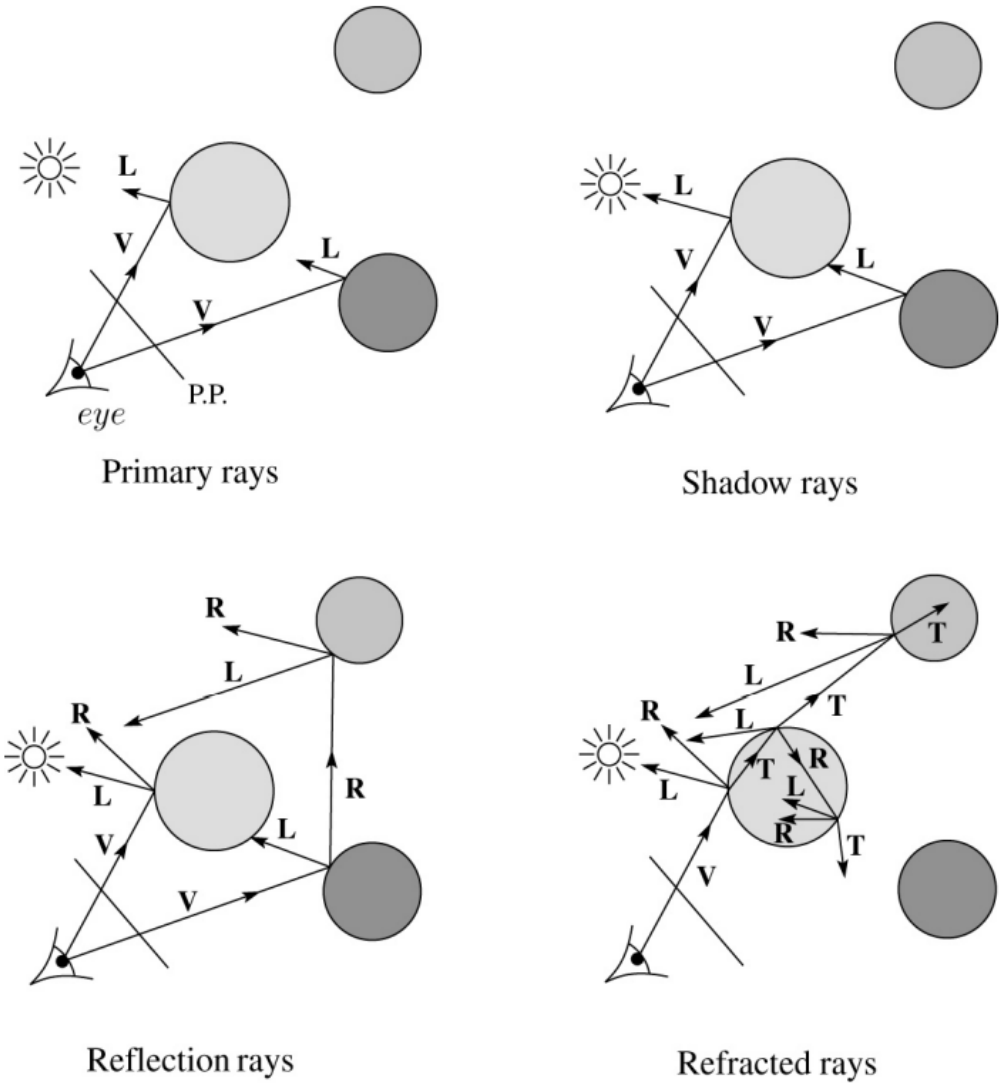


Figure 2.2.: *Illustration showing light (L) and reflection (R) rays for different scenarios; Image source: [6]*

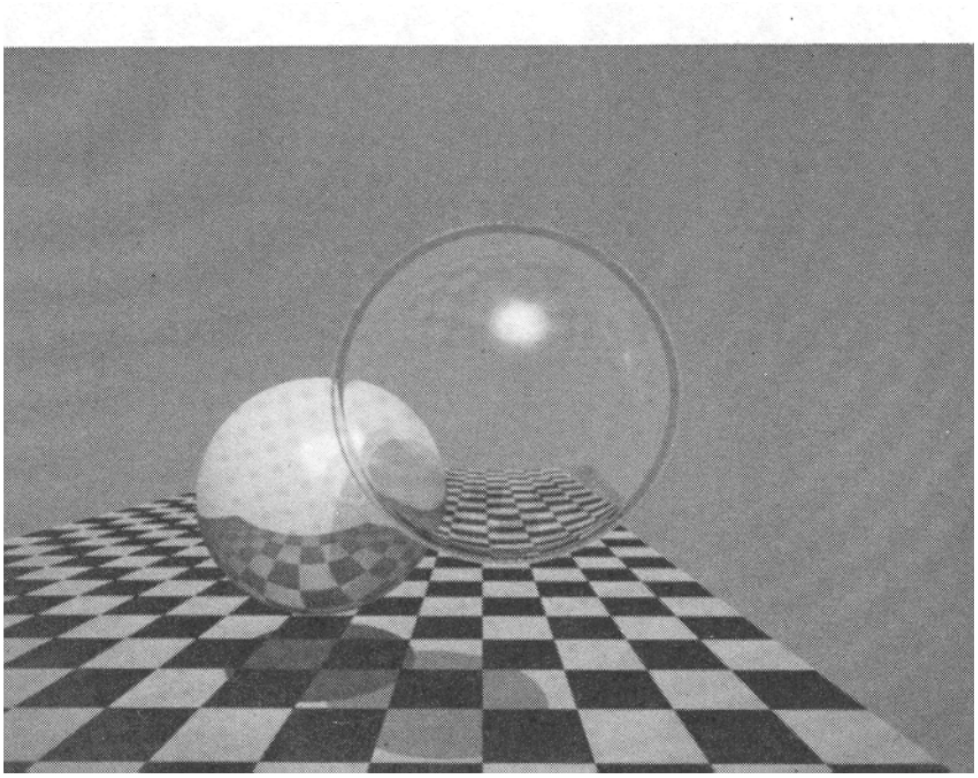


Figure 2.3.: *Example: Ray traced image by Turner Whitted. Showcases reflection, refraction and shadows; Image source: [7]*

II. Path Tracing

Path tracing is a light transport algorithms that functions similarly to ray tracing but expands the concept further. Similar to ray tracing, path tracing simulates light by casting rays into a virtual scene. Depending on whether the starting point of the light rays originates from the light source or from the camera, a distinction is made between 'forward-' and 'backward path tracing', the latter of which yields similar benefits as in ray tracing. A combination of both approaches is referred to as 'bidirectional path tracing'. In stark contrast to regular ray tracing, where rays have a deterministic path, path tracing uses multiple random paths to render an image.

Different materials show different surface properties. Diffuse materials, for example, show a rough surface, which forces light to scatter in different directions, while reflective materials, such as mirrors, show smoother surfaces, where light rays are reflected more evenly. This behavior is modeled with a bidirectional scattering distribution function, which determines the distribution of light around the hit point. This function is combined with a randomly uniform sampling of the hemisphere around the surface normal.

The combined result is a model, where specific ray directions are more likely to be taken than others.

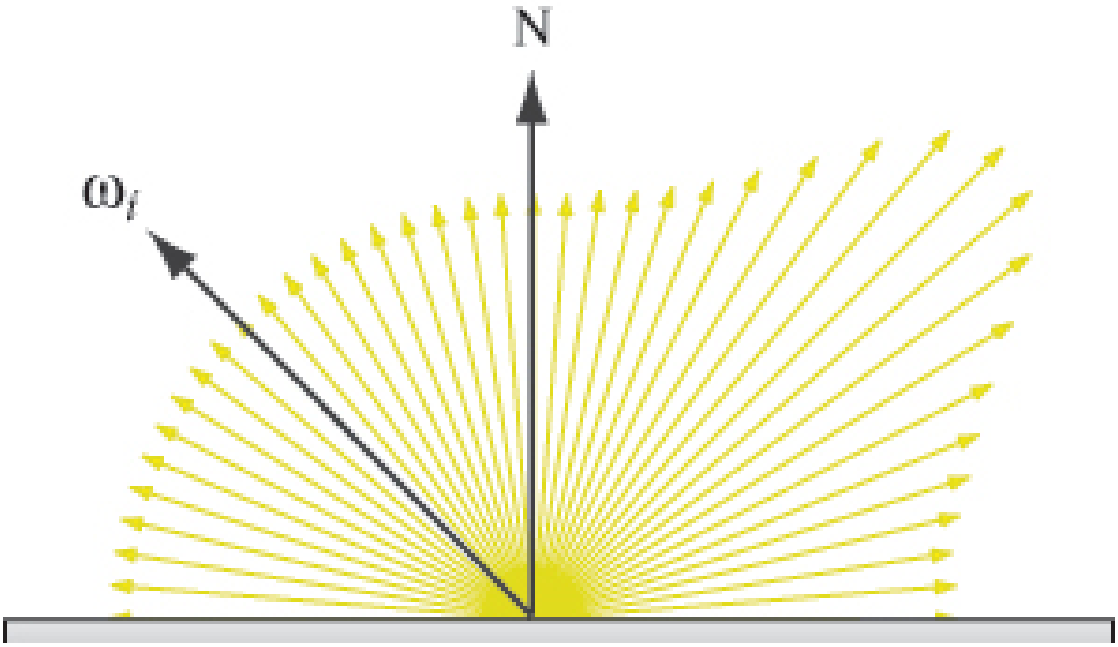


Figure 2.4.: *The Bidirectional Reflectance Distribution Function BRDF shows the distribution of light around the surface normal as seen in 2D space; Image source: [8]*

The repeated dispatch of rays estimates the whole range of paths a ray might take when hitting an object's surface.

This corresponds to the integral part of the rendering equation.

$$L\left(x,\vec{w}\right) = L_e\left(x,\vec{w}\right) + \int_\Omega f_r\left(x,\vec{w}^{'},\vec{w}\right) L\left(x,\vec{w}^{'}\right)\left(\vec{w}^{'}\cdot\vec{n}\right) d\vec{w}^{'} \tag{2.1}$$

The equation 2.1 specifies the illuminance of any light ray emanating from a given point.

This method is referred to as 'Monte-Carlo path tracing' as it approximates a numerical value by random sampling.

In addition, rays can be dispatched toward light sources to sample direct light influence. This is referred to as 'light sampling' and can instantly improve the visual result of the path tracing solution.

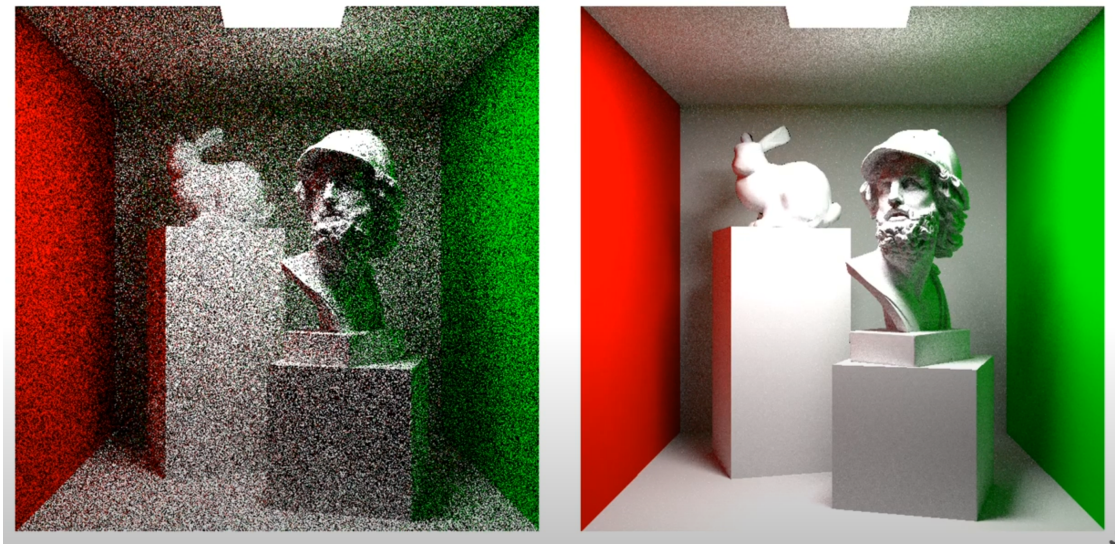


Figure 2.5.: *Example: Left image is a result of unbiased path tracing. Right image uses light sampling and next event estimation to reduce variance at same sample rate; Image source: [9]*

The process of determining the contribution weight of both rays is referred to as 'multiple importance sampling', which eliminates the oversampling of light sources.

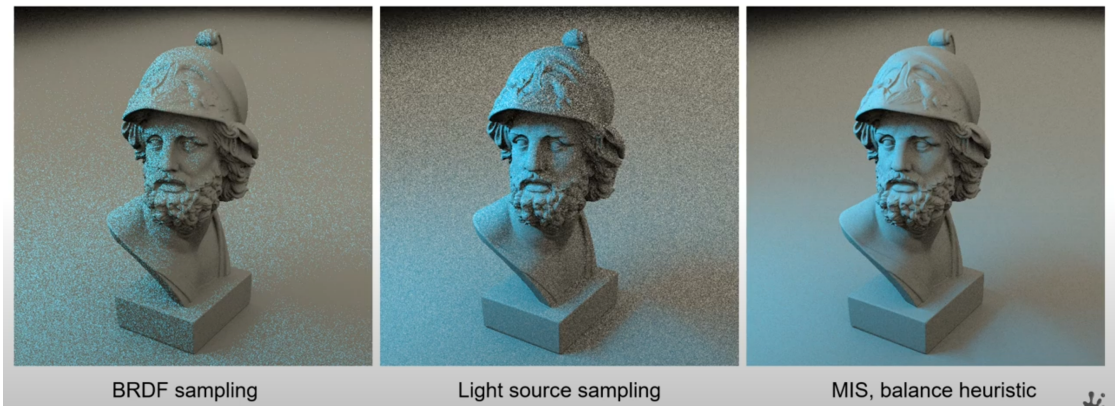


Figure 2.6.: *Example: The figures demonstrate light influence using different sampling methods. The right figure uses balance heuristic to weight the results of BSDF and light source sampling; Image source: [9]*

The multiple random sampling can simulate many effects such as soft shadows, depth of field and indirect lighting naturally that otherwise need to be additionally implemented in other rendering methods such as ray tracing or rasterization based rendering.

The brute force solution is also the biggest downside to path tracing, as rays have to be evaluated multiple times for the same pixel in order to estimate a color value. This makes this approach computationally expensive.

III. Tree structures

Tree structures are non-linear abstract data structures, in which each element references one or multiple child elements. When displayed visually, the structure resembles a tree with roots, branches, and leaves.

A useful advantage of trees over other linear data structures, such as lists, is the efficient search and retrieval of elements that can be performed in logarithmic time, compared to linear time for linear data structures. Therefore, tree structures play an essential role in computer science and are used to represent complex hierarchical structures that can be traversed efficiently.

Such tree structures see heavy use in ray tracing and path tracing, for collision detection or for computer assisted design, specifically for boolean operations. In games like 'Doom' and 'Quake', BSP trees are used for faster representation of level geometry [10].

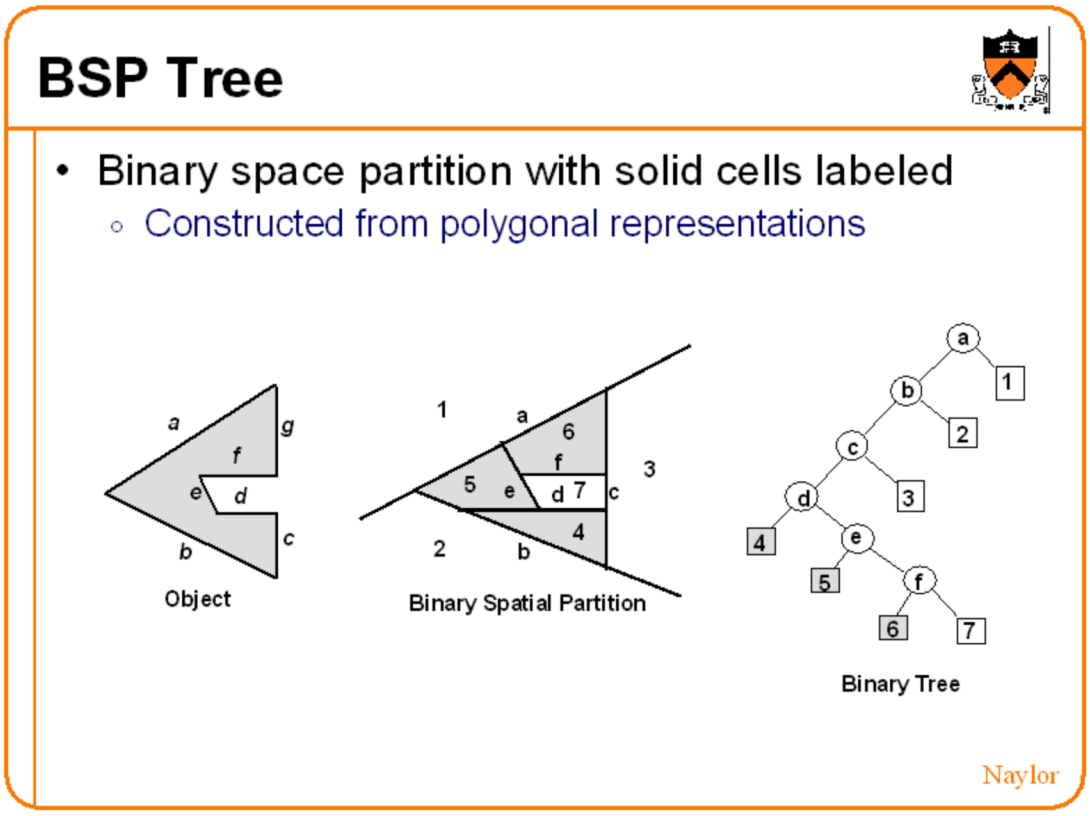


Figure 2.7.: *Example: Binary space partitioning performed on solid object and its representation as a binary tree; Image source: [11]*

IV. Acceleration structures

In ray tracing and path tracing, tree structures are used to reduce intersection calculations, which are essential to determine if a ray intersects an object. Without a suitable tree structure, the intersection calculation has to be performed for each individual object in a scene. The evaluation can become inefficient when thousands of objects are involved.

By partitioning the scene space, it is possible to rule out objects that will never intersect with the incoming ray. This evaluation can save a significant part of computational work, the benefit of which only increases with higher object counts.

The illustration below shows how a partitioning of the scene space can discard geometries to save computational work.

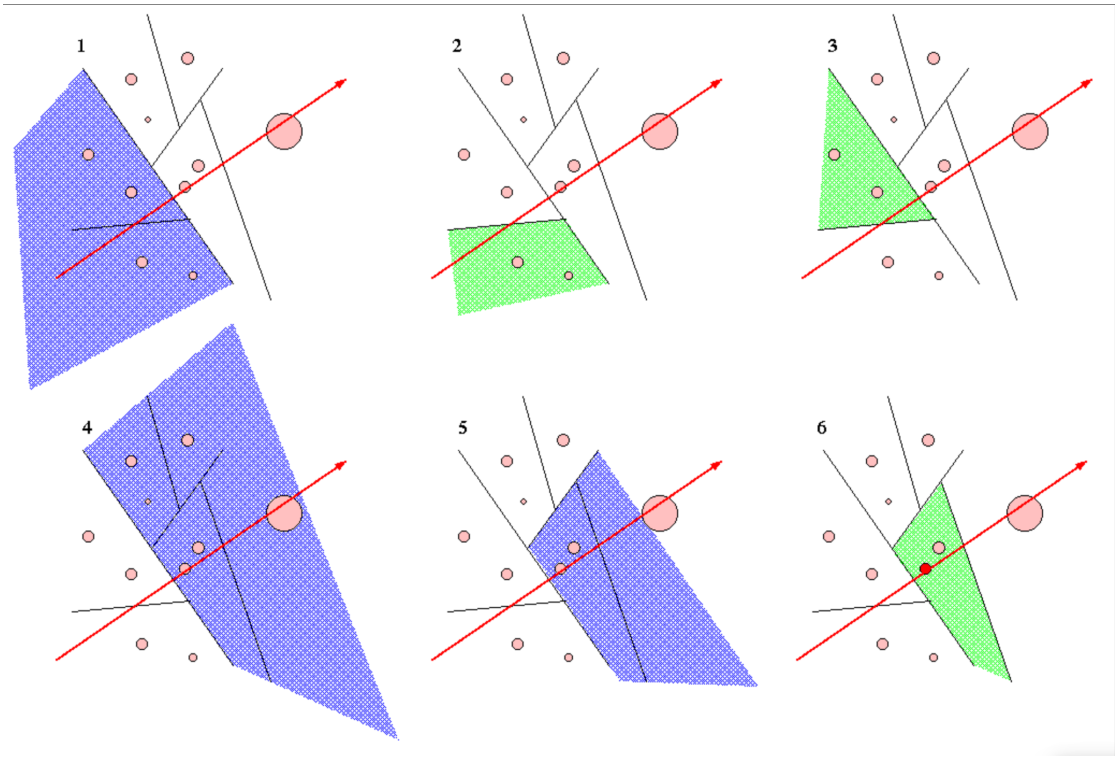


Figure 2.8.: *Example: Representation of a ray traversing a virtual scene. Figures 4 to 5 show how the upper two partitions are not considered for the calculation. Out of ten objects, only seven are considered for the intersection calculation; Image source: [12]*

The space partitioning is an important performance measure that makes real-time ray casting feasible. Modern ray tracing capable APIs benefit from tree structures and make them mandatory for ray tracing operations.

3. State of the art

Before a modification can take place, it is imperative to check by which means a modification can be realized. For this purpose, a detailed analysis of the state of technology is performed. This analysis includes the state-of-the-art application programming interfaces and software development kits. In addition, an overview of available game projects utilizing ray tracing for its rendering and the current state of hardware support is given.

The state-of-the-art analysis and overview in this chapter form an important part of the thesis. The results of the analysis are used to make an assessment of the feasibility and potential scope of the modification. The assessment includes estimated time and required resources necessary to develop a prototypical implementation.

I. Graphic APIs

The graphics API is responsible for all functions concerning the graphical presentation of a game. Depending on the API, software engineers receive a certain level of access to the GPU, which enables great control over the implementation which in turn increases the complexity of the programming interface.

As of March 2022, two graphic APIs are available that integrate hardware-accelerated ray tracing capabilities. The Microsoft DirectX Ray Tracing (DXR) API and the Vulkan API by the Khronos Group.

Both APIs offer low-level access to GPU functions and support for hardware-accelerated ray tracing. In addition, both APIs are free to use with no additional costs.

In terms of ray tracing capabilities, both APIs offer similar functionalities to enable a interchangeable code base. The code is therefore easy to port between the APIs as both utilize similar data structure and pipelines.

Table 3.1.: *Vulkan Ray Tracing and DirectX 12 DXR features [13]*

	Vulkan Ray Tracing	DirectX12 DXR
Ray Tracing Pipelines	At least one must be available	Yes
Ray queries		DXR Tier 1.1
Language for Ray Tracing Shaders	GLSL or HLSL	HLSL
Pipeline Libraries	Yes	DXR Tier 1.1

In terms of accessibility, both APIs offer different requirements and accessibility features. The table below gives an overview of some accessibility features present in both APIs:

Table 3.2.: *Vulkan Ray Tracing and DirectX 12 DXR accessibility functions*

Function	Vulkan Ray Tracing	DirectX12 DXR
Hardware-accelerated ray tracing support	Yes	Yes
Price	Free	Free
Open source	No	No
Cross-Plattform support	Windows and Xbox platforms	Windows, MacOS, Linux, Nintendo Switch, Stadia
Native programming language	C++	C
Programming language binding support	Not available	C#, C++, Java, Rust, JavaScript, D, Haskell, Racket
SDK operating system support	Windows 10 or higher	Windows, Linux and MacOS

For the sole use of hardware-accelerated ray tracing, both APIs offer a similar feature set. But in terms of accessibility, the Vulkan API offers more for different programmers. The strict requirements on the operating system for DirectX12 prevents non-Windows users from using DirectX12, making the Vulkan API the only option in some circumstances.

II. Selection of game project

Due to the rather limited selection of graphics APIs that support hardware-accelerated ray tracing, not all game projects are equally suitable for a modification.

For this thesis, a selection of games have been considered for modification. In the following subsections, the considered games are evaluated under several aspects. The aspects include the age of the game, game complexity, open-source aspect and used graphics API. At the end of the evaluation, one of the proposed games is selected and the reasons for the selection are explained.

i. Quake

Due to the age of the game 'Quake', developed by 'idSoftware' in 1996, it was the first consideration. Like its successor 'Quake II', 'Quake' is completely open source and licensed under the 'GNU General Public License', which meant that modification and distribution of this game was permitted [14].

Furthermore, this game supports the Vulkan graphics API through the "vkQuake" source port [15], so that the modification can be implemented directly into the game client without additional porting effort. Since 'Quake II' and the 'Quake II RTX' modification are based on similar engines, namely the 'Quake engine' and 'Quake II engine', some aspects of the path tracing modification could potentially be of use for the modification effort.

ii. Half-Life

The game 'Half-Life', which was created and published by 'Valve' in 1998, was also considered for its relative simplicity and its age. Unlike 'Quake', the code of 'Half-Life', as well as its engine, 'GoldSrc', are not open-source and thus the source codes are not publically available for a in-depth modification effort.

However, a custom game engine called 'Xash3D FSGW' [16], offers compatibility with the 'GoldSrc' engine, meaning that any game developed with 'GoldSrc', can be played using 'Xash3D FSGW'. This also means that, in theory, an alternative graphics render pipeline can be written.

The 'OpenGL' graphics pipeline that is currently present in 'Xash3D FSGW' needs to be replaced with either the Vulkan or DXR API, in order to support hardware-accelerated ray tracing. The replacement itself would require an enormous effort.A modification attempt would therefore be not realistic given the limited time frame available.

iii. The Elder Scrolls III : Morrowind

As with 'Half-Life', an open-source engine is available to play the open-world game 'The Elder Scrolls III : Morrowind' developed by 'Bethesda Game Studios' in 2002. However, this game engine faces similar problems as the 'Xash3D FSGW' engine which requires a rewrite of the 'OpenSceneGraph' graphics API present in the 'OpenMW' engine [17]. In addition to this, the age and technical state of 'The Elder Scrolls III : Morrowind' is

more advanced than 'Half-Life' which would result in a more demanding modification effort.

iv. Conclusion

The graphics engine present in the game is a cruical factor that determines the feasibility of a modification. Although a conversion of the graphics API is certainly possible, it requires substantial time and resources. For the limited time given in the thesis, the best and most realistic choice was 'Quake' since it allowed direct modification, made possible by the 'vkQuake' Vulkan port.

III. Existing game projects

The analysis of existing projects using hardware-accelerated ray tracing yields interesting results for the assessment of the feasibility of the modification. The sample projects also provide an invaluable resource for learning, besides official programming guides and tutorials, that can accelerate the development of the modification.

The data sources for this research were primarily official release and publishing lists of hardware-accelerated ray tracing games and applications, as well as 'GitHub' and other source-control management repositories.

Therefore, games that utilize hardware-accelerated ray tracing for rendering additional graphical effects were excluded. An example of such games is 'Cyberpunk 2077', developed by 'CD Project RED' released in 2020 [18]. In addition, games that use ray tracing and similar light transport algorithms exclusively for graphics rendering, but do not utilize hardware-acceleration from one of the available APIs, are also not considered. An example of such games is 'Teardown' developed by 'Tuxedo Labs' released in 2020 [19].

The results of this research were summarized the following table view.

Table 3.3.

Game	Released	Created by	Open Source	Technique used	API
Quake II RTX [20]	June 2019	Christoph Schied / Lightspeed Studios	Yes	Path tracing	Vulkan API
Minecraft with RTX	June 2020	Mojang	No	Path tracing	DirectX12 API (DXR)
Metro Exodus Enhanced Edition	May 2021	4A Games	No	Ray/path tracing	DirectX12 API (DXR)
Serious Sam TFE: Ray Traced [21]	August 2021	Sultim Tsyrendashiev	Yes	Path tracing	Vulkan API
xash3d-fwgs [16]	No	Ivan Avdeev	Yes	Path tracing	Vulkan API
Half Life: Ray Traced	No	Sultim Tsyrendashiev	TBA	Path tracing	Vulkan API
Quake RTX [22]	Discontinued	Justin Marshall	Yes	Ray tracing	DirectX12 API (DXR)
Quake III Arena with DXR [23]	On Hold	Michael Young	Yes	Ray/path tracing	DirectX12 API (DXR)

(fußnoten fehlen)

The project list shows a very limited amount of currently available projects. In addition, almost half of the unofficial modifications are discontinued or are currently on hold.

One key aspect of this research is that the information gathered is based on the version control repositories, where data such as commits, contributing users and the description page are the only sources. Only officially published games have official websites where information can be extracted.

Another aspect is the lack of documentation for most projects. The majority provide technical information about setup and build process, but documentation and implementation details are non-existent in most cases. It is therefore up to the user to extract relevant information from the source code.

i. Overview of feature complete games

To select good resources that can help with development of a path tracing render, a priority list of important requirements was created. The list includes:

- Open-Source repository
- A path tracing rendering solution
- Global Illumination
- Support for multiple materials representation
- Denoising solution
- Inclusion of documentation and implementation resources.

Of all listed games, only 'Quake II RTX' features all six features. Formerly known as 'Q2VKPT' [24] the project started off as a spare-time project by Christoph Schied, that turned into a research project for evaluation of denoising algorithms.

The result was a path-tracing renderer with a custom adaptive-temporal filter solution [25] that could handle low sample input signals and eliminate ghosting effects. The technically advanced state of the project, the open-source aspect and the availability of additional resource material makes this project the current state-of-the-art for full path traced games and of the few projects that have been officially released and published.

Figure 3.1.: *Screenshot from 'Quake II RTX'.*

In terms of quality and feature set, 'Serious Sam TFE: Ray Traced' is a close candidate. The presented game project and its corresponding engine 'Serious Engine: RT' were developed independently by Sultim Tsyrendashiev, starting in January 2020 with an official release in August 2021. It shares many technical similarities with 'Quake II RTX' and offers features like dynamic path-traced global illumination and a similar A-SVGF temporal denoising solution for low sample signals.

Compared to 'Quake II RTX' however, it lacks official documentation and does not provide new solutions, unlike the temporal denoising filter by 'Quake II RTX'.

IV. Hardware requirements

To run software that utilize the hardware-accelerated ray tracing capabilities of the DXR and Vulkan APIs, special hardware is required.

The graphic cards manufacturers 'Nvidia' and 'AMD' offer special graphic cards that include additional hardware components to accelerate ray tracing operations and give additional driver support for hardware-accelerated ray tracing use for software using the Vulkan or DXR graphics APIs.

i. Vendor product overview

Nvidia offers ray tracing support for every graphics card in the 'GeForce RTX' series, which includes RT cores in the chip design. Some cards of the predecessor series 'GeForce GTX' also support ray tracing, but only through FP32 shader cores for 'Pascal' and FP32 and INT32 shader cores for 'Turing' architecture cards [26].

AMD also offers hardware-accelerated ray tracing for select cards incorporating the 'AMD RDNA 2' architecture. The 'AMD RDNA 2' architecture implements additional specialized hardware for each compute unit that handles ray intersections, similar to Nvidia's 'RT cores'. As of March 2022, it includes only the 'AMD RX 6000' series of graphic cards [27]. Unlike some 'Nvidia GTX' cards that received official driver support for software level ray tracing on older GPU hardware, official AMD drivers do not support this feature. The Mesa 3D Graphics Library, however, releases unofficial AMD RADV Vulkan drivers that supports software level emulation for BVH traversal on older AMD GPUs with RDNA 1, Vega and Polaris architectures [28].

ii. Task of hardware units

Both the 'RT' core from 'Nvidia' [29], as well as the 'Ray Accelerator' from 'AMD' are based on the same principle to accelerate ray tracing operations. The additional hardware units perform the traversion of bounding volume hierarchies that form the basis of acceleration tree structures, used to encapsulate the virtual scene that is subject to ray casting operations. The hardware units calculate the intersection of the ray with the bounding volume and the intersection of the ray with the mesh triangle of the model.

Set in hardware, this instruction set is streamlined, which leads to a quick evaluation of rays in a bounding volume hierarchy.

iii. Speed comparison

As an example of the benefits of hardware-acceleration in ray tracing, Nvidia demonstrates the improvement by comparing their 'RTX 2080 Ti' GPU with the 'GTX 1080 Ti' GPU.

The reference RTX 2080 Ti comes equipped with 68 RT cores that can effectively perform around 10 Gigaray[1] calculations per second. A reference GTX 1080 Ti without

[1] 1 Gigaray = one billion rays

RT cores can effectively perform around 1.1 Gigaray calculations per second, which results in a theoretically faster ray tracing performance of around nine times in favor of the RTX 2080 Ti [29].

According to measurements from AMD engineering labs, based on an 'AMD RDNA 2' graphics cards using the 'Procedural Geometry' sample application from the DXR SDK, the graphics card sees a speedup of up to 13.8x (471 FPS) using hardware-accelerated ray tracing compared to 34 FPS when using the DXR Software fallback layer at the same clock speed [27].

V. Summary

The overview of state-of-the-art technologies, as well as currently available frameworks and projects gives interesting insights about the current situation of hardware-accelerated ray tracing support in games and applications.

The selection of APIs is modest and the number of existing projects with full ray tracing support is limited as well. All presented projects vary heavily in terms of scope and quality, which is due to the fact that the majority of the projects are independent modification efforts. The number of feature-complete games additionally reduces the already limited list of games to two out of nine found games.

The refactoring of the code base has to be performed using a low-level API meaning that many aspects of the code have to be managed manually, which requires additional care when modifying the original project.

This reveals an additional layer of complexity and requires programmers to have an intermediate understanding of programming and extended knowledge about computer graphics.

Overall, a game project with full path tracing support is feasible, but the limited choices for APIs and sample projects give engineers a limited variety of resources to work with. A modification effort in the scope of 'Quake II RTX', which can be considered the state-of-the-art path tracing project, may take many months to years for single developers, depending on the skill and knowledge of the programmer.

4. Concepts

I. Data structures and graphic pipelines

Games use different data structures and graphic pipelines to represent visual elements. Depending on the game, the visual elements can comprise models, textures, particle effects and other elements.

A problem that arises when switching from a rasterization to a ray traced or path traced solution is that many of the previously used data structures and graphics pipelines are not suitable for such a pipeline and require a restructuring of several elements.

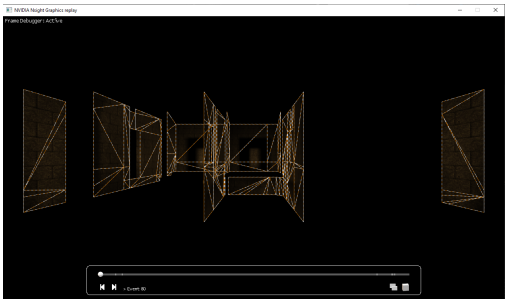
To understand why a restructuring is necessary, it is essential to understand the previous render method. For this purpose, a very simplified representation of the rasterization pipeline in Quake is presented.

i. In Rasterization

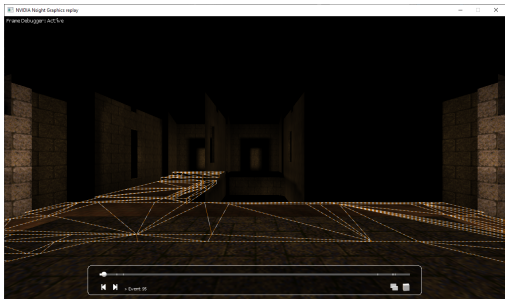
Using a rasterization pipeline, the underlying game code determines a list of all elements that need to be drawn for the current frame. The current element in the list is bound to its corresponding graphic pipeline, and various parameters are passed to the shader to ensure the correct visualization of the element.

The shader uses the information and executes the shader code to draw the element to the list. All individual elements are drawn in sequence until a complete image is created. At the end of the draw list, a new frame begins and the cycle begins again.

This process can be seen in the unmodified 'vkQuake' project, where parts of the game environment are added in sequence. The following steps were demonstrated using the 'Nvidia Nsight' graphics replay.



(a) *Draw call for set of polygons addressed as Event 80*



(b) *Consecutive draw call addressed as Event 95*

Figure 4.1.: *Both figures show the consecutive draw calls from left to right.*

ii. In Ray tracing

The rendering procedure for ray tracing is fundamentally different. Instead of drawing individual elements onto an image, ray tracing determines a color value for each pixel individually, by dispatching rays into a virtual scene.

Unlike in rasterization, where the element of interest is known in advance, in ray tracing such a guarantee does not exist. In theory, a dispatched ray can hit any position in the scene, so binding a graphic pipeline and an associated shader in advance is not possible.

It is therefore necessary to ensure that all elements required for the correct representation of the game are accessible at any given time. This not only includes the model geometry of the game, but also all associated textures, material properties, light sources and other elements.

A common method is to load elements in a common buffer, which can be accessed by special ray tracing shaders. The buffers can be used for different element types. It is therefore common to load static and dynamic models into individual buffers. Textures and other elements are also commonly separated.

Static elements can usually be loaded once, as they do not change in-between frames. This is very performance efficient, but can take a lot of allocated memory space when the amount of elements is large. Loading only elements necessary for the current frame will take less memory space, but the constant allocation of memory and required write operations to the buffer reduces the performance for each such operation. Depending on the game, a compromise between memory and performance has to be made.

II. Surface Visibility Determination

The computation speed of computers at the time of Quake's release in 1996 meant that the rendering had to be performance-optimized. Therefore, an appropriate number of techniques have been used to run games at an adequate performance, some of which

still see use in modern games.

One key aspect is the surface visibility determination, where, as the name suggests, an algorithm must determine which surface is visible in the current view. This can be used as a performance measure to reduce the amount of drawn elements.

i. Use in Quake

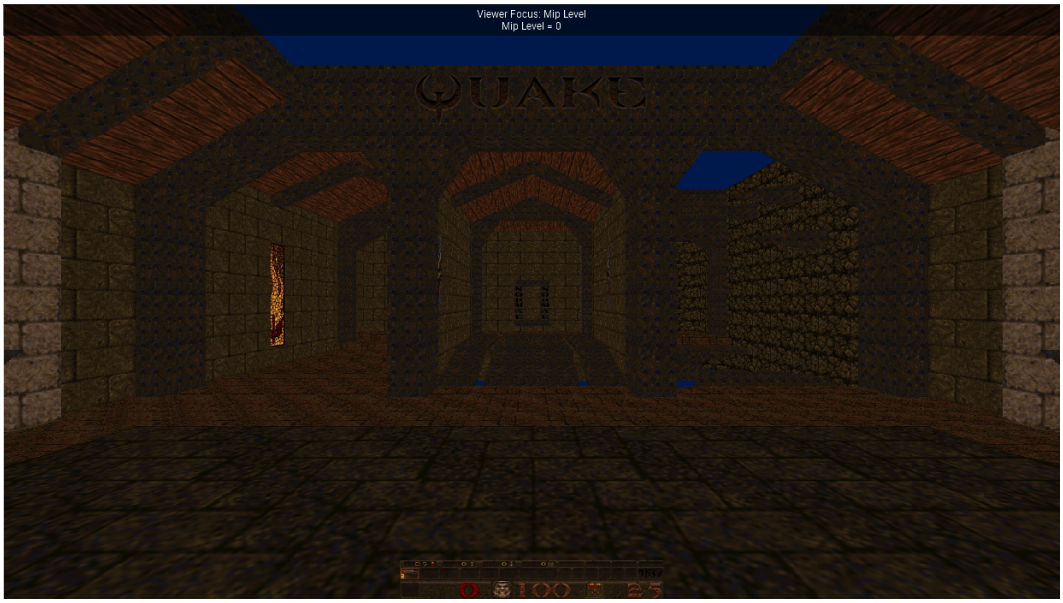
To minimize the required draw call of objects, levels were partitioned into a binary space partitioning tree structure, which allowed efficient traversal of polygons. The traversal allowed for a faster evaluation of level elements that were to be drawn on screen. This was not always efficient, as elements which have been drawn on the image could still be 'overdrawn' by other elements. This is also known as the 'overdraw' problem.

To further reduce necessary draw calls, Quake developed the 'potentially visible sets' technique to evaluate all visible polygons at each node of the BSP tree [30]. The general idea was to discard all surfaces that were not in the field of view or are behind other elements. This was done as a pre-process step and could be evaluated at runtime to determine which polygons can be excluded from the draw list. This technique, however, provides a significant problem for ray tracing applications.

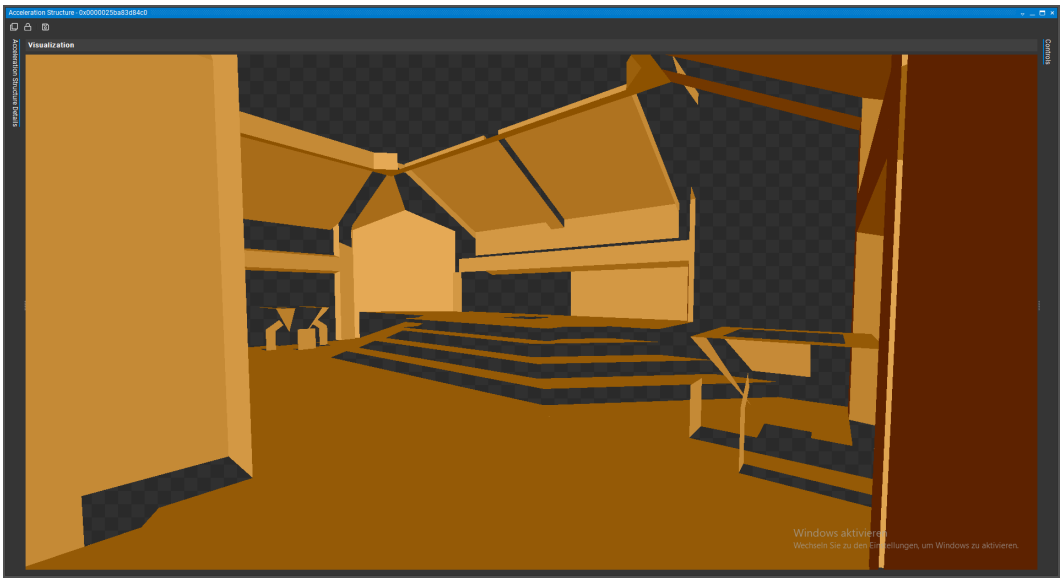
ii. Use in ray tracing

The omission of elements outside the view frustum leads to graphical glitches and false invocation of 'miss' results in the ray tracing pipeline, as rays can propagate in all directions in a virtual scene.

The following figures illustrate level geometry with enabled PVS.

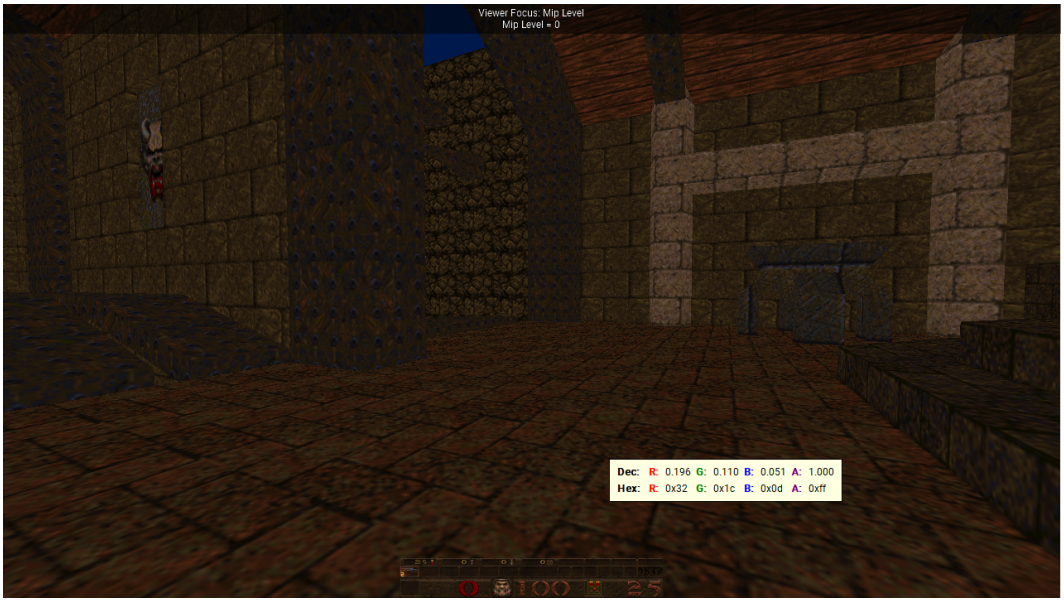


(a) *Position of view from spawn point*

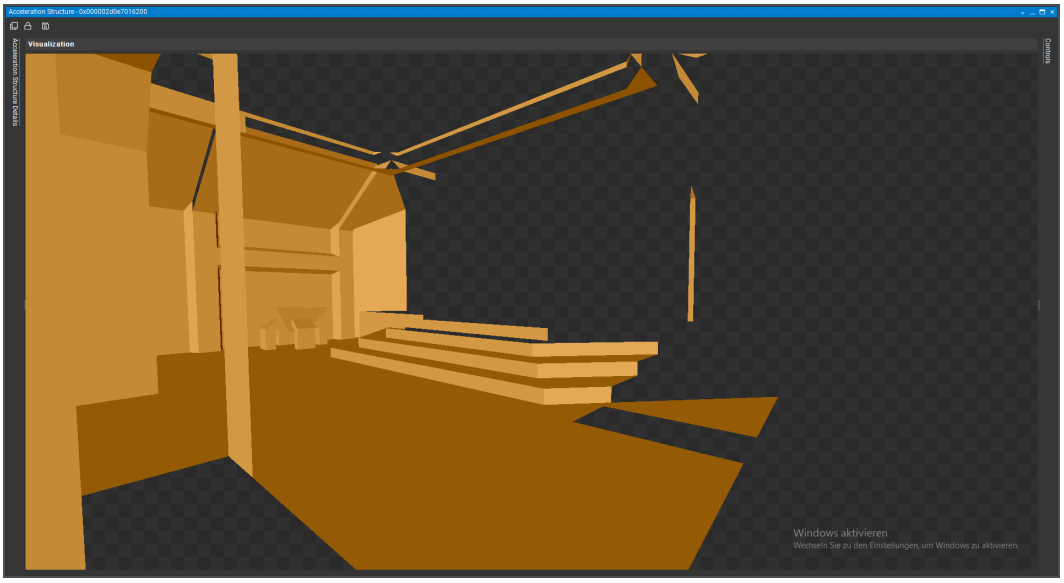


(b) *The drawn level geometry around him*

Figure 4.2.: *Elements not directly visible to the player are culled. The culling is applied to separate parts of a model.*



(a) *Player looks to the right side.*



(b) *The drawn level geometry around the player at new position and view*

Figure 4.3.: *Occlusion of models has changed. Elements on the left are drawn completely, the right side is culled more that in the previous illustrations*

The preloading of elements in a common buffer, as discussed in the previous section, solves this problem automatically for static objects, as the elements are already present in a buffer and thus can be excluded from the draw list.

Dynamic objects, that are subject to change each frame, still depend on a draw list, since the models provide updated transformation matrices, animation frames and other attributes, which require a rebuild of the model buffer.

It is therefore necessary to disable any techniques that allow geometry to be omitted.

This includes the PVS system as used in Quake, but also techniques like back-face and frustum culling.

III. Light calculation

The available light information and rendering method for lighting depends heavily on the game engine and may provide more or less information to the programmer. With 'vkQuake', some custom data structures had to be built in order to provide light information for the path tracing solution. In addition, previous lighting methods had to be disabled to prevent interference with the path tracing solution.

i. Lightmapping

The original 'Quake' as well as the 'vkQuake' source port use light-mapping to achieve light rendering for its models. Light-mapping is a technique where a light simulation is pre-calculated and results are saved as a texture. This texture is used as a form of lookup table to evaluate the illuminance for model surfaces.

The lightmap solution in 'vkQuake' is used for static and dynamic geometry elements alike. The solution is also dynamic, meaning that the lightmap texture can be updated at runtime. It is used to simulate dynamic lighting scenarios, such as flickering lights and other similar light fluctuations caused by light sources.

ii. Light entities

The path tracing method needs exact position data of individual light entities to perform its light simulation. Path tracing also performs its light simulation in real time so that the implemented light mapping method is not needed.

'vkQuake' has a particular problem, where the light entities found in the game files are not used in the games code. The light entities are solely used in the lightmap precalculation step, which happens outside the application by another software. It is therefore necessary to parse the light entities from the game file and construct a new light entities data structure. But the parsing reveals two problems.

For one, the light entities used in Quake's level format use so-called 'fill lights', which are invisible light sources used to artificially light up the environment. These can be filtered out to remove them from the light entities list and are therefore not a concern.

The second problem is that the parsed light entities are not complete. Textures that appear to emit light, such as lamps, screens and other glowing elements, do not have a dedicated light entity assigned to them and have therefore no impact on the lighting simulation. This ignores most available light sources in the game, as apart from the fill lights, which are ignored, only six model types are assigned a light entity. This makes the light entities found in the level file unusable for the most part.

iii. Light emitting textures

Instead of a list of point light entities, a list of textures and their position can be created. Textures that appear to emit light are marked as 'light emitting textures', so that the shader can identify the textures as valid light sources.

IV. Path tracing methods

Path tracing applications, primarily offline-render, use a path tracing method that, except for some modifications, is universally applicable. Current frameworks and sample projects show a method of path tracing that is used exclusively for real-time path tracing applications and differs from the path tracing method found in rendering software.

i. Naive Monte-Carlo path tracing

The brute-force or naïve Monte-Carlo path tracing pipeline approximates a color value for each pixel in an image by repeatedly dispatching rays and sampling light information. The detail of the image increases with the evaluation of multiple samples. Over time, the image "converges" and as a result, displays a realistic depiction of the scene. Depending on how interactive the application is, or has to be, there are two main methods of how images are built.

Real-time renderers use low sample counts to gather light information per frame. The light information is subsequently added to the previously rendered frame. When the view does not change, the image gains more detail over time, as new light information gets gradually added. Once the scene or camera changes, the rendering begins anew. The advantage of this method is that the low sample rate means that each image takes a relatively small time to render. The frames per second increase and changes in the scene can be applied faster. This is usually used for previews.

It is also possible to use a high sample rate to get the finished image immediately or gradually, by subdividing the image in multiple render threads. However, once rendering starts, it must be completed or aborted mid rendering. It is usually done once the scene or camera is finally set up and is used for the final render.

The brute-force approach aims for an accurate rendition at the cost of computation time. It relies purely on the repeated sampling of the scene and is therefore used primarily in offline 3D rendering software to achieve rendering of photorealistic imagery. A prominent example is Autodesk's 'Arnold' path tracing renderer or Pixar's 'RenderMan'.

For real-time applications, this architecture is not well suited, as the multiple-sample approach either results in a low frame rate, or in visible ghosting, as the previous result is kept in the next frame.

ii. Low-sample path tracing

The low-sample path tracing shader method is a modification of the brute-force path tracing architecture. Instead of waiting for a frame to complete rendering or using the result from the previous frame, a new image is generated each frame. The aim is to use low-sample rates ranging from 0.5 to 1 samples per pixel. The noisy image data is reconstructed using various denoising algorithms to achieve a noiseless image.

For a better denoising result, the sampled light information is divided into different channels. A common structure found among sample projects using a path tracing render and current state-of-the-art frameworks, is the division of light information by diffuse and specular distribution as well as light information sampled by direct illumination. These channels have different properties of noise and variance levels, which require different denoising algorithms to reconstruct an image in the best possible way.

External frameworks like Nvidia's Real-Time Denoising (NRD) [31] for example, provide solutions for diffuse, specular/reflective and shadows using the 'ReBLUR', 'SIGMA' and 'ReLAX' denoising algorithms.

The advantage of this architecture is the speed acquired by the low sample rate and the use of said denoising filters. The current state-of-the-art denoising algorithms reconstruct images exceptionally well, so that high sample rates are not always required. Therefore, real-time applications use this architecture and are seen in many modern path traced game solutions.

The disadvantage of this architecture is the dependence on denoising algorithms and the sampling strategies used by the path tracers. The result heavily depends on the noisy input signal that the path tracer returns.

V. Requirements analysis

After defining the necessary changes in the data structures, some functional and non-functional requirements arise. These requirements are based partly on the game itself and partly on the graphical changes as a result of the modification. The listed requirements are used as a quality indicator in a future chapter to evaluate the quality of the modification.

i. Functional requirements

Preservation of existing functions outside the game: The game should retain all previously known functions that were available outside the game. This includes menu navigation and the successful execution of settings present in the menu.

Preservation of existing functions within the game: The game should retain all previously known functions that were available within the game. This includes the

movement of the game character, the interaction of the game with the game environment, and others.

Preservation of existing game and business logic: The game should retain all built-in game logic. Functionalities that progress gameplay should therefore work as they had in the original game.

Implementation of required methods and data structures for hardware-accelerated ray tracing: The modified software should implement every method and data structure that is necessary for the implementation of hardware-accelerated ray tracing. This includes the use of acceleration structures, specific command calls and ray tracing graphic pipelines.

Successful launch of the path tracing pipeline: Provided that the user meets the hardware requirements for the modification, the user should be able to launch the modified application with no restrictions. If the hardware requirements are not met, the user should be notified accordingly.

Implementation of a denoising solution: The game should apply some form of denoising to the coarse path traced output.

Support for different materials: The game should support the representation of different materials to showcase the advantages of the path tracing solution.

Presentation of all visual elements: The game should retain all visual elements that were available in the original game. All previously used assets and their representation in the game are preserved in the modification. This includes models, textures, particle effects and dynamic lighting effects.

ii. Non-functional requirements

Performance: The game should run at a reasonable performance that corresponds to the capabilities of the used hardware. Due to the more expensive method of rendering, a user should not expect the same or similar performance as the unmodified game while using the same hardware configuration.

Visibility: The game should render objects and scenery in a way that is recognizable, so that visibility should not have an impact on the playability of the game.

Visual Identity: The game should render objects and scenery in a way so that the user is able to draw parallels to the unmodified version of the game. Familiar elements of the game should therefore still be recognizable despite the modified graphics pipeline.

Stability: The user should not find any bugs, crashes or memory leaks that were not present in the unmodified version.

iii. Missing functionalities

Due to time constraints and unforeseen technical problems that are to be expected during development, not all wanted features ended up in the prototype. In this section, a list of not implemented functionalities is listed that is sorted by the specific domain of the problem.

Material system: Quake's original rendering does not support a material system. The material is solely implied through texturing. Therefore, opaque, metallic or reflective surfaces have to be added manually, which requires a reimplementation of the current model data structure. At this stage of development, every surface is treated as opaque diffuse material.

Importance Sampling: The current limitation of the available material system impacts the ray tracing shader. At this point, the path tracing renderer uses uniform hemisphere sampling to evaluate the direction of the next ray bounce. A diffuse BSDF could be used to simulate a more realistic rendering of diffuse materials, but in reality, uniform sampling and diffuse BSDF importance sampling yield very similar results. It has therefore been decided to use uniform hemisphere sampling for the time being, as all surfaces are diffuse.

Next Event Estimation: The 'Light' subsection gives a theoretical implementation of a light entities buffer. Although an implementation exists in the prototype for point light entities that are parsed from the level files, the light emitting textures are missing in this list, which make up the majority of light sources in the game. Therefore, next event estimation and multiple importance sampling techniques cannot be applied at this stage of the implementation. Instead, only uniform hemisphere sampling will have a valid contribution.

Low-sample path tracing method: For the prototype, a brute-force approach has been initially considered, which would develop over time to use the low-sample path tracing method. The prototype in this stage does not use the techniques that the path tracing method recommends, i.e. the separation in different channels and use of denoisers.

PVS for dynamic models: Although static level geometry is not affected by culling of any kind, dynamic models disappear after a certain distance. This behavior is certainly linked to the PVS system and could not be disabled. It results in dynamic light entity models to have no light impact after a certain distance.

5. Implementation

This chapter deals with the implementation of individual components and structures, which are based on the ideas presented in the concept chapter. The implementation includes the general design and implementation details presented by code fragments and illustrations.

I. Graphic pipeline

The various graphics pipelines used in quake are to be replaced by specific ray tracing pipelines. The change affects graphics pipelines that were previously responsible for the rendering of dynamic, static, and transparent models.

Graphic pipelines responsible for the rendering of 2D textures and post-processing effects are still used to maintain rendering of user interface elements and post-processing effects like brightness adjustment or anti-aliasing.

i. Ray tracing pipeline

The ray tracing pipeline is a graphics pipeline that defines shader stages, which are invoked depending on the different 'states' of a dispatched ray. In addition, a so-called "Shader Binding Table" is created, which contains all ray tracing shaders that can be invoked.

In Vulkan, the usage requires the *VK_KHR_ray_tracing_pipeline* extension [32] as well as the *VK_KHR_acceleration_structure* and *VK_KHR_deferred_host* extensions.

The shader binding table is a collection of up to four arrays, also called shader groups, that contain the handles of the defined shader modules:

- Ray Generation Shader Group
- Miss Shader Group
- Hit Shader Group
- Callable Shader Group

The shader modules are the explicit shaders that are called when a ray changes its 'state'. The shader modules involve:

- The 'ray generation shader', that is responsible for the generation and dispatch of rays.
- The 'miss shader' that controls the behavior in case a ray hits no element in the acceleration structure.
- The 'closest hit shader' that is invoked when the closest intersection has been determined.

- The ‘intersection shader’ that computes the ray intersection and can optionally be rewritten to define custom intersection calculations for custom geometry.
- The ‘any hit shader’ that gets invoked at any hit that occurs. This is primarily used for transparent geometry.

The invoked shader modules help distinguish the ‘kind’ of hit a ray has evaluated when traversing the scene. The concrete logic for correct shading of each element has to be evaluated and calculated inside each individual ray tracing shader.

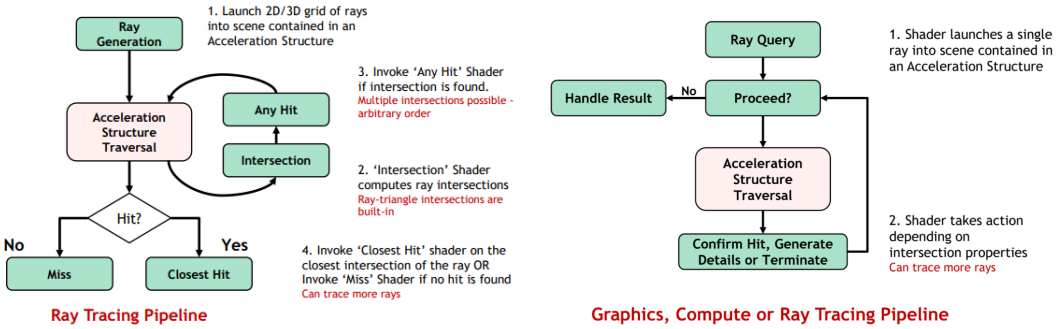
ii. Ray tracing query

Alternatively, ray queries or inline ray tracing provides the possibility of dispatching individual rays without the construction of a separate ray tracing pipeline and shader binding table. This has the advantage that ray queries can be executed from any shader types, including all graphic and compute shader.

In Vulkan, the usage requires the VK_KHR_ray_query extension [33]. It requires the VK_KHR_acceleration_structure and VK_KHR_deferred_host extension as well.

Unlike the ray tracing pipeline, ray queries cannot directly invoke the execution of other ray tracing shaders. This means that a hit result from a ray query will not invoke a “hit” shader.

Instead, each ray dispatch is queried and returns an immediate result that can be used by the shader for further calculations. Ray queries are therefore used primarily in raster-ray pipelines for effects such as shadows and reflections.



(a) *Ray tracing pipeline flow diagram.* **(b)** *Ray query flow diagram.*

Figure 5.1.: *The figures show the flow diagram of the ray tracing pipeline (a) and ray query (b) extensions.*

Due to the listed functionality of both options, the "ray tracing pipeline" is particularly suitable for the project, as the invocation of different shader stages is helpful to easily determine the current state of the ray and construct shader logic around it. The path tracing logic found in this modification could also be realized using ray queries exclusively.

iii. Shader binding table

For the successful creation of the shader binding table, care must be taken to ensure that the respective memory addresses of the shader groups are aligned correctly, since different graphics cards have different specifications on how shader groups have to be accessed. Otherwise the ray tracing shaders will not be executed correctly.

The current implementation uses four shader modules for the shader binding table:

- The "Ray generation shader", which is responsible for the generation of the rays.
- The "Ray miss shader", which describes what will happen in case a ray hits “nothing”
- The second “Ray miss shader”, which defines different behavior when a ray hit occurs.
- The "Ray closest hit shader", which is executed at the very first intersection with a geometry.

The buffer containing the shader binding table data must be explicitly marked as a shader binding table buffer, and its memory address must also be visible. This is ensured with the *VK_BUFFER_USAGE_SHADER_DEVICE_ADRESS_BIT* and *VK_BUFFER_USAGE_SHADER_BINDING_TABLE_BIT_KHR* usage property flags.

The size of the buffer is the sum of the sizes of all used shader groups aligned to the "shaderGroupBaseAlignment".

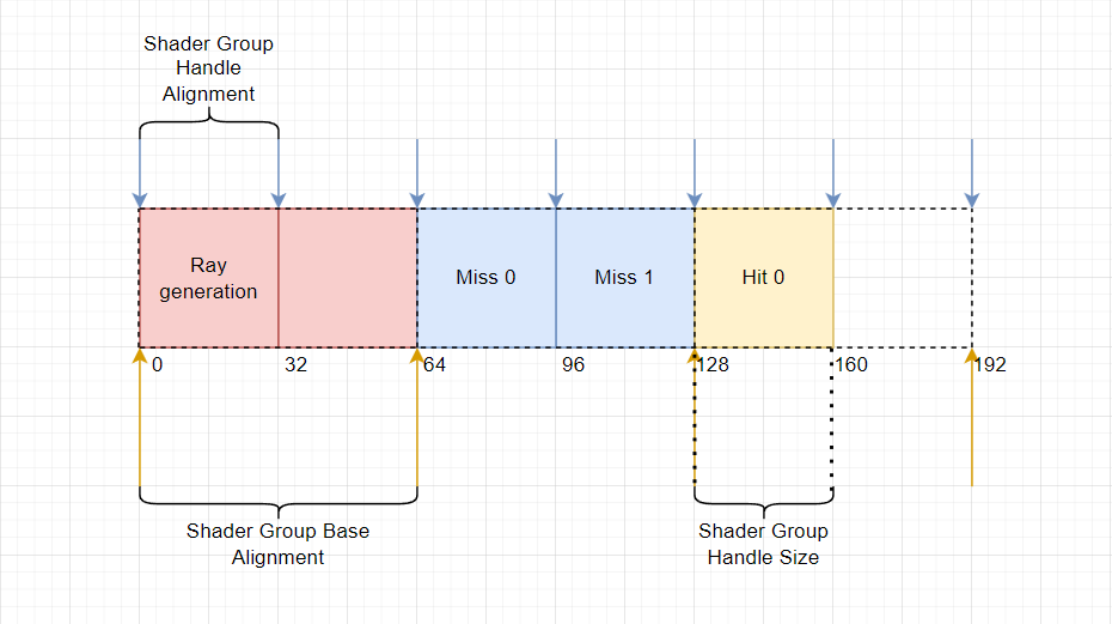
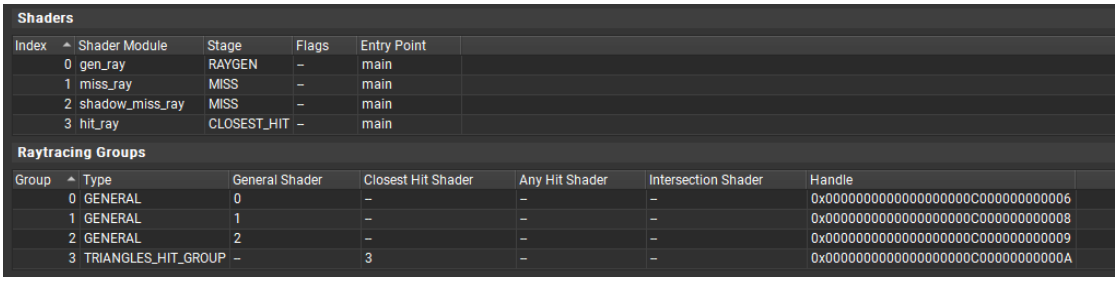


Figure 5.2.: *Illustration of a shader binding table: It shows the configuration of four shader module across three shader groups.*

Once the shader groups are correctly aligned, the shader binding table can be created. Debug Tools like ‘Nvidia’s Nsight’ can verify the correct implementation of the ‘shader binding table’ by checking the size and memory address of each shader module.



Shaders

Index	Shader Module	Stage	Flags	Entry Point
0	gen_ray	RAYGEN	–	main
1	miss_ray	MISS	–	main
2	shadow_miss_ray	MISS	–	main
3	hit_ray	CLOSEST_HIT	–	main

Raytracing Groups

Group	Type	General Shader	Closest Hit Shader	Any Hit Shader	Intersection Shader	Handle
0	GENERAL	0	–	–	–	0x0000000000000000000C000000000006
1	GENERAL	1	–	–	–	0x0000000000000000000C000000000008
2	GENERAL	2	–	–	–	0x0000000000000000000C000000000009
3	TRIANGLES_HIT_GROUP	–	3	–	–	0x0000000000000000000C00000000000A

Figure 5.3.: *All shader modules and shader groups listed in 'Nvidia NSight'.*

iv. Shader compilation

The ray tracing shaders are written in the OpenGL Shading Language (GLSL). However, Vulkan uses its own compile format, which enables the platform-independent use of shader code across multiple platform targets. It is called the Standard, Portable Intermediate Representation - V format, or SPIR-V for short. Before a shader can be used, it must be compiled into a SPIR-V format.

'vkQuake' provides a batch file to convert vertex, fragment and compute shaders to the SPIR-V format. However, ray tracing shaders do not belong to any of these shader formats, so an adaptation of the batch file was necessary to support ray tracing shaders.

The 'glslangValidator' provided by Vulkan allows the compilation of GLSL shaders into the '.rspv' ray tracing shader format. For easier differentiation of the different shader modules, the file extensions have been selected with respect to their shader group. Thus, the written ray tracing shaders can be found by the following '.rgen', '.rchit' and '.rmiss' file extensions. The file extensions can be named freely, as long as Vulkan recognizes the file as a ray tracing SPIR-V module. The compiled shaders can then be used as shader modules in the shader binding table.

```
1 for %%f in (*.rgen, *.rchit, *.rmiss) do (
2   %VULKAN_SDK%\bin\glslangValidator.exe --target-env vulkan1.2 %%f -o
    Compiled/%%~nf.rspv
3   bintoc.exe Compiled/%%~nf.rspv %%~nf_ray_spv > Compiled/%%~nf_ray.c
4 )
```

Listing 5.1: *Batchfile command for '.rspv' compilation*

The implementation of the ray tracing extension, the shader binding table and the generation of compatible ray tracing shader files fulfill the formal requirements for using the ray tracing extension and the ray tracing pipeline.

II. Data structures

The unmodified Quake game provides the *'R_RenderScene'* method, which contains all rendering methods that invocates all 'draw calls'.

The method consists of the following methods:

- *R_SetupScene*

- *Fog_EnabledGFog*
- *R_DrawWorld*
- *S_ExtraUpdate*
- *R_DrawEntitiesOnList(false)*
- *Sky_DrawSky*
- *R_DrawWorld_Water*
- *R_DrawEntitiesOnList(true)*
- *R_DrawParticles*
- *Fog_DisableGFog*
- *R_DrawViewModel*
- *R_ShowTris*

The methods provide access to the individual elements that are drawn each frame. In some cases, this draw list can be repurposed for building a collective buffer. In other cases, the drawing method has to be rewritten.

i. Models

The *'R_DrawWorld'*, *'R_DrawEntitiesOnList'* and *'R_DrawViewModel'* methods are responsible for model rendering.

Static models: The *'R_DrawWorld'* model uses the PVS system to determine the applicable static geometry polygons for the current frame. As the PVS system causes issues in ray tracing and path tracing, as seen in the concept chapter, the system must be disabled. As the level geometry is completely static, it is more efficient to load it once at level load, then to iterate a polygon list.

The *'R_DrawWorld'* method is therefore omitted and instead retrieves the parsed level file from the internal cache manager and loads each model into a static vertex and index buffer. The corresponding method is called *'GL_BuildBModelRTVertexAndIndexBuffer'*.

This procedure is in fact already used by Quake, but the vertex structure of the static and dynamic model files does not suit the needs of the ray tracer.

For the uniform rendering of 3D models, a custom *'rt_vertex_t'* data structure is used to make texture and material lookup easier in the ray tracing shader.

```
1  typedef struct rt_vertex_s {
2    float vertex_pos[3]; # vec3 world position
3    float vertex_tx_coords[2]; # vec2 texture coordinates
4    float vertex_fb_coords[2]; # vec2 texture coordinates for 'fullbright'
     textures
5    int tx_index; # index of texture in texture array
6    int fb_index; # index of fullbright texture in texture array
7    int material_index; # index describing the associated material
8  } rt_vertex_t;
```

Listing 5.2: *The rt_vertex_t struct*

The buffers in mind use the *VK_BUFFER_USAGE_STORAGE_BUFFER_BIT*, *VK_BUFFER_USAGE_ACCELERATION_STRUCTURE_BUILD_INPUT_READ_ONLY _BIT_KHR* and *VK_BUFFER_USAGE_SHADER_DEVICE_ADDRESS_BIT* usage flags to signalize that the buffer in question is a storage buffer, which is used for the building of acceleration structures. In addition, the acceleration structure also needs access to the memory address, which is set with the last flag.

The vertex buffer uses the *VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT* and *VK_MEMORY_PROPERTY_HOST_COHERENT_BIT* memory flags to enable memory mapping on the CPU side. This is done as some methods require read access for drawing dynamic models. A rewrite of the methods has to be considered, as the memory flags reduce the read and write speed of the memory. The index buffer uses the *VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT* memory flag, to ensure the fastest read and write speed possible.

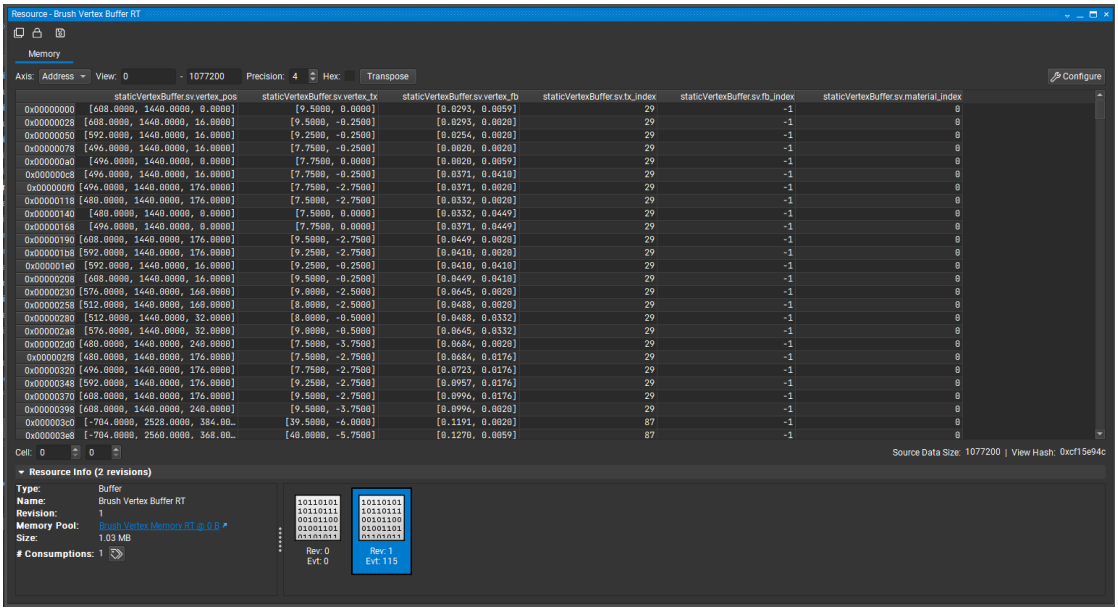


Figure 5.4.: *The byte data of static environment models in the static vertex buffer.*

Dynamic models: The *'R_DrawEntitiesOnList'* method is responsible for the rendering of dynamic opaque and transparent models. Internally, this method checks whether the dynamic model from the draw list is opaque or transparent and determines the kind of dynamic model.

A dynamic model can either be a static level element with a transformation matrix, such as a movable wall, an animated model such as an enemy, or a 2D texture for particle effects. The corresponding 'R_DrawBrush' and *'R_DrawAlias'* retrieve the information, perform vertex animation lerping and determine the texture indices and material index, similar to the procedure for the static level geometry. Dynamic vertex and index buffers hold the data of the dynamic models. The internal methods of 'vkQuake' keep track of the correct alignment and allocation of dynamic buffers.

At last, the *'R_DrawViewModel'* is solely responsible for the rendering of the view model. The view model is an animated 'Alias Model' and therefore uses the *'R_DrawAlias'* method to add the geometry to the dynamic model buffer. The view model was separated to prevent clipping with the world geometry. Since dynamic geometry models are in the same domain, this issue will reappear.

ii. Textures

Textures are loaded using the *'TexMgr_LoadActiveTextures'* method. It uses the *'active_gltextures'* data that 'vkQuake' provides, to get a list of textures that are actively used in the level.

In contrast to models and other data, the textures are not loaded in a buffer. The shader uses a list of 'sampler2D' objects instead. To implement this, the shader needs access to allocated memory filled with *'VkDescriptorImageInfo'* data structures. Instead of the texture itself, the data requires a *'VkImageView'*, *'VkSampler'* and *'VkImageLayout'* data structure. These define how the texture is laid out in the memory and how it is sampled by the shader [34].

Additionally, the entries of the *'active_texture'* pointer are randomly ordered. An ordering is applied to match the order present in the 'texture' heap, where the textures also reside. It is also possible to fetch the texture data directly from the heap, as both hold the same texture data. However, this was discovered later in development.

Beside the list of textures, the shader needs to know which texture to sample from at any ray hit. As mentioned earlier, each draw call that was previously issued had the exact model and other information.

The texture belonging to the model, as well as the correct texture coordinates for each vertex, are present in the draw list. The only thing left to do was to determine the index of the texture in the heap, by traversing all nodes backwards until we hit the start node. The determined texture index is written for each vertex in the *'rt_vertex_t'* struct, for fast lookup in the shader.

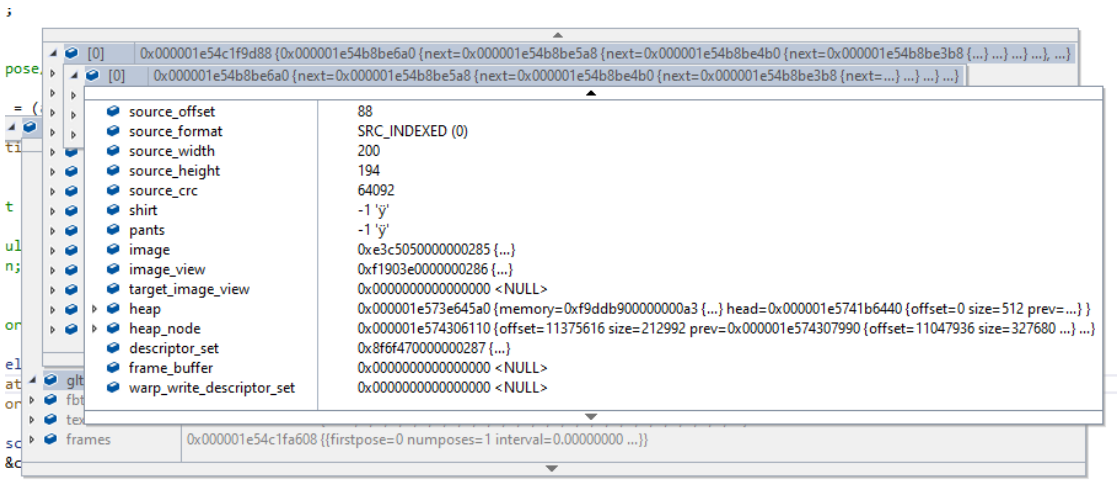


Figure 5.5.: *The 'gl_texture' instance allows access to the heap and heap node. The heap node can be traversed backwards.*

This works, but it can be better. Instead of determining the texture index each time a model is passed, the methods providing the current draw list can be modified so that the index is precalculated. This is not the case in the current version of 'vkQuake'.

The lookup of the texture index may be efficient, but not efficient in terms of memory space. Essentially, the same texture index is applied for every vertex in a model. Referring back to the illustration, it can be observed that the *'tx_index'* and *'fb_index'* entries repeat.

iii. Acceleration structures

At this point, models and textures have a unified buffer structure that ensures the presence of all available entities needed for the path tracing implementation, with the exception of the light entities buffer.

In the following steps, the acceleration structures are built that allows for the traversion of rays in the scene. As mentioned in the 'theoretical background chapter', the use of acceleration structures is mandatory, as every ray dispatch call requires a reference to an acceleration structure.

The way acceleration structures are structured depends on the individual use case and scope of the project. The semantically correct usage for bottom-level acceleration structures is the encapsulation of individual geometry objects. For projects with large object counts, this becomes non-optimal. There are two reasons for this.

For one, each creation of an acceleration structure issues a command to the command-buffer, which is responsible for the execution of GPU commands. The invocation itself adds additional performance overhead. It is therefore better to create one acceleration structure for multiple geometries than building multiple acceleration structures for a single geometry.

The second reason is that top-level acceleration structures become non-optimal when the axis-aligned bounding boxes of the BLAS instances overlap. It is therefore recommended to merge multiple geometries and use as few bottom-level acceleration structures as possible.

For the developed model structure, the creation of a static and dynamic bottom-level acceleration structure instance is therefore most useful. As mentioned in the model section of the implementation, only opaque objects are considered. For the implementation of transparent objects, a third BLAS instance could be considered.

The static bottom-level acceleration structure is created at level load, similar to the static geometry models. The created BLAS instance is reused for the duration of the level and recreated when a new level is loaded. The build process takes an average of 1.12 milliseconds to perform according to the 'Nvidia Nsight' profiler.

The dynamic bottom-level acceleration structure is created each frame, as the draw list and the properties of the dynamic models also change at each frame. Depending on the scene, different readouts showcase an average of 0.43 milliseconds build time. The build time presented is not representative of the final debug or release build performance, as the profiler uses extensive resources, which alters the actual numbers.

The creation of bottom-level acceleration structures requires access to the vertex and index buffer, as well as a description of the underlying vertex structure for the correct interpretation of the bytes. It expects a list of vertex position values and indices. Since vertices contain more than position data, the bottom-level acceleration structure needs to know how the position data can be acquired.

The vertex format specifies in which format the vertex position is given. In the *'rt_vertex_t'* struct the position is given as *float[3]*, so the vertex format is given as *VK_FORMAT_R32G32B32_SFLOAT*. The index type specifies what data size the index values have. The static geometries use the *uint_16* and the dynamic ones the *uint_32* data type. It is also possible to create a bottom level acceleration structure without an index buffer, if the vertex data has the correct ordering.

The stride value specifies the size of the vertex structure. In case of the *rt_vertex_t* struct, it has a size of 40 bytes

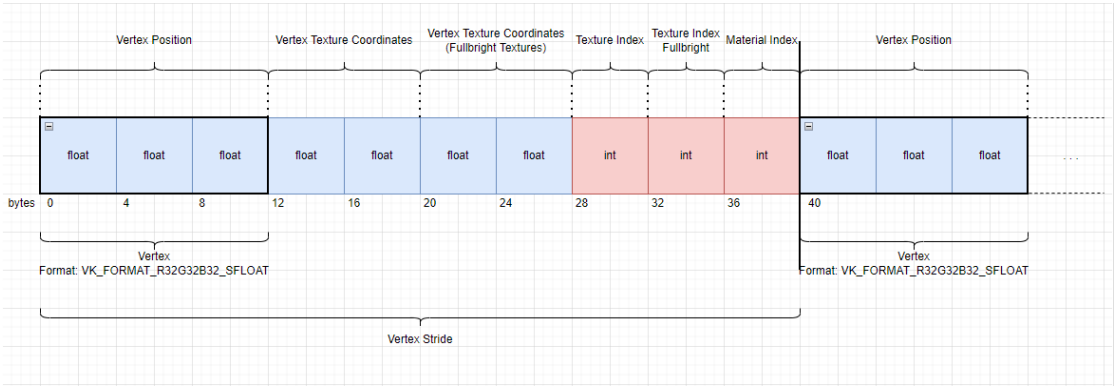
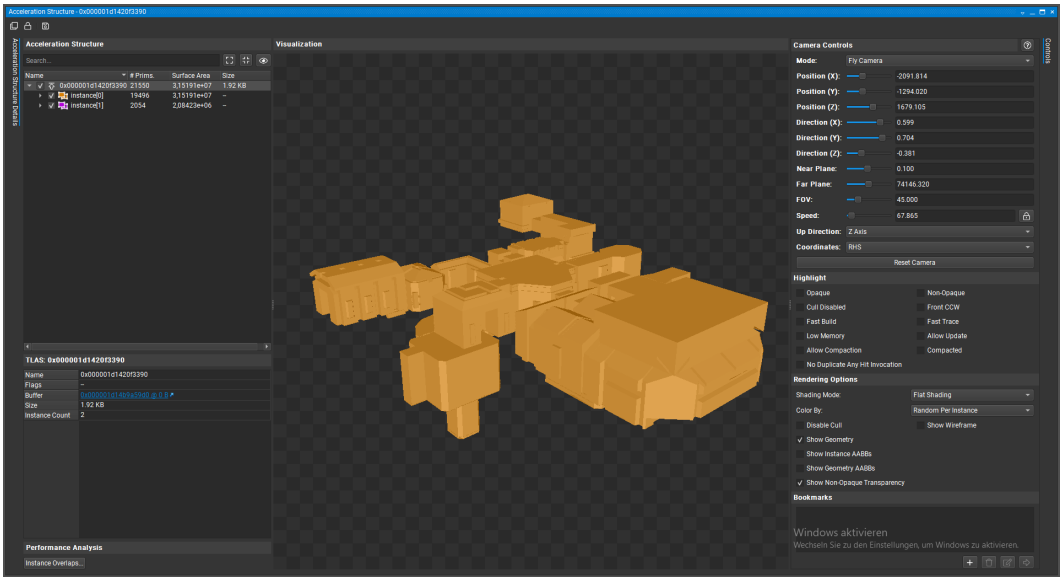


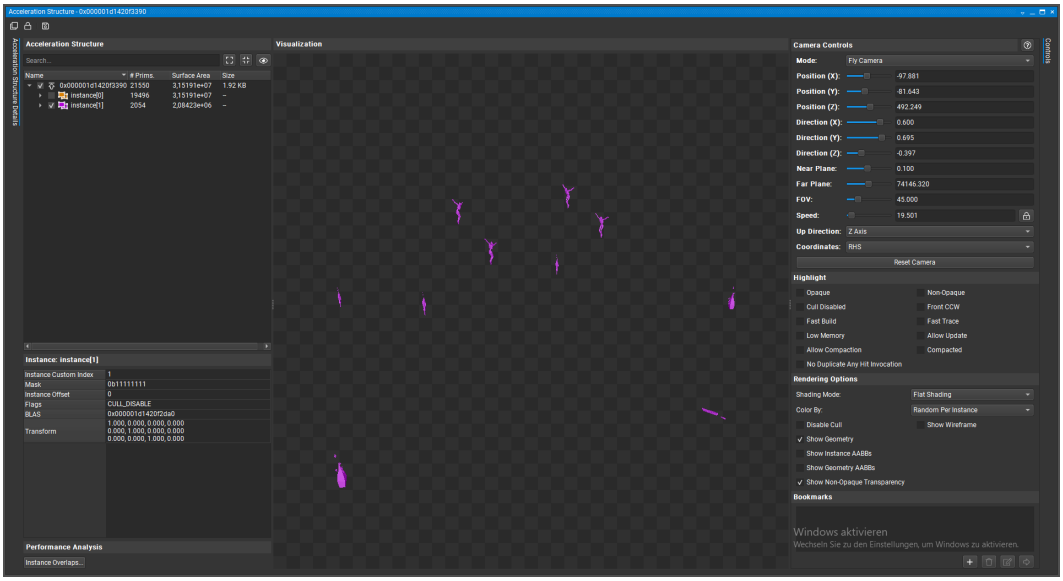
Figure 5.6.: *The rt_vertex_t data structure. The bold border around the vertex position is what the acceleration structure reads. The vertex stride tells the acceleration structure the byte distance to the next vertex position.*

Once it is set, additional scratch buffers and buffers for the acceleration structure itself have to be assigned for the final build step on the GPU.

The two bottom-level acceleration structures instances are used to build the top-level acceleration structure that encapsulates both structures. The generation process is virtually identical with the exception of the *VkAccelerationStructureGeometryDataKHR* struct where the geometry type is set to accept 'instances' instead of 'triangles'.



(a) *The static bottom-level acceleration structure*



(b) *The dynamic bottom-level acceleration structure*

Figure 5.7.: *Both static (a) and dynamic (b) acceleration structures are part of one top-level acceleration structure.*

III. Shaders

This section deals with the implementation of the individual shader stages of the brute force path tracing method. As the ray traverses the acceleration structures, the shader modules ensure that at each state of the ray, a correct set of data is calculated and returned for evaluation of the final pixel value.

i. Descriptor Sets

The data that is passed to the shaders, has to be set up using a 'descriptor set'. It informs the shader about the data it can expect, the size of the data and which ray tracing shaders can access the data. This table view gives an overview of all data storages that the shaders have access to.

Table 5.1.

Data	Description	Shader Groups
Top Level Acceleration Structure	The top level acceleration structure, that is used for ray traversal	Hit Group, Ray generation
Ray Output Image	A image view instance that encapsulates the image	Ray generation
Uniform buffer	Uniform Buffer containing camera matrix current frame etc.	Hit Group, Ray generation
Static vertex buffer	Has vertices of static models	Hit Group
Static index buffer	Has indices of static models	Hit Group
Dynamic vertex buffer	Has vertices of dynamic models	Hit Group
Dynamic index buffer	Has indices of dynamic models	Hit Group
Texture list	List of active textures.	Hit Group
Light entities buffer	List of light entities (not used in current version)	Hit Group
Light entities index buffer	Sorted light entities index list. (not used in current version)	Hit Group

The *'RT_UpdateRaygenDescriptorSets'* method updates the descriptor sets at each frame. For each descriptor set update, the buffer instance, offset and buffer range is set.

Because Quake uses double buffering, the current dynamic buffer has to be updated, as the used buffer alternates each frame.

ii. Ray generation shader

The 'ray generation' shader handles the initial dispatch of rays and the dispatch of consecutive rays that have been reflected or refracted for a given pixel position. It is also responsible for the retrieval of color values, which the ray returns and for writing the final color value to the image.

The outer loop handles the invocation of multiple samples and the inner loop handles the dispatch of the initial and consecutive rays. The depth of these loops is determined by the *'maxSamples'* and *'maxDepth'* variables.

Sample loop: The 'sample' loop controls how often a ray is dispatched for the current pixel. As mentioned in the short introduction to path tracing, the 'Monte-Carlo' path tracing algorithm approximates the rendering equation by uniformly sampling a random direction from a hemisphere that surrounds the hit geometry and continuing the ray path in the sampled direction. With more samples per pixel, more directions from the hemisphere can be sampled, which results in more areas of the scene being considered.

The camera position and world space position are used to determine the start position and dispatch direction of the ray. A random number is determined to apply a small variation to the pixel position and direction to maximize the potential paths over time. The seed is acquired by using the current frame of the application in the 'FrameData' uniform buffer, which also stores the current camera matrices. The random variation also reduces shimmering effects around edges, which automatically improves image quality when multiple samples are used.

Each ray is given a reference to a *'hitPayload'* struct that is used to keep track of various information while traversing the scene. This struct is updated at each intersection in the inner ray iteration loop.

```
1  struct HitPayload
2  {
3    uint sampleCount;
4    vec3 contribution;
5    vec3 origin;
6    vec3 direction;
7    bool done;
8  };
```

Listing 5.3: *The ray payload objet*

Ray iteration loop: The inner loop is responsible for the dispatch of rays. The top-level acceleration structure is selected and other parameters, such as the origin point, direction, minimum and maximum range and the payload object are set. When the ray is

dispatched, the shader awaits the result. If an object is hit, the origin and direction of the payload object is read and assigned as the new origin and direction for further bounces. If the 'done' attribute of the payload object is set to 'true', the inner loop will stop. This means that the ray has hit a light source or hit nothing.

Once the inner loop is finished or has been canceled prematurely, the contribution attribute of the payload object is added to the *'summedPixelColor'* variable that, as the name suggests, is the sum of all contributions for the current pixel.

When the outer loop completes, the summed color value of the pixel is divided by the maximum number of samples to average the contribution of all samples. In the final step, the pixel value is written in the output image.

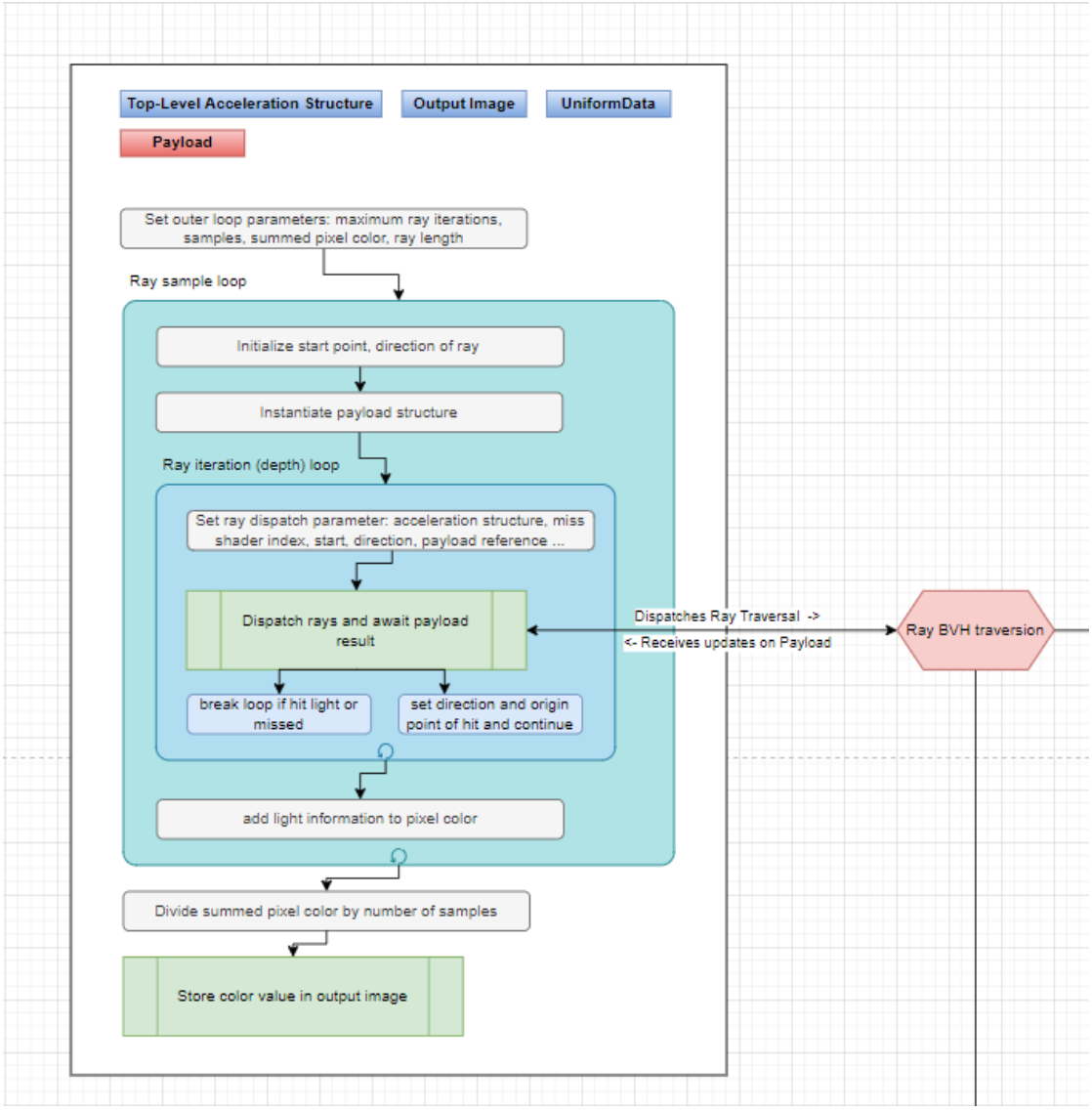


Figure 5.8.: *A visual representation of the ray generation shader.*

iii. Closest hit shader

The closest hit shader is called whenever a closest hit is generated during the evaluation of the ray path. As this prototype does not deal with transparent elements, every hit will invoke a closest hit shader call.

The shader determines the hit object by the *'gl_PrimitiveID'* and the *'gl_InstanceCustomIndexEXT'*. The *'gl_InstanceCustomIndexEXT'* returns a custom index that has been set for each individual bottom-level acceleration structure. In the acceleration structure creation process, a custom index of 0 and 1 has been set for the static and dynamic object acceleration structure.

The *'gl_PrimitiveID'* returns the index position of a primitive in the index buffer. Here, the primitive elements are triangles. The primitive ID takes the consecutive three indices that form a triangle mesh.

```
uvec3 getIndices(int primitiveId, int instanceId){
  int primitive_index = primitiveId * 3;

  if(instanceId == 0){
    return uvec3(staticIndexBuffer.si[primitive_index],
    staticIndexBuffer.si[primitive_index + 1],
    staticIndexBuffer.si[primitive_index + 2]);
  }
  else{
    return uvec3(dynamicIndexBuffer.di[primitive_index],
    dynamicIndexBuffer.di[primitive_index + 1],
    dynamicIndexBuffer.di[primitive_index + 2]);
  }
}
```

Listing 5.4: *Example: Method for indices retrieval*

```
Vertex getVertex(uint index, int instanceId){
  if(instanceId == 0){
    return staticVertexBuffer.sv[index];
  }
  else{
    return dynamicVertexBuffer.dv[index];
  }
}
```

Listing 5.5: *Example: Method for vertices retrieval*

Additionally, the barycentric coordinates are calculated with the world space hit coordinates that have been passed on as well.

The texture index and the texture coordinates provided by the vertices, are used to retrieve the corresponding texture from the texture array and sample the correct texture value by interpolation all three texture coordinates with the barycentric coordinates. The exact hit position is interpolated as well, and the geometric normal is also calculated.

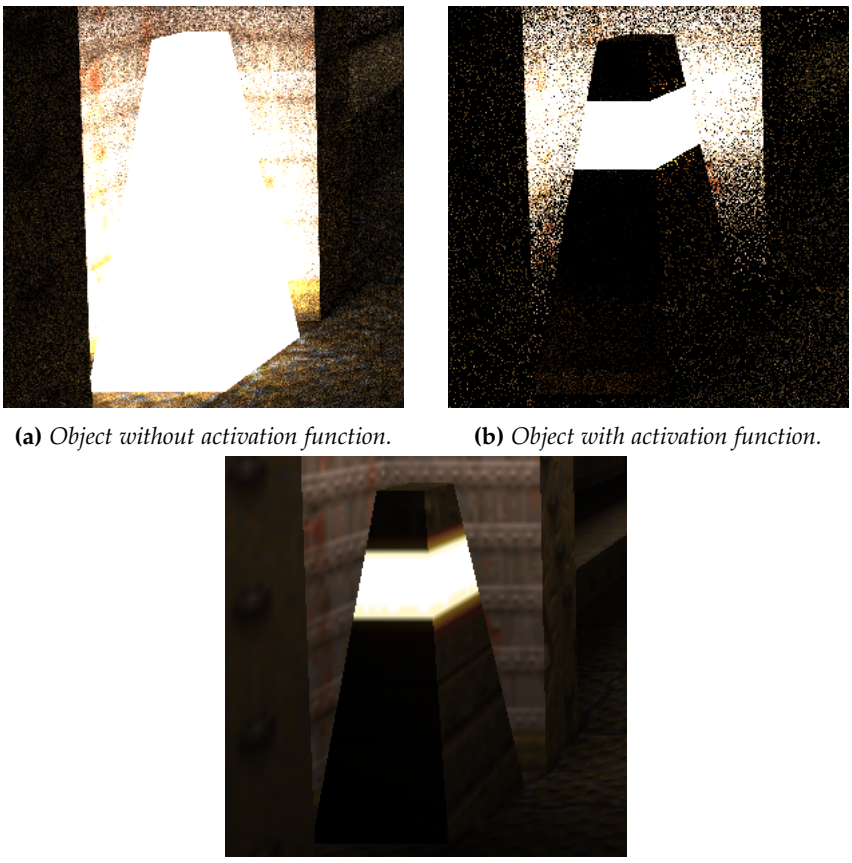
Models in Quake can have up to two textures. A 'gltexture' instance, which handles the default albedo and is affected by lightmaps and a 'fbtexture' instance, which stands for 'fullbright' textures that emulate emitting light sources and which are not affected by lightmaps.

The shader treats 'fullbright' textures as emissive light sources if the sampled pixel value is exceeding a certain luminance threshold. The shader applies a simple activation function to return the luminance value between 0 and 1 [35] If the relative luminance exceeds a defined threshold, for example 0.5, it will treat the surface as a light source.

```
1 float getRelativeLuminance(vec3 tex_color){
2   return tex_color.x * 0.2126 + tex_color.y * 0.7152 + tex_color.z *
    0.0722;
3 }
```

Listing 5.6: *Example: Simple relative luminance method. Returns float value from 0 to 1 range*

An activation function similar to this is needed, or the entire texture will be treated as a light source.



(a) *Object without activation function.* **(b)** *Object with activation function.*

(c) *Object in unmodified 'vkQuake'.*

Figure 5.9.: *Light emitting textures are checked for their relative luminance before the hit registers a light source.*

If the relative luminance exceeds the threshold, the currently hit object will be regarded as a light source. In this case, an emission term is calculated and the 'done' attribute of the payload is set to 'true', to indicate that a light source has been hit. Else, the emission term is omitted and the albedo term is assigned the current texture value.

Finally, a uniformly sampled random direction from the hemisphere is set as the ray direction for the consecutive bounce. The origin point of the new ray is the current hit point with an additional offset to prevent rays from accidentally hitting the same object that has been hit previously. The contribution of the payload is then multiplied by the current emittance plus albedo terms, which corresponds to the term found in the rendering equation

```
1  const float theta = M_PI * 2 * rand(seed);  // Random in [0, 2pi]
2  const float u     = 2.0 * rand(seed) - 1.0;  // Random in [-1, 1]
3  const float r     = sqrt(1.0 - u * u);
4
5  hitPayload.direction = geometricNormal + vec3(r * cos(theta), r * sin(theta
     ), u);
6  hitPayload.origin = position + 0.0001 * hitPayload.direction;
7  hitPayload.contribution *= luminance + albedo;
```

Listing 5.7: *Calculation of the random uniform hemisphere sampling and hitPayload assignment*

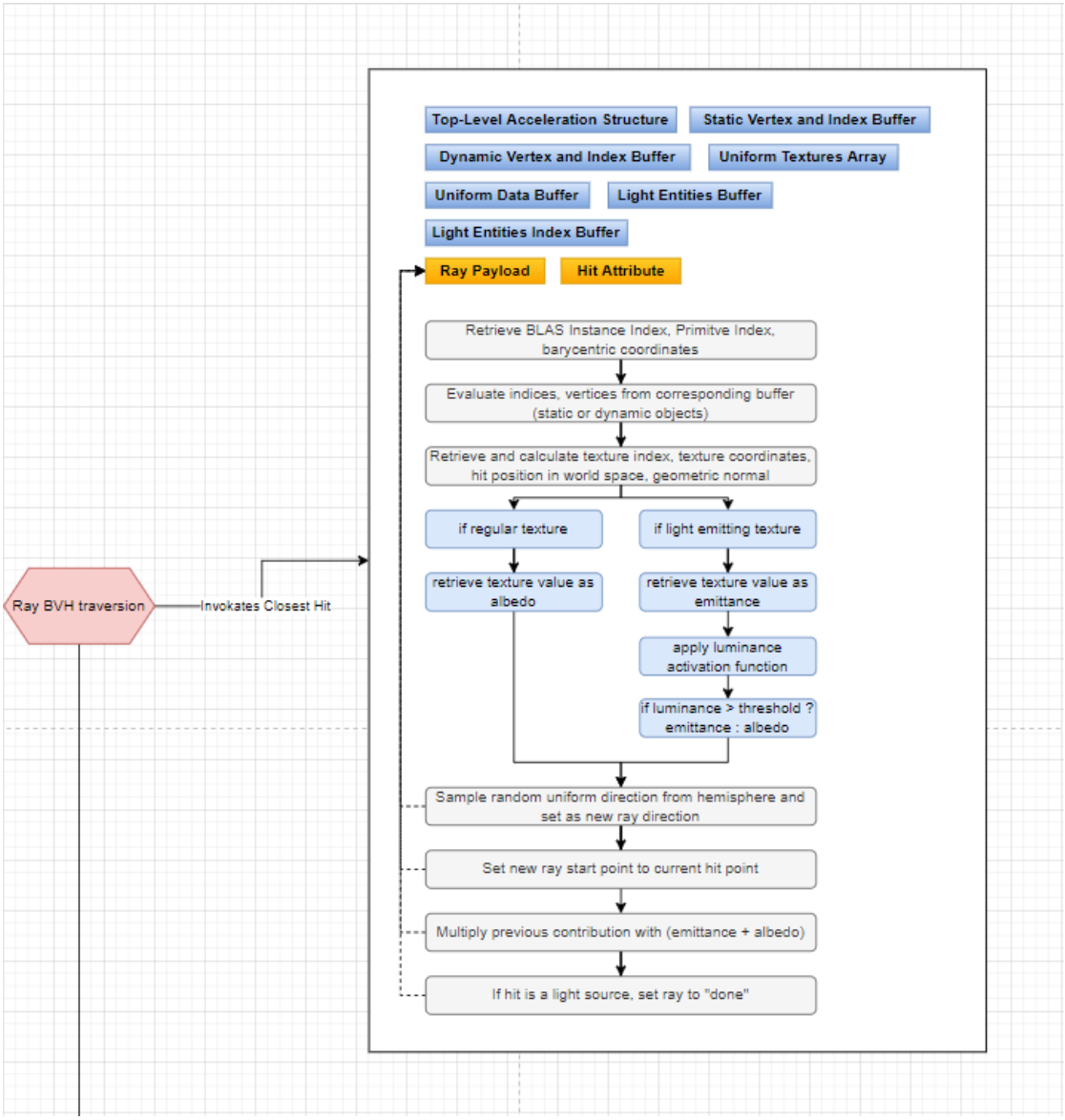


Figure 5.10.: *A visual representation of the closest hit shader.*

iv. Miss shader

The 'first miss' shader is executed each time the ray does not hit any geometry, while traversing the acceleration structure. In most cases, a color value like black is returned, to indicate that no light source has been hit that could radiate any energy to illuminate anything in the scene.

```
1  void main()
2  {
3    hitPayload.contribution = vec3(0);
4    hitPayload.done = true;
5  }
```

Listing 5.8: *Miss shader assigns zero vector indicating that no light energy can contribute to the ray path*

The second ‘miss’ shader is used to simulate the shadow effect in situations where single rays are sent that are not part of the main path tracing loops. In this prototype, the second ‘miss’ shader sees no use, however, it may be used for techniques like next event estimation.

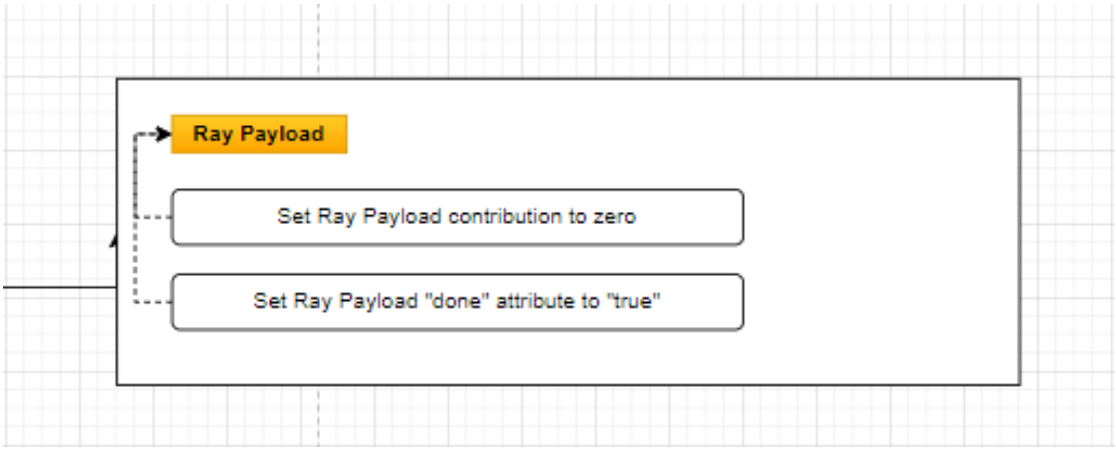
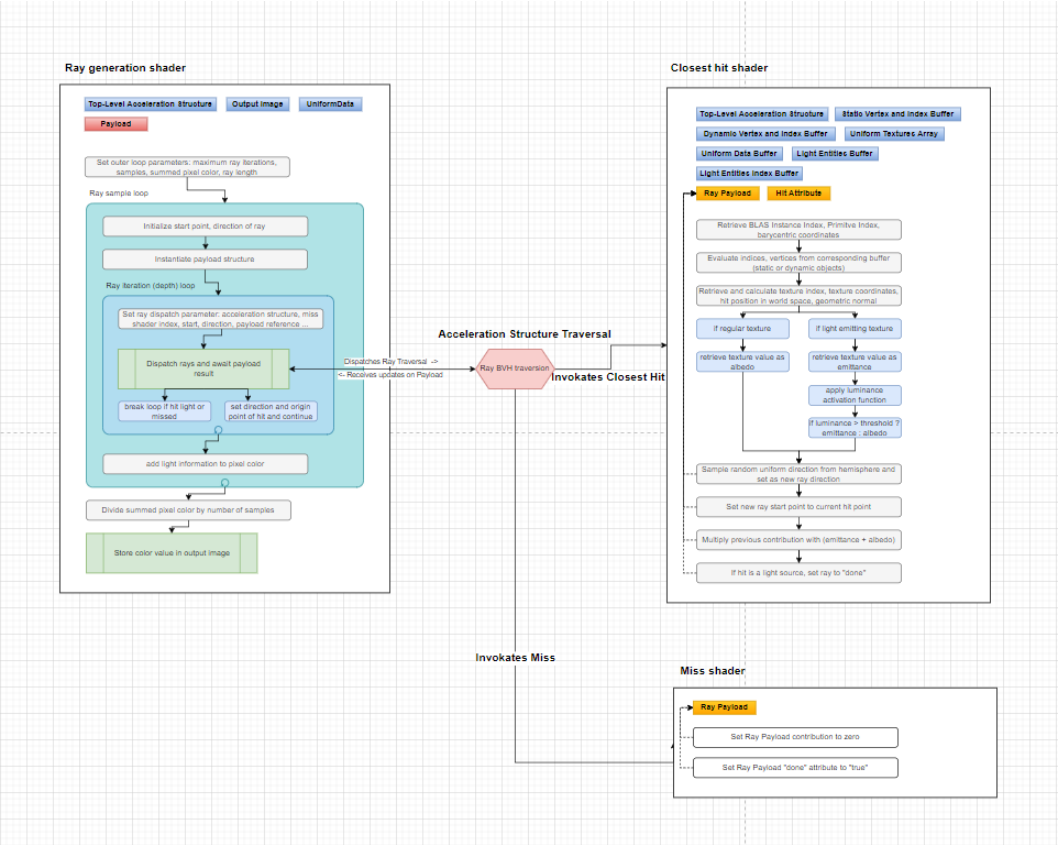


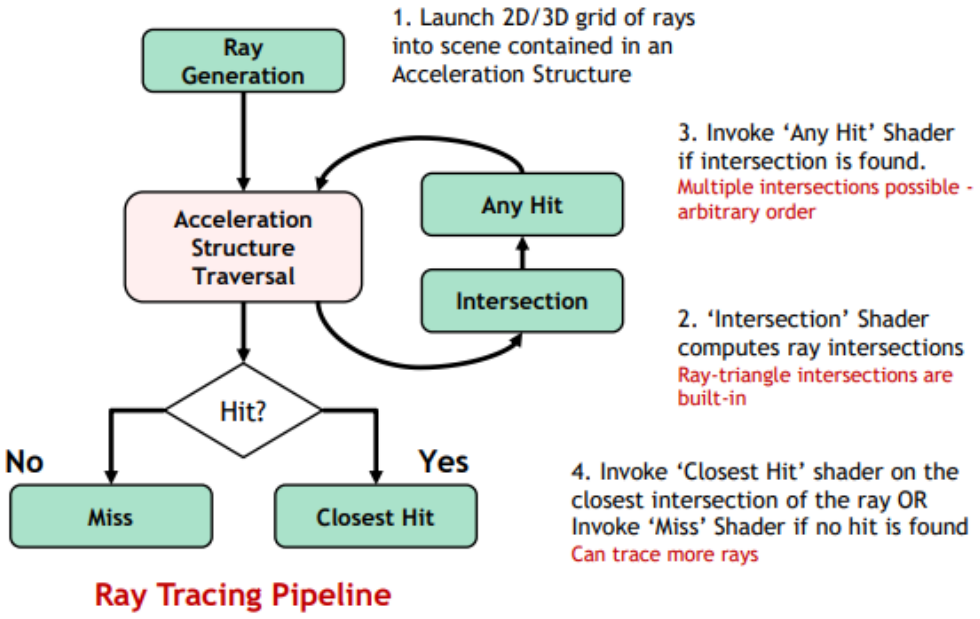
Figure 5.11.: *A visual representation of the miss shader.*

v. Shader structure

The individual shader modules are linked through the shader binding table, which enables the invocation of shader by the ray state. In this illustration, it can be seen how the individual shaders are connected and how it resembles the ray tracing pipeline.



(a) *Ray tracing shader diagrams.*



(b) *The flow diagram of a ray tracing pipeline*

Figure 5.12.: *The shader structure is build around the ray tracing pipeline call flow. The intersection and any hit shader invocations are not implemented.*

6. Evaluation

This chapter focuses on the comparison of the visual results presented by the modification. The rendered images are compared in different scenarios to evaluate the visual quality. In addition, some scenarios are used to make a statement about the playability and the visual identity. Finally, the whole application will be checked for the functional and non-functional requirements in order to evaluate the prototype.

I. Visual quality

For the evaluation of the visual results that the current modification prototype provides, the results will be compared with the 'Quake II RTX' project, since it is the most advanced path tracing project and a good point of reference.

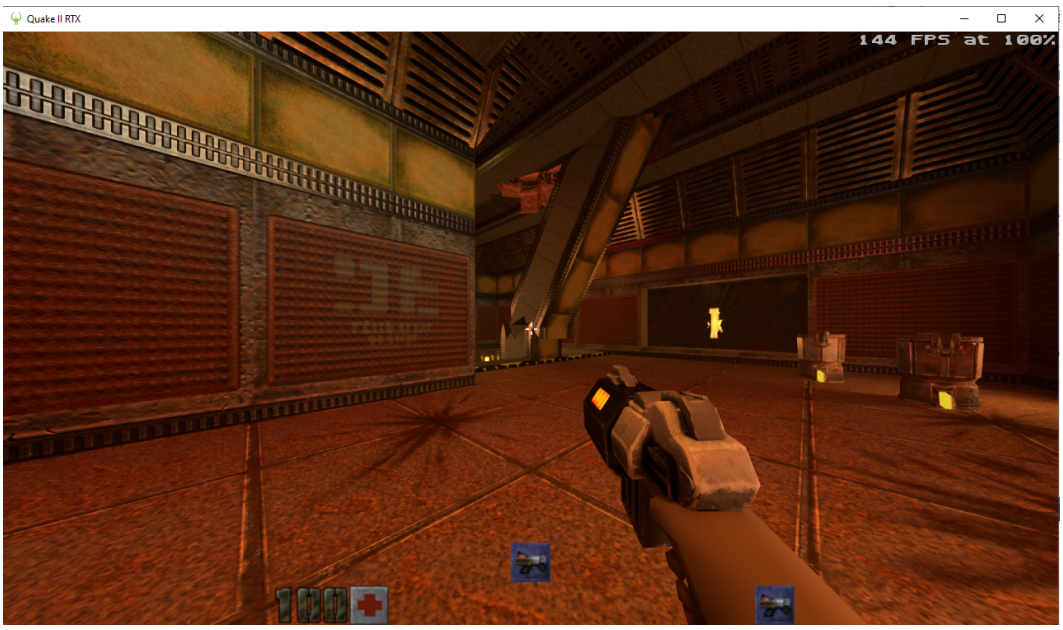
Some quality settings are made to compare the games in the fairest possible circumstance.

i. High sampling rate

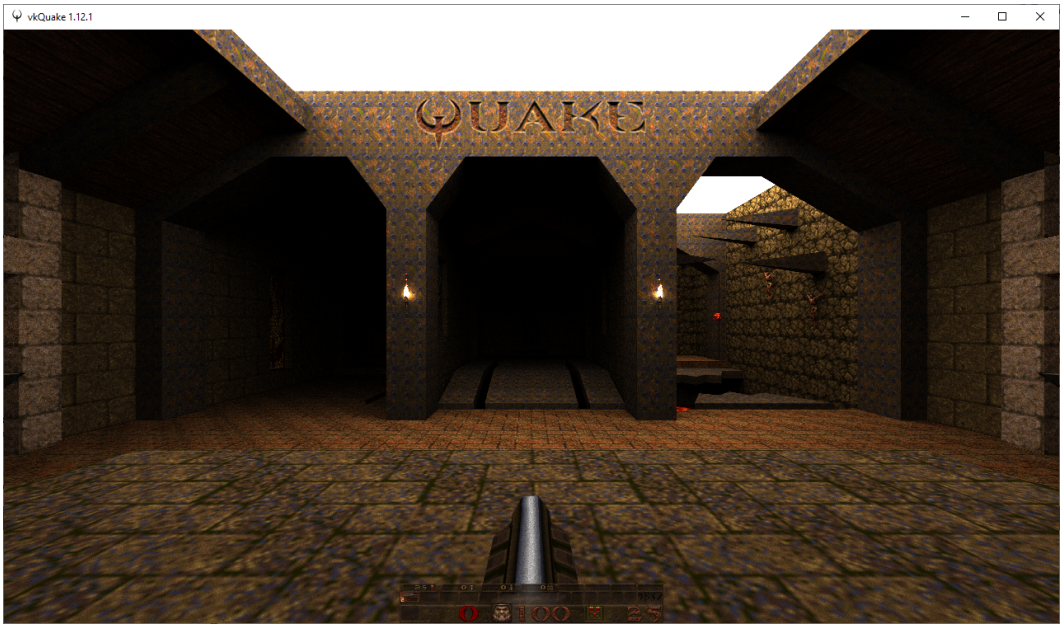
Since the modification does not implement a denoising solution, a sample rate of 512 samples is used for a similar, noise-free rendering. It should showcase the modification at its best possible rendering. Higher sample counts could also be possible, but sample counts higher than 512 provided similar results that were not worth the additional performance hit.

Quake II RTX still uses one sample per pixel, but enables denoising to remove any noise from the raw input data.

Scenario 1:



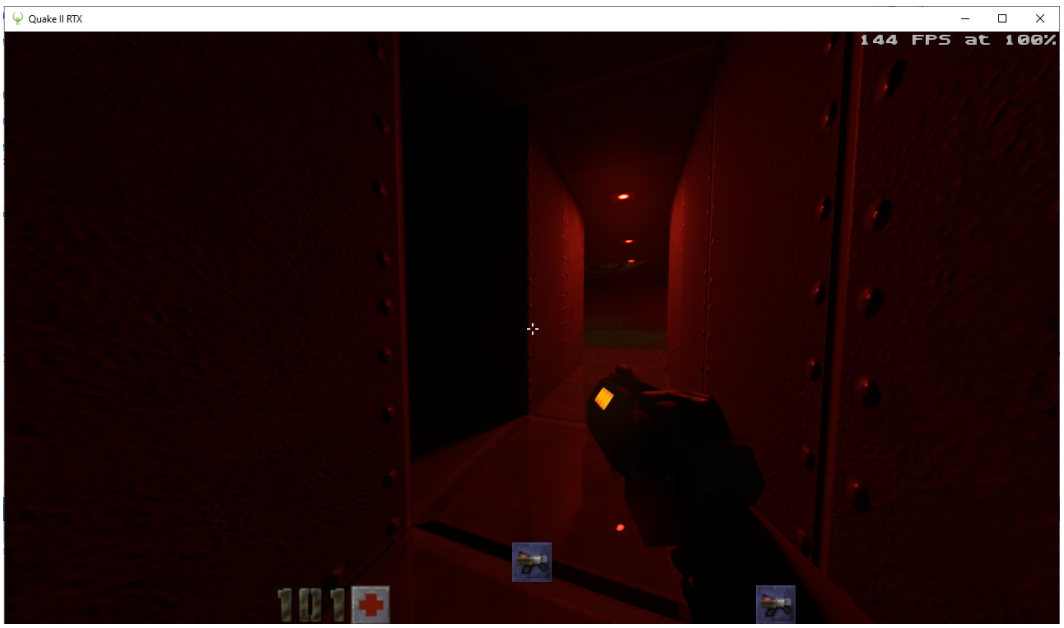
(a) *'Quake II RTX' start scene with enabled denoiser.*



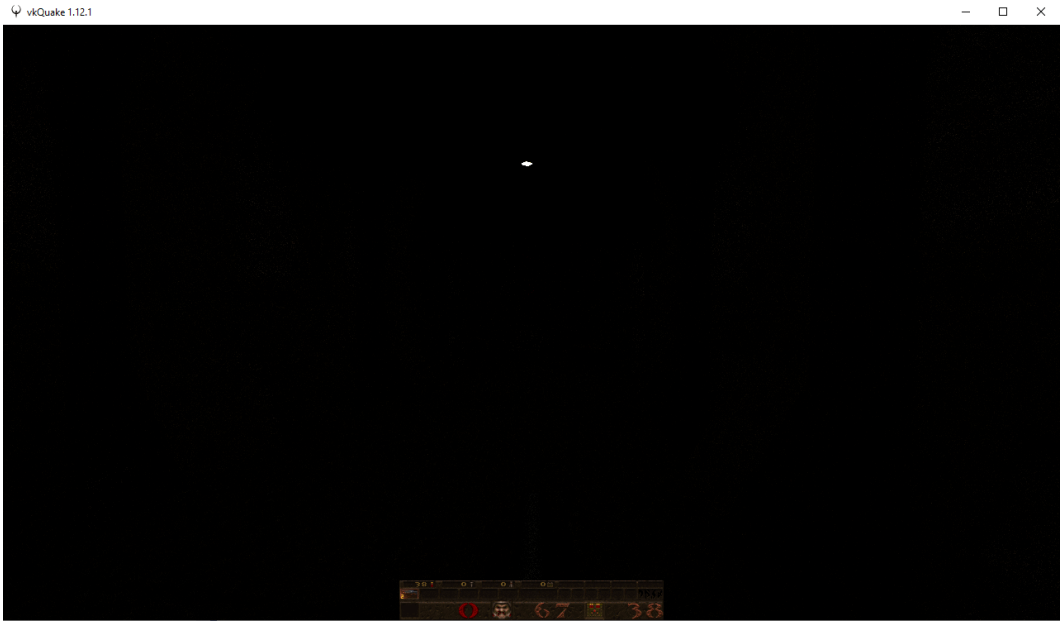
(b) *Modified 'vkQuake' start scene with 512 sampling with iteration depth of 3.*

The images illustrate the first level and the first visible scene. The modified Quake shows smooth light transitions, accurate shadows and soft-shadows that can be seen on the spike-like structures on the right. It can be seen that Quake II RTX illustrates different material properties, such as metallic surfaces, while the modified 'Quake' renders every material as diffuse, as the original game did not support different material rendering.

Scenario 2:



(a) *'Quake II RTX' low-light scenario with enabled denoiser.*



(b) *Modified 'vkQuake' low-light scenario with 512 sampling with iteration depth of 3.*

In this low light scenario, both scenes are lit up only by small ceiling lamps. The image in Quake illustrates the ceiling light, but the rest of the scene is non-visible. This illustrates a failure of the naïve path-tracing renderer in indoor, low-light scenes. An additional sampling is needed to receive enough light information to light up the scene. A random uniform sampling is not sufficient for small light sources. Quake II RTX illustrates what 'next event estimation' can do in a low-light scenario. The three ceiling lights are sufficient to light up the surrounding area. Additionally, the red color brings additional atmosphere to the scene. The water surface beneath the player reflects one

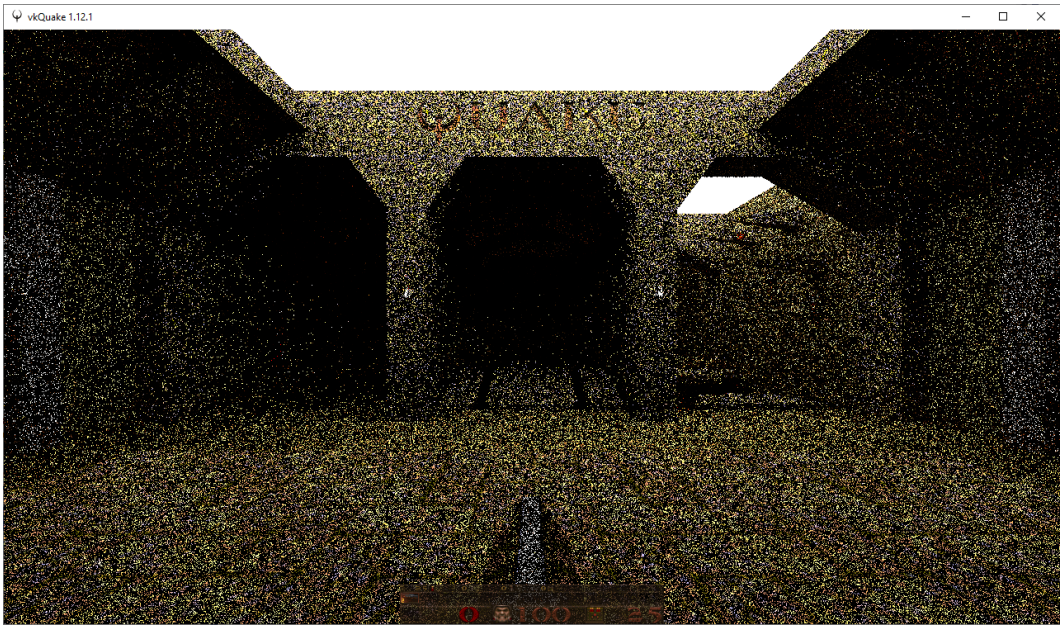
ceiling light, while the others are obscured by the player's weapon model.

ii. Low sampling rate

In these examples, the same scenes are used as for the high sample count examples. However, for this comparison, the denoiser in Quake II RTX was turned off, to show the raw input data and the sample rate of the modified Quake was set to one sample per pixel. This comparison should illustrate the current sampling solutions of both projects.

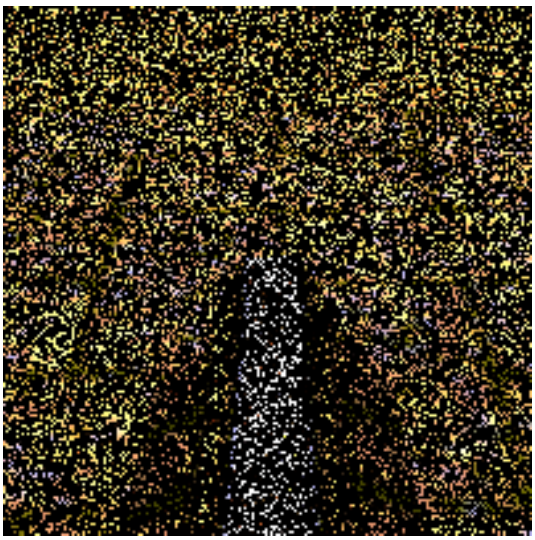
Scenario 1:

(a) *'Quake II RTX' start scene with disabled denoiser.*

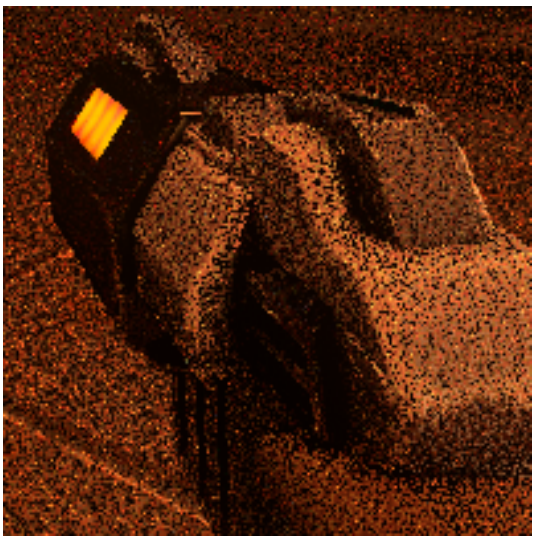


(b) *Modified 'vkQuake' start scene with 1 sample and iteration depth of 3.*

In this example, the modified Quake can generate a relatively recognizable image at a low sample rate. In comparison to Quake II RTX, the pixel grid illustrates more bright pixels. The modified Quake still shows many patches of black color.

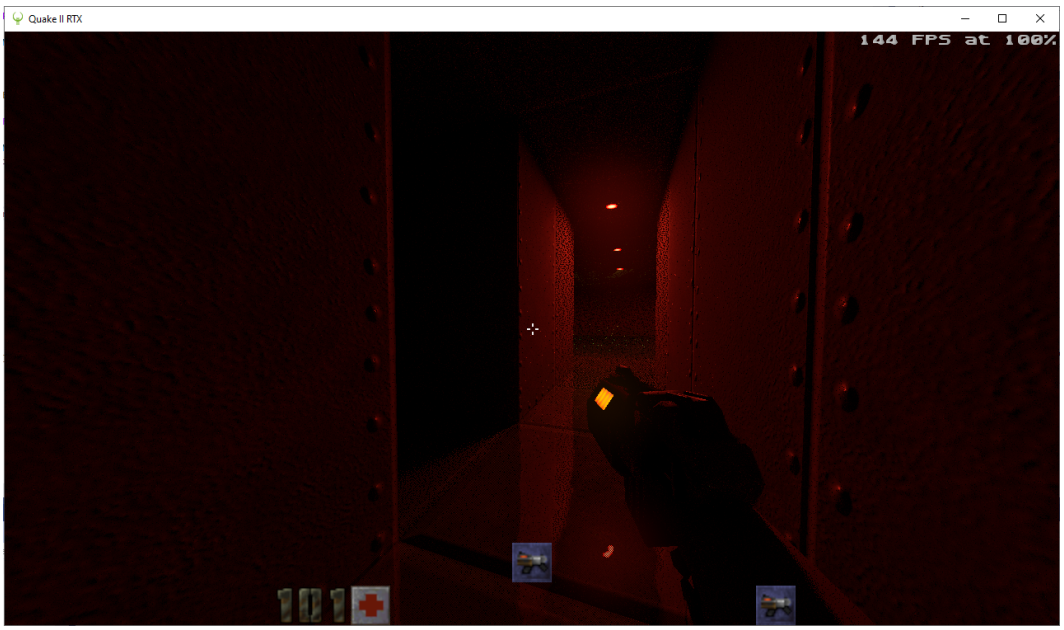


(a) *Close up of modified 'vkQuake' start scene with 1 sample and iteration depth of 3.*

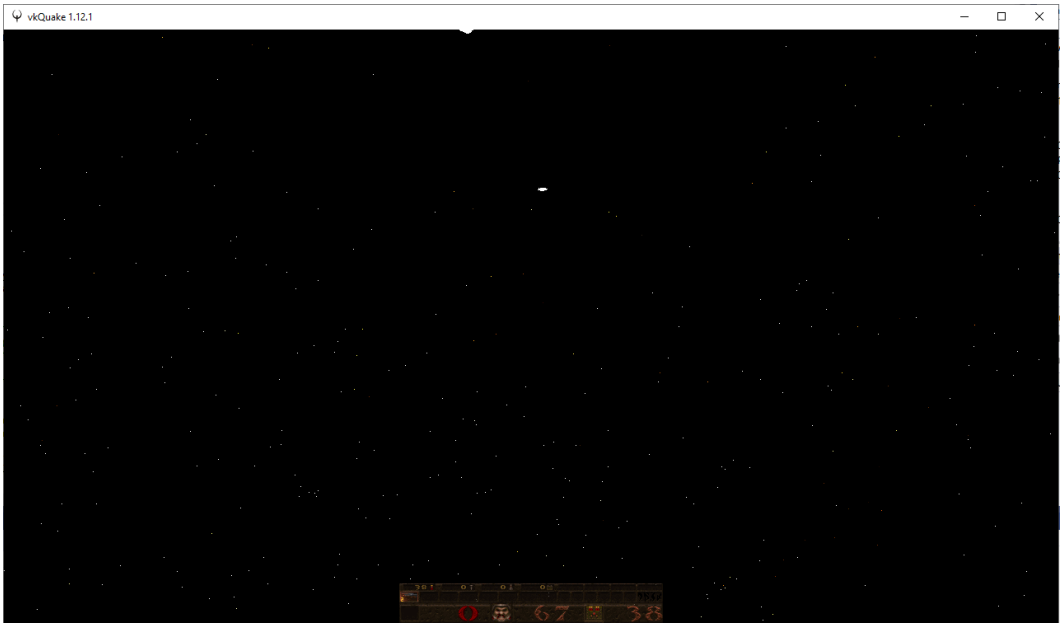


(b) *Close up of 'Quake II RTX' start scene with disabled denoiser.*

Scenario 2:



(a) *'Quake II RTX' low-light scenario with disabled denoiser.*



(b) *Modified 'vkQuake' low-light scenario with 1 sample and iteration depth of 3.*

As already mentioned in the high sample example, this scene demonstrates the limits and failure of a naive path tracing implementation. The unmodified Quake illustrates clearly that the current implementation samples nearby light randomly, as all hit points are distributed rather equally across the visible image.

Quake II RTX is able to generate an image almost indistinguishable from the denoised solution.

II. Application quality

In this section, the quality of the application is evaluated using the functional and non-functional requirements set in a previous chapter. For this, all requirements are listed and a small description is given.

i. Functional requirements

Preservation of existing functions outside the game: At this stage, all menus and navigation elements are accessible. It is also possible to change most settings during gameplay and new settings have been added. Some settings are not applicable, like anisotropic filtering, due to the new rendering technique. Other settings, like texture style, may lead to unexpected results when changing mid game.

Preservation of existing functions within the game: The movement and interaction of the game character within the game works without any issues. Buttons can be pressed, items can be picked up and used, doors can be open to progress gameplay.

Preservation of existing game and business logic: Triggers in the level work, as expected. Levels can change, button presses execute scripts and weapon mechanics work as intended. In some cases, respawning causes the game to fail, which brings the player to the main menu.

Implementation of required methods and data structures for hardware-accelerated ray tracing: This project successfully implements a ray tracing pipeline, shader binding table, utilizes acceleration structures and successfully dispatches rays that invocate all shader modules present in the shader binding table.

Successful launch of the path tracing pipeline: This test was done on two computer systems. The system with a compatible graphics card was able to start and play the application. The other system, without a compatible graphics card, was successfully notified about not meeting the hardware requirements. Unfortunately, more computer systems with compatible graphics cards could not be tested.

Implementation of a denoising solution: The game does not support a denoising solution at this point.

Support for different materials: The game does not support the representation of different materials as the original models only provide diffuse textures.

Presentation of all visual elements: The current path tracer can render and display models, textures, and light. The built-in particles system and other effects, such as explosions that utilize 2D sprites, are not supported. User interface elements are rendered and visible to the user.

ii. Non-functional analysis

Performance: The performance depends on the sample rate and the ray iteration depth of the path tracer and the memory allocation strategy of the used resources. For the performance analysis, three settings have been tested to evaluate the current average frames per second. At a resolution of 720p and a sample rate of 2, the game achieved frame rates of up to 144 FPS. With 16 samples, the average FPS were 70 to 80. With 64 samples the average frames per second were 20.

Visibility: Due to the poor sampling in indoor scenes, the brightness becomes extremely low. This makes the game unplayable in the current state.

Visual Identity: In the current state, the visual identity cannot be properly evaluated, as major parts of the elements are missing. Since the game does not alter the level files, the appearance of the elements appears to be the same. The current skybox solution changes the atmosphere dramatically, but is only a temporary solution for a sky implementation in the future.

Stability: The current build features some bugs regarding respawning and texture swapping. In some instances, allocated memory is not freed, which leads to memory leaks that have to be resolved. The rate of allocated memory per minute is not critical.

7. Conclusion

I. Summary

The goal of this thesis was the creation and modification of a hardware-accelerated path tracing renderer based on the analysis of the used data structures and system architecture of an open source game project. For this purpose, a detailed analysis of the current state-of-the-art was performed, which includes the API used, as well as research of games that use hardware-accelerated ray tracing for the entire rendering process.

For this analysis, the choice was made to use the game 'Quake' and the port 'vkQuake' respectively, as well as the Vulkan graphics API for graphics rendering and ray tracing support. The choice of the game was based on the fact that the basic integration of the Vulkan API was already established by 'vkQuake', which avoided porting the game to another graphics API. As this process can be very complicated, the actual goal of the work would not be achievable in the limited time.

The thesis focused on the data structures around the areas that received the most significant changes. In this case, it was primarily the procedure in which the data was previously processed and presented. The utilized draw methods could be repurposed for dynamic models but were not suitable for rendering of static objects.

In addition, Quake used special processing methods that posed problems for the ray tracing process. Some of these methods could be circumvented in a simple way, such as the light-mapping method. The PVS method was bypassed for static objects, but dynamic objects are still affected by those systems. The level of abstraction of the implementation is also very low, meaning that working with the integrations is by no means trivial.

Once the biggest hurdles were identified, several requirements emerged. The requirements are semantically split in two areas. For one, functions that existed prior to the modification had to function afterwards. Second, all modifications to the graphics renderer brought new requirements that were not considered before.

Once the requirements were identified and the common data structures designed, the implementation was performed. A basic ray tracing graphics pipeline was integrated with an additional shader binding table. Furthermore, the shader compilation was modified to include ray tracing shaders.

The shader structure was built around the functionality of the ray tracing graphics

pipeline to take advantage of the automatic invocation of the shader modules. The prototypical implementation implemented a brute-force 'Monte-Carlo' path tracing solution at first, which was to develop into a low-sample procedure in later development stages.

In cases where the light situation is sufficient, the naïve path tracing procedure produces a subjectively good looking representation of light gradients and soft shadows. Due to the lack of physically based materials, the current solutions represent everything as a diffuse material.

In addition, the naïve approach fails completely for very small light source instances, which in this current state is not good in terms of visibility and playability.

The list of features to be implemented is far from complete, but the systems that are integrated are working as expected for the most part. In the following section, an outlook is given.

II. Outlook

This first iteration of the path tracing renderer has a very high potential and shows that a modification is feasible. Whether the game 'Quake' was the best decision for the implementation of a path tracer can be disputed, since the game does not have many dynamic elements that justify the use of a dynamic light calculation model. This prototype is to show the status of current real-time path tracing research.

Research in light simulation has already been thoroughly explored for the past 50 years. There already exist good solutions for a variety of problems. However, the application of these approaches in real time opens up new research opportunities that are to be explored yet. Especially in low-sample sampling methods and denoising filters for real-time applications, there is still a lot of research potential.

As already hinted at in the thesis, the next goals would be the implementation of multiple importance sampling and next event estimation to improve light sampling to increase the playability.

For further research, different sampling patterns can be tested in this implementation. As of right now, the ray dispatch in the pixel grid is aligned randomly. Certain dispatch patterns can improve sampling distribution, which can lead to better denoising improvements.

Research could be done to implement an experimental indirect light sample technique called 'ReSTIR GI', published by Nvidia. It promises to converge images at low sample rates better in low light scenarios, which 'Quake' has a lot of.

8. Bibliography

[1] Andrew Burnes. *https://www.nvidia.com/en-us/geforce/news/nvidia-rtx-games-engines-apps/*. May 2021. URL: https://www.nvidia.com/en-us/geforce/news/nvidia-rtx-games-engines-apps/.

[2] Andrew Burnes. June 2019. URL: https://www.nvidia.com/en-us/geforce/news/nvidia-rtx-games-engines-apps/.

[3] Brian Curless. *Eye vs. light ray tracing*. Online: https://courses.cs.washington.edu/courses/csep557/19sp/assets/lectures/ray-tracing-1pp.pdf; Page 4; last visit: 01 02 22. 2019.

[4] Arthur Appel. "Some Techniques for Shading Machine Renderings of Solids". In: *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*. AFIPS '68 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, 1968, pp. 37–45. ISBN: 9781450378970. DOI: 10.1145/1468075.1468082. URL: https://doi.org/10.1145/1468075.1468082.

[5] Turner Whitted. "An Improved Illumination Model for Shaded Display". In: *SIGGRAPH Comput. Graph.* 13.2 (Aug. 1979), p. 14. ISSN: 0097-8930. DOI: 10.1145/965103.807419. URL: https://doi.org/10.1145/965103.807419.

[6] Brian Curless. *Whitted's algorithm*. Online: https://courses.cs.washington.edu/courses/csep557/19sp/assets/lectures/ray-tracing-1pp.pdf; Page 4; last visit: 01 02 22. 2019.

[7] Whitter Turner. *Fig. 7*. In: Association for Computing Machinery Volume 23 Number 6. Online: https://dl.acm.org/doi/10.1145/358876.358882; Page 347; last visit: 02 02 22. 1979.

[8] Brian R. Johnson. *Bidirectional Reflectance Distribution Function*. 2014. URL: http://courses.washington.edu/arch481/1.Tapestry%20Reader/4.Rendering/9xPath%20Tracing/0.default.html.

[9] Bernhard Kerbl. *Rendering: Next Event Estimation*. 2022. URL: https://www.cg.tuwien.ac.at/sites/default/files/course/4411/attachments/08_next%20event%20estimation.pdf.

[10] Eddie Edwards. *DOOM Style BSP Trees*. Aug. 1999. URL: https://www.gamedev.net/reference/articles/article654.asp.

[11] Thomas Funkhauser. 2000. URL: https://www.cs.princeton.edu/courses/archive/spr00/cs598b/lectures/reps/sld019.htm.

[12] Joshua Napoli and Douglas De Couto. *BSP Trees for Ray Tracing:* Mar. 1998. URL: http://groups.csail.mit.edu/graphics/classes/6.838/S98/meetings/m13/bsp.html.

[13] Neil Trevett. Apr. 2021. URL: https://www.khronos.org/assets/uploads/developers/presentations/Vulkan_Ray_Tracing_Overview_Apr21.pdf.

[14] John Carmack. 2012. URL: https://github.com/id-Software/Quake.

[15] Axel Gneitling. *vkQuake*. July 2016. URL: https://github.com/Novum/vkQuake.

[16] Unkle Mike. *Xash3d FWGS*. 2019. URL: https://github.com/FWGS/xash3d-fwgs.

[17] Feb. 2014. URL: https://github.com/OpenMW/openmw.

[18] Andrew Burnes. *Cyberpunk 2077: Ray-Traced Effects Revealed, DLSS 2.0 Supported, Playable On GeForce NOW — nvidia.com*. https://www.nvidia.com/en-us/geforce/news/cyberpunk-2077-ray-tracing-dlss-geforce-now-screenshots-trailer/. [Accessed 04-Mar-2022]. Andrew Burnes.

[19] Dennis Gustafsson. Oct. 2020. URL: https://www.teardowngame.com/.

[20] Christoph Schied. *Quake II RTX*. June 2019. URL: https://github.com/NVIDIA/Q2RTX.

[21] Sultim Tsyrendashiev. Aug. 2021. URL: https://github.com/sultim-t/Serious-Engine-RT.

[22] *GitHub - jmarshall23/QuakeRTX: Raytracing port of Quake 1 — github.com*. https://github.com/jmarshall23/QuakeRTX. [Accessed 04-Mar-2022].

[23] *GitHub - megayuchi/Quake-III-Arena-Kenny-Edition-With-DXR: Quake III Arena with DRX Ray Tracing — github.com*. https://github.com/megayuchi/Quake-III-Arena-Kenny-Edition-With-DXR. [Accessed 04-Mar-2022].

[24] Christoph Schied. 2019. URL: http://brechpunkt.de/q2vkpt/.

[25] Christoph Schied, Christoph Peters, and Carsten Dachsbacher. "Gradient Estimation for Real-Time Adaptive Temporal Filtering". In: *Proc. ACM Comput. Graph. Interact. Tech.* 1.2 (Aug. 2018). DOI: 10.1145/3233301. URL: https://doi.org/10.1145/3233301.

[26] Andrew Burnes. *Ray Tracing, Your Questions Answered: Types of Ray Tracing, Performance On GeForce GPUs, and More*. Apr. 2019. URL: https://www.nvidia.com/en-us/geforce/news/geforce-gtx-dxr-ray-tracing-available-now/.

[27] 2022. URL: https://www.amd.com/en/technologies/rdna-2.

[28] Michael Larabel. *Mesa 21.3 RADV Vulkan Driver Lands Ray-Tracing Support For Older AMD Radeon GPUs*. Oct. 2021. URL: https://www.phoronix.com/scan.php?page=news_item&px=RADV-Mesa-21.3-Older-RT.

[29] NVIDIA, 2018. URL: https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[30] Michael Abrash. *Inside Quake: Visible-Surface Determination*. 2000. URL: https://www.bluesnews.com/abrash/chap64.shtml.

[31] *Nvidia real-time Denoisers (NRD)*. Mar. 2022. URL: https://developer.nvidia.com/rtx/ray-tracing/rt-denoisers.

[32] 2022. URL: https://www.khronos.org/registry/vulkan/specs/1.3-extensions/man/html/VK_KHR_ray_tracing_pipeline.html.

[33] 2022. URL: https://www.khronos.org/registry/vulkan/specs/1.3-extensions/man/html/VK_KHR_ray_query.html.

[34] Kyle Halladay. *Kyle Halladay - Using Arrays of Textures in Vulkan Shaders*. http://kylehalladay.com/blog/tutorial/vulkan/2018/01/28/Textue-Arrays-Vulkan.html. (Accessed on 03/04/2022). Jan. 2018.

[35] 2008. URL: https://www.w3.org/TR/WCAG20/#relativeluminancedef.

A. Appendix

I. Source code and media

The following internet links link to the public GitHub source code repository, as well as two Google Drive links that contain a playable debug build of the modified application as well as a zip that contains video and photo media.

The debug build contains the first 'id1' level of the free shareware version of Quake.

The media zip contains three videos of the first level in Quake. It contains footage of different settings for sample rates. It also contains several screenshots of the game, as the bitrate of the recording software blurs the recorded footage, despite high bitrate settings.

GitHub repository.

Debug build Google Drive link.

Media Google Drive link.

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt durch meine Unterschrift, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und alle Stellen, die ich wörtlich oder annähernd wörtlich aus Veröffentlichungen entnommen habe, als solche kenntlich gemacht habe, mich auch keiner anderen als der angegebenen Literatur oder sonstiger Hilfsmittel bedient habe. Die Arbeit hat in dieser oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

4. März 2022, Bonn, [illegible]

Datum, Ort, Unterschrift