2023

Evaluating the Performance of Vulkan GLSL Compute Shaders in Real-Time Ray-Traced Audio Propagation Through 3D Virtual Environments

James Buggy
Technological University Dublin, Ireland

Evaluating the Performance of Vulkan GLSL Compute Shaders in Real-Time Ray-Traced Audio Propagation Through 3D Virtual Environments

James Buggy

D18124329

A dissertation submitted in partial fulfilment of the requirements of
Technological University Dublin for the degree of
M.Sc. in Computer Science (Advanced Software Development)

March 2023

DECLARATION

I certify that this dissertation which I now submit for examination for the award of MSc in Computing (Advanced Software Development), is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the test of my work.

This dissertation was prepared according to the regulations for postgraduate study of the Technological University Dublin and has not been submitted in whole or part for an award in any other Institute or University.

The work reported on in this dissertation conforms to the principles and requirements of the Institute's guidelines for ethics in research.

Signed: James Buggy

Date: 03/04/2023

ABSTRACT

Real time ray tracing is a growing area of interest with applications in audio processing. However, real time audio processing comes with strict performance requirements, which parallel computing is often used to overcome. As graphics processing units (GPUs) have become more powerful and programmable, general-purpose computing on graphics processing units (GPGPU) has allowed GPUs to become extremely powerful parallel processors, leading them to become more prevalent in the domain of audio processing through platforms such as CUDA. The aim of this research was to investigate the potential of GLSL compute shaders in the domain of real time audio processing. Specifically regarding real time ray tracing tasks. To do this a number of GLSL compute shaders were created, along with a C++ Vulkan application with which to execute them. These shaders facilitate the propagation of audio, using ray tracing, through a virtual environment, and implement 3D space partitioning and ray intersection prediction in order to gauge the effectiveness of these optimisations for this task. Statistically significant results show that the GLSL compute shaders successfully propagated audio through a virtual environment, returning results to the host system in real time, within 30 milliseconds. However, while this capability was shown, significantly detailed virtual environments prevented results from being returned in real time. Indicating a potential for future research and optimisation.

Keywords: Vulkan, GPGPU, Ray Tracing, Audio Processing, KD-Tree, Ray Intersection Prediction

ACKNOWLEDGEMENTS

I would like to express my thanks to my supervisor Eoin Rogers for his help and support throughout the research project.

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

1. INTRODUCTION

1.1 Background

Audio processing has been a popular field of study for some time, with early research utilising general purpose central processing units (CPUs) to execute processes in parallel. In more recent years, the use of CPUs to perform this processing has declined due to the availability of programmable graphics processing units (GPUs). While GPUs were initially designed for fixed-pipeline graphics processing, programmable GPUs allow general-purpose computing on GPUs (GPGPU). General purpose applications can be executed on GPUs using a variety of application programming interfaces (APIs) such as CUDA, Vulkan and OpenGL. (Bailey, 2020; Nickolls et al., 2008) As GPUs have become more powerful, the processing of audio in real time has become much more prevalent. However, real time audio comes with strict performance requirements. According to Jack et al. (2016) and Ye et al. (2018), in order for audio to be perceived by the listener as real time, it must be presented within 10 to 30 milliseconds of the triggering event. Chen (2003), presenting a lip synchronised video conferencing system, found that this could be extended to 50 milliseconds if audio preceded corresponding video, and 300 milliseconds if the audio was also time stretched. In order to meet this performance requirement, the use of GPUs as hardware accelerators has been increasingly employed in the area of real time audio processing.

1.2 Research Project/Problem

Many researchers have examined the use of ray tracing in spatial audio processing (Cowan et al., 2011; Beig et al., 2019), a popular area of audio processing research. Much research in this field has been carried out using CUDA as the means of GPGPU (Nickolls et al., 2008), however there is minimal research utilising GLSL compute shaders. Signalling a potential gap in the literature. As mentioned previously, performance of real time audio processing is critical, and so ray tracing optimisation techniques such as spatial acceleration structures (Vinkler et al., 2014; Wu et al., 2011; Havran and Bittner, 2007; Popov et al., 2009) and ray intersection prediction (Liu et al., 2021) could potentially be applied to real time audio processing problems.

The aim of this research is to examine the feasibility of spatial acceleration structures (Vinkler et al., 2014; Wu et al., 2011; Havran and Bittner, 2007; Popov et al., 2009) and ray intersection prediction (Liu et al., 2021) in optimising GLSL compute shaders, and in doing so, answer the following research question:

Research Question: Can spatial acceleration structures and ray intersection prediction be used to enhance real-time ray-traced audio propagation in 3D virtual environments using GLSL Vulkan compute shaders, so that results are returned to the host system within 30 milliseconds of the triggering event?

The following hypotheses will be used to answer the research question:
Null Hypothesis: If spatial acceleration structures and/or ray intersection prediction are used to optimise ray traced audio propagation within a GLSL Vulkan compute shader, then audio propagation results will not be returned to the host system within 30 milliseconds of the triggering event.

Alternate Hypothesis: If spatial acceleration structures and/or ray intersection prediction are used to optimise ray traced audio propagation within a GLSL Vulkan compute shader, then audio propagation results will be returned to the host system within 30 milliseconds of the triggering event.

1.3 Research Objectives

In order to answer the above research question, and determine if the above optimisations can be used to optimise a GLSL Vulkan compute shader so that ray traced audio propagation calculated by the shader can be considered real time, the following research objectives were undertaken.

1. To develop a Vulkan application allowing the execution of GLSL compute shaders
    a. This is done by extending an existing C++ Vulkan application, adding support for compute shader execution.[1]
2. To build four GLSL compute shaders
    a. Build a base, unoptimised audio propagation shader which propagates audio from source to listener, using ray tracing, within a 3D virtual environment
    b. Extend the base shader with 3D space partitioning acceleration structure, with the aim of reducing ray-geometry intersection tests
    c. Build two ray intersection prediction shaders:
        i. Extend the space partitioning shader with an intersection prediction acceleration structure, with the aim of improving traversal of the spatial partitioning acceleration structure
        ii. Build a shader which constructs a ray intersection prediction table to be used by the audio propagation shader

---

[1] Buggy, J. (2022, August 13) Siofra_Engine. *Github*. https://github.com/JamesBuggy/Siofra_Engine

3. To gather shader execution data through profiling each shader
    a. Shaders are executed using an Nvidia RTX 3080 Ti
    b. Execution time of the shaders is recorded (from the host system perspective)
    c. Data transfer time of shader results from the GPU to host memory is recorded (from the host system perspective)
    d. Internal shader execution data/metrics such as number of ray-geometry intersection tests, spatial partition traversal metrics, correct/incorrect prediction metrics is gathered for analysis
4. To analyse and compare shader profiling results
    a. An paired-samples sign test is used to determine if there is a statistically significant difference between the performance of each shader
5. To answer the research question using the analysis results

1.4 Research Methodologies

Primary research methodology was used to investigate the research problem, as a Vulkan compute application was created with which data was generated through the execution of GLSL compute shaders.

An element of secondary research was performed through a literature review, to gain knowledge of current research in the domain and to identify potential gaps in the literature.

Paired-samples sign tests were performed on the gathered data to determine if there is a statistically significant difference between the results of each compute shader. The results of these tests determine whether the null or alternative hypothesis should be accepted.

1.5 Scope and Limitations

The scope of the research is the investigation of time taken to propagate audio from source to listener in 3D virtual environments using GLSL compute shaders executed using the Vulkan API.

This investigation assumes that the ray tracing optimisation techniques found during literature review are feasible, and that the prior research on these techniques is valid.

Due to a limited frame and restricted device access, this investigation is limited to Vulkan 1.3 GLSL compute shaders, running on an NVidia RTX 3080 Ti in a non-VR Windows 10 desktop environment. Only spatial acceleration structures and ray intersection prediction optimisation techniques are

considered and the investigation does not cover a variety of compute shader languages, GPU devices or multiple graphics APIs due to the time and complexity involved in examining all possible options.

Delimitations are placed upon the research, in the interest of time. The required compute shader functionality is added to an already existing C++ Vulkan renderer.[1] The investigation is only concerned with the propagation of audio from one location to another within a 3D virtual environment, no acoustic alterations are made to the audio based on the environment.

1.6 Document Outline

Chapter 2 contains the literature review. In which existing literature in the audio processing domain is examined and discussed. A wide variety of use cases are examined. Including both GPU and non-GPU processing, and both real time and non-real time processing.

Chapter 3 contains an overview of Vulkan compute, 3D space partitioning and ray intersection prediction. Also included in this section is the design and implementation of the Vulkan application, the implementation of each GLSL compute shader, and the profiling data gathered on each shader.

Chapter 4 contains the evaluation and discussion of the results found in chapter 3. In this section the results of each compute shader are discussed and compared, the results are used to test the hypotheses and answer the research question.

Chapter 5 contains the conclusion. Which includes an overview of the research, problem definition, experiments and results. Also discussed in this section is potential areas of future work.

[1] Buggy, J. (2022, August 13) Siofra_Engine. *Github*. https://github.com/JamesBuggy/Siofra_Engine

2. LITERATURE REVIEW

This chapter details the literature review performed within the domains of audio processing and GPGPU. The literature review is broken into five sections. Section one introduces the concept of audio processing, covering a number of examples. Section two introduces the concept of GPGPU and methods of executing applications using GPUs. Section three explores the use of GPGPU in the domain of audio processing. Section four explores the use of ray tracing in audio processing, and techniques used to optimise ray tracing tasks. Section five identifies gaps in the reviewed literature, leading to the formulation of the research question.

2.1 Audio Processing

Audio processing is a broad domain with many applications, from improving the quality of existing audio, to applying effects such as amplification, equalisation, noise reduction, frequency filtering, and compression.

In audio processing, a room impulse response (RIR) is a common method of simulating the acoustics of a room for the purpose of audio correction, or the simulation of audio in a virtual environment. The RIR of a room is a description of the room's acoustic characteristics, found by measuring sound pressure from a fixed point within the room (Carini et al., 2016).

The impulse response of a room is an important detail in the research performed by Kontomichos et al. (2015), Gu et al. (2014) and Junwei and Mengyao (2013).

Kontomichos et al. (2015) present a method of real time room equalisation on audio streams by measuring the impulse response of a room, calculating equalisation filter coefficients based on the impulse response, and convolving any input audio streams with the filter coefficients to equalise the signals in real time. It is concluded that the presented room correction method is an effective, inexpensive solution for real time audio quality improvement, covering the needs of both experienced and novice users.

Gu et al. (2014) propose a ray tracing method to sample a room and simulate its impulse response. The paper proposes the distribution of rays according to equal area rather than equal angle. The researchers compare the results of their proposed ray distribution method to that of a Monte-carlo distribution. The paper concludes that the proposed method results in "better result for early reflection and later reverberation." (Gu et al., 2014, p. 832).

Junwei and Mengyao (2013), regarding the simulation of audio in virtual environments, propose a method of using ray tracing in the frequency domain to simulate a combined head and room impulse response. This is proposed as an alternative to simulating room impulse response and head-related transfer function, a description of how sound waves are altered by the human head and torso prior to reaching the eardrum, separately. It is concluded that the results are “very promising and motivate further research” (Junwei and Mengyao, 2013, p. 363)

Other areas of the audio processing domain include the encoding/decoding of audio data. Kurtisi and Wolf (2008) present a method of real time audio coding over a distributed network using WavPack, an open audio compression format, in an attempt to overcome the processing latency of music codecs and the low quality of voice codecs. WavPack is compared to Adaptive Differential Pulse-Code Modulation (ADPCM) in terms of audio quality. It was found that WavPack, even with highest compression, outperformed ADPCM.

2.2 General-Purpose Computing on GPUs (GPGPU)

With the advent of programmable GPUs, GPUs are now commonly utilised as parallel processors, rather than simply graphics processors, to accelerate physics computations, video transcoding, image and audio processing, and more (Nickolls & Dally, 2010; Luebke, D., 2009).

There are many ways in which to execute applications using GPUs, such as with CUDA (Nickolls et al., 2008), using kernels written in C++ or C, or the Vulkan API (Bailey, 2020) using shaders written in GLSL or HLSL which are compiled to Spir-V.

Uses of GPGPU can be found in the research performed by Schutz and Wimmer (2019), Junker and Palamas (2020) and Xu et al. (2022).

Schutz and Wimmer (2019) explore two methods of point cloud rendering using compute shaders rather than classic OpenGL rendering pipelines. Using their compute based rasterizer, they achieved up to 10 times greater performance in point cloud rendering using vs the traditional OpenGL pipeline. However they also concluded that the traditional OpenGL pipeline scales better when point sizes are larger than 2 pixels.

Junker and Palamas (2020) present a real time snow simulation using compute shaders which simulates snow deformation, accumulation and the effect of wind. They present a solution which achieves up to thirty frames per second with up to one hundred agents active in the virtual environment.

A method for building high-performance, modular graphics shaders is presented by He et al. (2017). Crawford et al. (2018) investigate how compiler optimizations affect graphics shaders. Multiple vendors' desktop and mobile GPUs are used to evaluate the effect.

2.3 Audio Processing Using GPGPU

Given the rise of GPUs as parallel processors, GPGPU has become a popular parallel computing solution in the domain of audio processing. This section provides examples of research performed in the audio processing domain which utilises GPGPU.

Pascuzzi and Goli (2022), Qi et al. (2011), and Chen and Li (2013) explore the uses of GPGPU in fast fourier transform (FFT) calculation. FFT is a powerful tool, used for many tasks in audio processing, such as frequency analysis of an audio signal, frequency filtering, pitch detection, and much more. As FFT is computationally expensive, and the processing of audio signals is generally parallelizable (Nikolov et al., 2015), GPGPU is a promising solution for this computation. Pascuzzi and Goli (2022) present a SYCL based FFT library, and benchmark it against cuFFT and rocFFT on Nvidia and AMD GPUs respectively, finding significant performance improvements if launch overheads of their SYCL approach are disregarded. Chen and Li (2013) present an FFT library which uses a hybrid CPU-GPU approach for large FFT problems in an attempt to overcome the data transfer and memory restrictions of GPUs. They found that their solution outperforms many currently available large scale FFT implementations. Qi et al. (2011) used OpenCL as a solution to calculating FFTs on a GPU. Their approach uses mixed precision by varying single precision vs double precision floating point numbers throughout the FFT as a method of achieving greater performance. They report a lower error rate compared to cuFFT, and a significant increase in performance compared to Intel MKL.

Using CUDA as a GPGPU platform, Ouali et al. (2015) present a parallel audio fingerprint similarity search algorithm. Using the short time Fourier transform to generate a spectrogram matrix, from which a 2D binary image is derived. In this image, a 1 signifies a time-frequency peak. The image is then partitioned into a 10 by 10 grid, and an image compression technique, quantisation, is performed by summing each value within each grid square. This gives the audio fingerprint which can then be used to identify similar audio signals.

2.4 Ray Tracing & Optimisations

This section discusses research into the use of ray tracing and ray tracing optimisations.

Profiling and debugging is an important step in the creation of ray tracing applications in order to ensure accurate and performant solutions. Pankratz et al. (2021) present a solution allowing performance profiling of ray traced Vulkan graphics applications. Gribble et al. (2012) present a toolkit that allows the recording and visualisation of rays traced through an environment. Allowing users to view ray state at any time during the ray tracing process.

Spatial acceleration structures (Alfrink et al., 2021) are perhaps the most common ray tracing optimization technique. These data structures represent a virtual object or scene which has been subdivided or partitioned into smaller areas. The aim of this is to reduce the number of intersection tests required between a scene or object, and a ray in order to find an intersection. Examples of spatial acceleration structures include volumetric trees (Hossain et al., 2015), and grids (Lagae and Dutré, 2008). Two of the most common spatial acceleration structures used in ray tracing optimisation are bounding volume hierarchies (BVHs), and KD-trees.

A BVH is a tree structure in which each leaf node describes an object in a scene, interior nodes describe axis aligned bounding boxes (AABBs) which encapsulate sets of these objects, and the root node describes an AABB which encapsulates the entire scene. When a ray is cast into the scene, the BVH is traversed by performing intersection tests between the ray and AABBs within the tree, starting at the root, until a leaf node is found. If a leaf node is found, intersection tests are performed between the ray and leaf node object. This eliminates interaction tests between the ray and many objects with which it will not intersect. Given the popularity of BVHs, improvements in their construction and traversal performance are common areas of research. Popov et al. (2009), Viitanen et al. (2016), and Benthin et al. (2017) investigate methods of improving the construction of BVHs. Popov et al. (2009) provide an overview of several BVH construction algorithms, and present their own algorithm which can identify “optimal partitions in polynomial time” (Popov et al., 2009, p. 21). Viitanen et al. (2016) evaluate the performance of BVHs in which each node may have up to four children, in contrast to a traditional BVH which is binary in nature. Using their proposed BVH structure, they found both reductions in energy requirements and an increase in performance when compared to traditional BVH structures. Benthin et al. (2017) propose an alteration to the two-level BVH structure, a BVH in which each leaf node object also has its own BVH. Their proposal allows the contents of leaf object BVHs to be merged into the top level BVH in the event of object BVH overlap. Using their proposed construction algorithm, they report significantly faster build times compared to traditional BVHs, and a faster traversal time compared to two-level BVHs not constructed with their algorithm. Laine (2010), Benthin et al. (2019), and Hapala et al. (2013) explore improvements to the traversal of BVHs. Laine (2010) evaluates a stackless traversal method in which the traversal progress is encoded within the tree nodes during the traversal process. They do this by adding an extra bit of data to each node signalling if the node or its subtree has been traversed. Using

their solution they report a 1.3 to 2.5 times performance increase over stack based traversal. Benthin et al. (2019) expand upon the traversal method presented by Laine (2010). They do this by using a short stack alongside the restart trail presented by Laine (2010). They also propose the addition of an array of counters, one for each layer of the BVH. These counters record how many nodes at each level of the tree have been processed, allowing traversal to skip already processed sub trees in the event of a traversal restart. They report that their additions resulted in a 10 percent restart overhead when compared to the proposal of Laine (2010). Hapala et al. (2013) present an iterative approach to BVH traversal, rather than stack based. Their approach uses a state machine with knowledge of from where the traversal entered the current node, from a child, or from a parent. This allows traversal to continue correctly without the need of a stack to recall nodes to be processed.

Similar to a BVH, a KD-tree is a hierarchical structure in which the root node describes an AABB which encapsulates an entire scene. In the case of a KD-tree, this root AABB is recursively split until a cease condition is met. This may be a maximum tree depth, or a minimum number of data elements in a node. As a popular spatial acceleration structure, much research has been done regarding the improvement of KD-trees. Wu et al. (2011) present a parallel KD-tree construction algorithm that utilises the surface area heuristic to determine the best split plane. They report that their algorithm constructs KD-trees of equal quality to those constructed using CPU based algorithms, though does so with much higher performance. Li et al. (2014) also present a parallel KD-tree construction algorithm, however, they elected to use Morton code to identify split planes. They report significantly lower KD-tree construction times compared to state of the art processes. Choi et al. (2013) propose a KD-tree construction method in which triangles duplicated across multiple leaf nodes are removed from those leaf nodes, and placed into inner nodes of the KD-tree. They present a heuristic which decides when this should be done, and they use the three least significant bits of each node to specify whether the node contains triangles. They report a significant reduction in memory requirements through the use of this construction method. Havran and Bittner (2007) propose a KD-tree construction algorithm in which AABBs are not constructed for every non-leaf node. AABBs are distributed sparsely throughout the tree in order to reduce the memory requirements of the tree. They also present several traversal algorithms which make use of the new sparse AABB tree. They report that the “current version of the algorithm brings speedup only for scenes with very high depth complexity and short rays” (Havran and Bittner., 2007, p. 52).

Given the popularity of BVHs and KD-trees, much research comparing the two structures has been performed. Vinkler et al. (2014) compare the performance of BVHs and KD-trees as ray tracing optimisations on GPUs. This performance comparison focuses on traversal performance rather than tree construction performance. The trees are constructed using the SAH for node subdivision. They report that BVHs have higher performance on low to moderately detailed scenes, while KD-trees had

higher performance on highly detailed scenes. Wu et al. (2011) note that BVHs are often preferred for optimising dynamic scenes as they are generally faster to construct, while Havran and Bittner (2007), and Popov et al. (2009) note that KD trees are preferred for use in static scenes and small ray counts.

MacDonald and Booth (1990) explore two methods of determining optimal space subdivision during the construction of spatial acceleration structures.

Hierarchical spatial acceleration structures, such as those mentioned above, are not the only means by which to optimise ray tracing tasks. Meister et al. (2020) compare current methods of ray reordering, and propose a modification to an existing method of computing ray sorting keys, using a parallel radix sort. They report a significant performance increase using their proposed method. Nabata et al. (2013) propose a divide and conquer ray tracing method which uses ray sampling rather than purely focusing on the distribution of primitives as was done with previous divide and conquer methods. Xu et al. (2022) propose a method of computing ray-traced soft-shadows for dynamic objects in real time. Concial ray culling is used to avoid casting rays which will not contribute to the shadow. Using a bounding sphere around a dynamic object, a cone is generated and used to cull unneeded rays. With this approach, they found that fewer and shorter rays are required, significantly improving the performance of rendering soft shadows. However, as the number of dynamic objects increases, the performance gain decreases. Ray tracing performance can also be improved by skipping redundant ray casts or spatial acceleration structure traversal, according to a technique proposed by Liu et al. (2021). They found that ray hashes could be used to identify previous, similar rays, and reuse the results. They report that this decreased memory accesses and total execution time by 13 percent and, on average, 26 percent, respectively.

2.5 Gaps In The Literature

The literature review performed as part of this research found much research utilising platforms and frameworks such as CUDA and OpenCL for GPU audio processing. However, the review also highlighted a lack of research regarding the use of GLSL compute shaders for audio processing. Specifically, real time ray traced audio. However real time audio comes with strict performance requirements, according to Jack et al. (2016) and Ye et al. (2018), in order for audio to be perceived by the listener as real time, it must be presented within 10 to 30 milliseconds of the triggering event. Chen (2003), presenting a lip synchronised video conferencing system, found that this could be extended to 50 milliseconds if audio preceded corresponding video, and 300 milliseconds if the audio was also time stretched. Many of the ray tracing optimisation techniques identified during the review may be applied to GLSL shaders, potentially addressing these performance requirements, leading to the research question:

Research Question: Can spatial acceleration structures and ray intersection prediction be used to enhance real-time ray-traced audio propagation in 3D virtual environments using GLSL Vulkan compute shaders, so that results are returned to the host system within 30 milliseconds of the triggering event?

3. DESIGN AND IMPLEMENTATION

This chapter describes the design and implementation of the GLSL compute shaders, and of the Vulkan C++ application with which they are executed. Also described in this section are the virtual test environment in which the GLSL shaders will propagate audio, and the experiments performed on each GLSL shader, along with the data gathered during these experiments as results to be analysed.

3.1 Vulkan Compute Implementation

This section provides an overview of the Vulkan components required for computer shader execution, and the details of their implementation using the Siofra renderer.[1]

Bailey (2020) provides a more comprehensive overview of the Vulkan API.

Vulkan Instance

A Vulkan instance is a handle that allows interaction with the Vulkan API. It is responsible for the initialisation of the Vulkan library, specifying the Vulkan API version, and required/desired extensions. It is also responsible for enabling debug callback functionality and validation layers. It is the first object that must be created when interacting with the Vulkan API, as it is used to create logical devices, which represent the physical devices on which an application will run. Which in turn are used to create almost all other Vulkan objects.

See appendix 6.1.1 for C++ code snippet.

Logical Device

The logical device is a representation of the physical device on which an application will be executed. The logical device is created using the Vulkan instance handle, is used to create almost all other Vulkan objects, and is responsible for synchronisation between Vulkan objects through the use of fences and semaphores. Validation layers, if desired, must also be specified on the logical device. The logical device is created using a physical device handle, using the Vulkan instance to query for available GPUs, and selecting one based on desired queue, memory and extension support.

See appendix 6.1.2 for C++ code snippets.

---

[1] Buggy, J. (2022, August 13) Siofra_Engine. *Github*. https://github.com/JamesBuggy/Siofra_Engine

Command Buffers and Queues

A command buffer is an object containing a series of commands to be executed. These commands can include graphics, compute and transfer operations. Operations are pre-recorded into a command buffer which is then submitted to a queue for execution. Command buffers are created using a logical device and are allocated from a pre-allocated block of memory, a command pool.

See appendix 6.1.3 for command buffer creation C++ code snippets.
See appendix 6.1.9 for compute pipeline execution command recording C++ code snippet.

Storage Buffers

Vulkan buffer objects are handles to blocks of GPU memory in which data is stored. Buffers are created using the logical device, and their intended use must be specified at this time. The intended use of a buffer is specified using VkBufferUsageFlagBits, and includes uses such as a general data storage buffer, a uniform buffer or a data transfer buffer. Buffers can be allocated in two types of GPU memory, device local or host visible. Device local memory is accessible only to the GPU, while host visible memory can be accessed also by the CPU. Host visible memory is often used as a hand-off area for data transfer between host system memory and GPU memory.

See appendix 6.1.4 for C++ code snippets.

Descriptor Sets

Descriptor sets are objects that contain the resources, such as buffers and images, that are required by a specific shader. A resource is bound to a descriptor set using a descriptor binding, which contains the information needed by the shader to access the resource. Such as the size, format and layout of the data. Descriptor bindings are specified by creating a descriptor set layout, which is used in the creation of the descriptor set. Descriptor sets, and therefore resources, to be used by a shader are specified by recording a vkCmdBindDescriptorSets command to a command buffer as part of the shader execution command recording. Descriptor sets are created using the logical device, and are allocated from pre-allocated blocks of memory, called descriptor set pools.

See appendix 6.1.5 for C++ code snippets.
See appendix 6.1.9 for vkCmdBindDescriptorSets command recording C++ code snippet.

Push Constants

Push constants are an additional method of passing small pieces of data to a shader, without requiring storage buffers or descriptor sets. Push constants are specified during Vulkan pipeline creation, and

data is passed to the shader by recording a vkCmdPushConstants command to a command buffer as part of the shader execution command recording.

See appendix 6.1.7 for push constant specification during pipeline creation C++ code snippet.
See appendix 6.1.9 for vkCmdPushConstants command recording C++ code snippet.

Shader Modules

A shader module is a handle to the compiled code for an application to be executed on the GPU. Shader modules are created using the logical device, and in the case of this implementation, GLSL code is compiled to SPIR-V format, the bytes of which are then read from the .spv file, and are used to create the shader module. Shader modules are specified during pipeline creation, at which time the pipeline stage to which the shader applies is specified. Pipeline stages include, vertex, fragment, geometry, compute stages.

See appendix 6.1.6 for shader module creation C++ code snippets.
See appendix 6.1.7 for shader module specification during pipeline creation C++ code snippet.

Compute Pipeline

The compute pipeline is the culmination of each of the Vulkan objects introduced so far. Vulkan pipelines can be complex to create. However, for this implementation, only a compute stage is required, and so only the descriptor set layouts, push contents and shader modules must be specified. The compute pipeline is executed by binding to a command buffer by recording a vkCmdBindPipeline command to the command buffer, and is executed by recording a vkCmdDispatch command to the command buffer.

See appendix 6.1.7 for pipeline creation C++ code snippets.
See appendix 6.1.9 for pipeline execution C++ code snippet.

Fences

A fence is a handle to a synchronisation object that allows CPU - GPU synchronisation. Fences can be specified during command buffer submission to a queue, and can be used by the CPU to track the progress of the command buffer. The status of a fence can be checked in a non-blocking fashion using vkGetFenceStatus, alternatively, the application can block until the fence is unsignaled using vkWaitForFences.

See appendix 6.1.8 for fence creation and queue submission C++ code snippets.
See appendix 6.1.9 for fence usage during pipeline execution C++ code snippet.

Shader Execution

The compute shader is then executed by:

1. Beginning command recording on a command buffer
2. Bind the created pipeline to the command buffer
3. Record the push constants command to pass push content data to the shader
4. Record the bind descriptor sets command to allow the shader to access the required resources
5. Record the vkCmdDispatch command which executes the shader
6. End command buffer recording and submit the command buffer to the logical devices compute queue for execution, along with the fence allowing the CPU to track the command buffer progress

See appendix 6.1.9 for compute pipeline execution C++ code snippet.

3.2 Test Application Overview

This section provides an overview of the application and virtual environment used to investigate ray traced audio propagation. All coordinates below are interpreted as metres.

Virtual Environment

The virtual environment in which experiments will be run consists of six rooms, measuring four by four metres when viewed from above, and two metres in height. One central room, surrounded by five outer rooms. The rooms are connected through a number of hallways in which audio will propagate.

Audio Listener

The audio listener is placed in the centre of the central room, at a position of **X**: 0.0, **Y**: 1.0, **Z**: 0.0.

If any audio propagation rays pass within one metre of the audio listener, that ray is considered to have reached the listener.

Audio sources

Five audio sources are placed in the five outer rooms, one audio source in each room.

The audio sources are colour coded for identification and are positioned as follows:
Red: **X**: -3.0, **Y**: 1.0, **Z**: 8.0.
Blue: **X**: -12.0, **Y**: 1.0, **Z**: -5.0.
Yellow: **X**: 0.0, **Y**: 1.0, **Z**: -6.0.

Pink: **X**: 8.0, **Y**: 1.0, **Z**: -2.0.
Green: **X**: 5.0, **Y**: 1.0, **Z**: 8.0.

When an audio source is triggered, a number of rays originating from the audio source position will be cast into the environment at equal angles. These rays will propagate through the environment, reflecting as a result of geometry intersections, until they have reached the audio listener or exceed a reflection limit, at which point they are considered to have not reached the listener.

Figure 3.1 depicts the virtual test environment, viewed from above.
Figure 3.2 depicts an example of sixteen rays propagating from the red audio source, viewed from above. Green lines indicate walls, red lines indicate the wall normals.

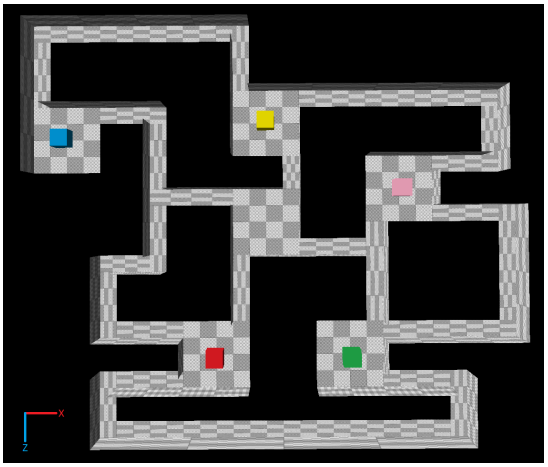

Figure 3.1 - Virtual Test Environment

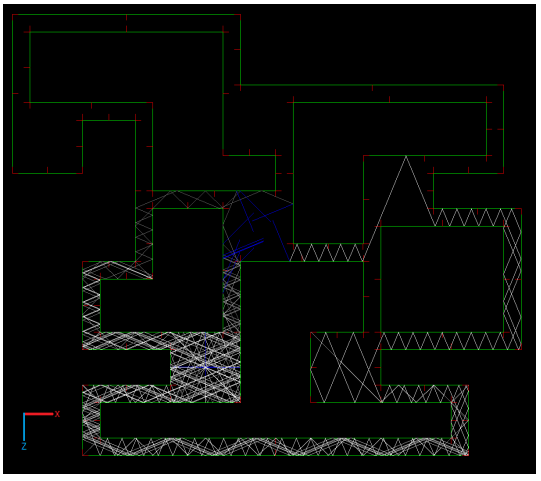

Figure 3.2 - Audio Propagation Example

3.3 Base Shader

This section details the implementation of the base audio propagation shader. The base audio propagation shader contains no ray tracing optimisation techniques. It is used to obtain a baseline to which the two optimised shaders, built upon this shader, will be compared.

Also in this section, the experiments performed on the base shader are detailed.

3.3.1 Implementation

Shader Inputs

The base audio propagation shader takes five inputs, these take the form of:

1. A vertex buffer containing the vertex position and normal data of the virtual environment geometry.
2. An index buffer containing indices into the vertex buffer. Each group of three contiguous entries in the index buffer describes the three vertices of a triangle in the virtual environment geometry. Figure 3.3 shows the relationship between vertex and index buffers storing data for a pair of triangles.
3. A geometry metadata buffer containing metadata on the vertex and index buffers, such as the number of elements in each.
4. An output buffer, into which the shader's audio propagation results are stored, to be accessed by the CPU upon completion of the shader execution.
5. A push constant containing the triggered audio source position, and the audio listener position.

Geometry vertex and index data is read from .obj files containing the data describing the floor, walls and ceiling of the virtual environment. Once loaded, this data is copied to the vertex and index buffers, and the metadata buffer is updated with the count of vertices and indices. The output buffer contents are populated by the GLSL shader during execution.

The four buffers are implemented as storage buffers, and are linked to a pair of descriptor sets which are bound while recording the compute pipeline execution operations to a command buffer. The push constant values are also provided to the shader at this time.

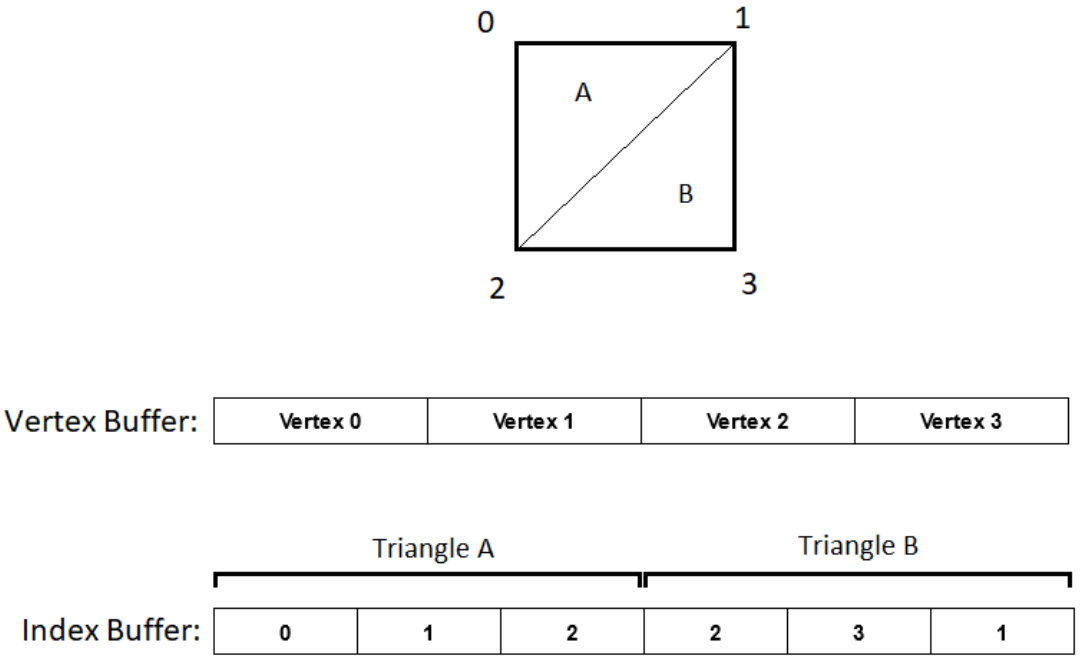

Figure 3.3 - Vertex and Index Buffer Layout

See appendix 6.1.10 for vertex/index/metadata/output buffer setup C++ code snippet.
See appendix 6.1.9 for C++ code snippet showing input descriptor set binding and push constant updates prior to pipeline execution.
See appendix 6.1.11 for the base shader GLSL code, containing the layout of the input data.

Shader GLSL Implementation

The shader generates a number of rays, each originating from the audio source position, with a direction determined by an equal angle distribution based on the number of rays. Each ray is processed by an individual thread.

Each thread manages a queue of rays to process, which is seeded with the threads initial ray originating from the audio source. While the queue contains unprocessed rays, the next ray is taken from the queue, and intersection tests are performed between the ray and the virtual environment geometry, using the Möller-Trumbore algorithm for ray/triangle intersection detection (Möller and Trumbore, 1997). A ray audio listener intersection test is also done using the quadratic formula to detect intersections between the ray and a sphere, one metre in diameter, centred at the audio listener position.

While searching for ray intersections, the closest found intersection is remembered. After completing all intersection tests, if the closest intersection was between the ray and environment geometry, the ray is reflected around the intersection normal, and the reflected ray is added to the ray queue to be processed. Alternatively, if the closest intersection is between the ray and the audio listener, processing completes for this thread and the details of the final reflected ray, its origin and direction, are placed in the output buffer.

Each thread continues processing rays in this way until the thread's ray queue is empty, a ray reaches the audio listener, or rays have been reflected a maximum number of times, defined as five hundred, without reaching the audio listener.

See appendix 6.1.11 for the base shader GLSL code.

Shader Output

The shader output buffer contains an array of output structs, one for each thread. Thread ids are used to index into the output buffer in order to write to the output struct for a given thread.

Each thread outputs the following data. A boolean indicating if any of the thread’s rays reached the audio listener and the origin and direction of the thread’s final reflected ray.

Additional debugging and metrics data is also output when not recording the shader execution time.

See appendix 6.1.11 for the base shader GLSL code, showing the layout of the output data.

3.3.2 Experiments

For the purposes of the research experiments, three levels of geometry detail are defined, each with differing vertex and index counts. These are shown in Table 3.1.

Geometry Detail Level	Vertex Count	Index Count
1	560	840
2	1260	3360
3	3500	13440

Table 3.1 - Virtual Environment Geometry Detail Levels

Five levels of ray detail are also defined, with an equal number of GPU threads:

Ray Detail Level	Rays
1	64
2	128
3	256
4	512
5	1024

Table 3.2 - Ray Detail Levels

To gather shader execution data, the shader was executed for each audio source one hundred times for every combination of geometry and ray detail level.

For each of these executions, the following data was gathered:

- Shader execution time (in microseconds).
- Time (in microseconds) taken to copy shader execution results from GPU memory to host system RAM.

Each audio source was executed an additional time to gather the following data:

- Total geometry intersection tests in each thread
- Successful geometry intersection tests in each thread
- Total intersection tests across all threads
- Successful geometry intersection tests across all threads
- Mean intersection tests across all threads
- Mean successful intersection tests across all threads

3.4 3D Space Partitioning

This section details the implementation of the 3D space partitioning approach used to optimise the base audio propagation shader, the experiments used to evaluate optimisation, and the results of those experiments. The aim of this optimisation is to partition the scene in order to reduce the number of ray-geometry intersection tests required to find the closest geometry intersection of a ray.

There are many acceleration structures which can be used to partition 3D space for ray tracing tasks, such as octrees, grids, bounding volume hierarchies (BVH) and KD trees. Wu et al. (2011) note that BVHs are often preferred for optimising dynamic scenes as they are generally faster to construct,

while Havran and Bittner (2007), and Popov et al. (2009) note that KD trees are preferred for use in static scenes and small ray packets.

With these points in mind, and given that the virtual environment used in this research contains no dynamic objects, it was decided that a KD tree is the preferred approach to optimise the base audio propagation shader.

3.4.1 Implementation

The optimisation is built into the base shader described above, and so shares many of the same details. This section describes the differences.

Shader Inputs

Li et al. (2014) discuss a method of quickly constructing KD-trees in parallel, executed on a GPU using CUDA. The research performed here however is not concerned with the performance of tree construction, so here the KD-tree is constructed once using the CPU and is then transferred to GPU memory, and linked to the same descriptor set as the geometry vertex and index buffers. As the scene is static, the KD-tree will not need to be rebuilt.

The KD-tree is represented using a pair of buffers:

- The KD-node buffer, a flat array containing the nodes of the tree
- The leaf node geometry buffer, which is used by leaf nodes to identify the geometry overlapping the nodes axis aligned bounding box (AABB). This buffer contains indices into the geometry index buffer. In order to reduce memory usage, the leaf node geometry buffer only contains the first index of a triangle in the index buffer, as any three indices describing a triangle are contiguous in the index buffer.

Each KD-node contains:

- The details of an AABB, position and extents
- Indices for the node's near and far child, which index back into the KD-node buffer
- In the case of leaf nodes, start and end indices into the leaf node geometry buffer, describing a range of elements relevant to the leaf node and thus the geometry overlapping the leaf node AABB. Figure 3.4 shows the relationship between the leaf node geometry buffer, vertex buffer and index buffer, for a leaf node, containing two triangles, with a leaf node geometry start index of 0 and an end index of 1.

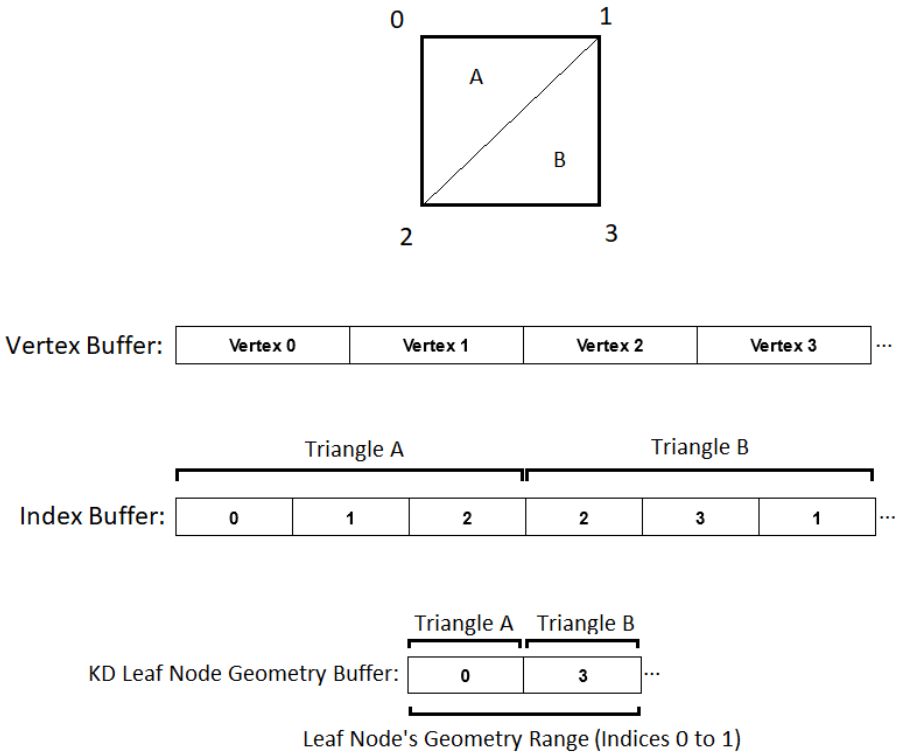

Figure 3.4 - KD Leaf Node Geometry, Vertex and Index Buffer Layout

The KD-tree itself is modelled as a balanced binary tree. To begin the construction of the tree, a root node is generated, which contains the entire scene geometry data. This root node's AABB is created using the scene geometry's most extreme vertex on each axis. To partition the scene, this root AABB is recursively split along the X and Z axes, by selecting a splitting plane halfway between the most extreme vertices on the split axis. When performing each split, scene geometry contained within the parent node is placed in the new child nodes, by identifying overlaps between the geometry and child nodes AABBs. This overlap check is done using the fast ray box intersection detection method (Woo, 1990), treating each triangle side as a ray. Geometry is then removed from the parent node, and as such, once KD-tree construction is complete, only leaf nodes will contain references to scene geometry. Geometry to be added to leaf nodes is appended to the leaf node geometry buffer, and the start and end indices of the appended range are added to the leaf node, which is then appended to the KD-node buffer.

Once the KD-tree has been constructed, the KD-node and leaf node geometry data are transferred to storage buffers in GPU memory. These storage buffers are then linked to the same descriptor set as the geometry vertex and index buffers, which is bound prior to the compute pipeline execution, thus providing the compute shader with access to the KD-tree data.

See appendix 6.1.12 for full propagation GLSL shader code showing the new input buffers and data layout.
See appendix 6.1.15 for KD-tree construction C++ code.
See appendix 6.1.16 for KD-tree data transfer to GPU memory C++ code.

Shader GLSL Implementation

The base audio propagation shader is extended with the KD-tree optimisation. As such the majority of the shader functionality has not changed.

The shader's findClosestIntersection function is updated to take advantage of the KD-tree. In the base shader, this function simply performs intersection tests against the entire scene geometry for each ray in order to find an intersection. The updated shader however uses a stack of KD-nodes which is seeded with the KD-tree's root node. While this stack is not empty, the top KD-node is removed, and an intersection test is performed between the ray and the KD-node's AABB using the fast ray-box intersection method (Woo, 1990). If an intersection is found, one of two actions are performed. If the KD-node is an inner node, the node's children are added to the stack. If the KD-node is a leaf node, intersection tests are performed between the ray and the geometry contained within the leaf node.

This process is performed for each ray, traversing from the root of the KD-tree to leaf nodes in order to find the closest geometry intersection for the ray.

See appendix 6.1.12 for the full, updated GLSL shader code.
See appendix 6.1.13 for GLSL implementation of the fast ray-box intersection method (Woo, 1990).
See appendix 6.1.14 for GLSL closest intersection code, updated to traverse the KD-tree.

Shader Outputs

The shader output remains the same as the base audio propagation shader, with additional metrics data concerning the number of AABB intersection tests performed.

See appendix 6.1.12 for the full shader GLSL code, showing the updated layout of the output data.

3.4.2 Experiments

To gather KD-tree performance data, six KD-trees were constructed, with depths of three to eight.

As with the base audio propagation shader experimentation, this shader was executed for each audio source one hundred times. However, for these experiments, the shader was executed for each geometry detail level, ray detail level and KD-tree depth combination.

Along with the data gathered during base shader experimentation, the following additional data was gathered here:

- Total AABB intersection tests in each thread
- Successful AABB intersection tests in each thread
- Total AABB intersection tests across all threads
- Successful AABB intersection tests across all threads
- Mean AABB intersection tests across all threads
- Mean AABB successful intersection tests across all threads

3.5 Ray Intersection Prediction

This section details the implementation of the prediction approach used to optimise the 3D space partitioning shader, the experiments used to evaluate optimisation, and the results of those experiments. This optimisation attempts to predict the leaf node of the KD-tree in which rays will find the closest geometry intersection, thus reducing the number of ray-AABB intersection tests required.

This is done by spatially hashing rays in such a way that similar rays produce a hash collision. When a ray-geometry intersection is found, the ray hash and the index of the leaf node in the KD-tree buffer are added to a prediction table. This hash and leaf node index can be used by subsequent rays to predict where they can find the closest geometry intersection in the scene, avoiding traversal of the KD-tree.

The ray hashing method used here is similar to that used by Liu et al. (2021), and is described below.

3.5.1 Implementation

Shader Inputs

The prediction optimisation consists of two shaders. One which constructs the prediction table, and the audio propagation shader used previously, updated to use the prediction table.

Two additional input storage buffers were created to store the prediction table and the data required to build it, both bound to the same descriptor set as the geometry and KD-tree buffers. The first of the new buffers, the ray collision buffer, replaces the ray queue introduced in the base shader to track rays to be processed. This new buffer doubles as a ray queue, and a history of rays processed by each

thread during the execution of the shader. For each thread, the buffer stores the details of each ray, and the index to the KD-tree leaf node in which the ray's closest geometry intersection was found. The second of the new buffers, the ray collision prediction buffer, stores the prediction table. The prediction table is structured as a two dimensional array of integers, in which the first dimension corresponds to a hashed ray origin, and the second corresponds to a hashed ray direction. The integers stored in the table are indices of leaf nodes in the KD-tree buffer.

The existing audio propagation shader is updated to accept these two new buffers as inputs. This shader writes data on all rays processed to the ray collision buffer, and reads predictions from the ray collision prediction buffer.

The new shader, the prediction table generator, also accepts the two new buffers as input. This shader reads processed ray data from the ray collision buffer, and uses this to construct the prediction table which is written to the ray collision prediction buffer. The prediction table generator also accepts the KD-tree buffer as an input as the AABB of the root KD-node is needed to spatially hash a ray.

See appendix 6.1.17 for full GLSL shader code showing the new input buffers and data layout.
See appendix 6.1.20 for full prediction table generator GLSL shader code.
See appendix 6.1.21 for new buffer creation C++ code snippet.

Shader GLSL Implementation

Two hash functions are used to spatially hash a ray, the first hashes a ray's origin, the seconds hashes a ray's direction.

A ray's origin is hashed by first defining an n by n grid on the xz plane, then mapping the x and z coordinates of the ray origin to the range [0, n), using the extents of the KD-tree's root AABB. This gives the two dimensional coordinate of the grid square in which the ray origin resides. This two dimensional coordinate is then mapped to the range $[0, n^2)$ to serve as the first of two indices into the ray collision prediction table.

A ray's direction is hashed by first defining a circular area divided into n equal angle sectors on the xz plane, then utilising the atan function with the ray direction's x and z components to determine in which sector the direction resides. This maps the ray direction to the range [0, n) which serves as the second of two indices into the ray collision prediction table.

The existing audio propagation shader was updated to use the above ray hash functions in an attempt to predict the KD-tree leaf node in which the closest geometry intersection will be found for a ray. For

each ray, prior to traversing the KD-tree, the hashes of the ray's origin and direction are used to index into the ray collision prediction buffer. If a prediction is found, intersection tests are performed between the ray and the geometry in the predicted leaf node. If an intersection is found in the predicted leaf node, KD-tree traversal is skipped, and the found intersection is used to reflect the ray. If no intersection was found in the predicted leaf node, KD-tree traversal is performed, from the root, as usual. Once an intersection is found for a ray, the details of the ray, and the index of the leaf node in which the intersection was found, are placed in the ray collision buffer.

As mentioned previously, along with the existing audio propagation shader, a new prediction table generator shader was created. This shader uses the contents of the ray collision buffer to generate the prediction table. For each entry in the ray collision buffer, the details of the ray are hashed and used as indices to insert the KD-tree leaf node index into the ray collision prediction buffer.

In order to execute the new prediction table generator shader, a new compute pipeline was required.

See appendix 6.1.17 for full updated audio propagation GLSL shader code.
See appendix 6.1.18 for GLSL closest intersection code, updated to predict ray intersections.
See appendix 6.1.19 for GLSL ray processing code, updated to populate the ray collision buffer.
See appendix 6.1.20 for full prediction table generator GLSL shader code.
See appendix 6.1.22 for ray hashing functions GLSL shader code.
See appendix 6.1.23 for prediction table generator pipeline creation C++ code.
See appendix 6.1.24 for prediction table generator pipeline execution C++ code.

Shader Outputs

The shader output remains the same as the space partitioning audio propagation shader, with additional metrics data concerning the accuracy of ray intersection predictions.

See appendix 6.1.17 for the full propagation shader GLSL code, showing the updated layout of the output data.

3.5.2 Experiments

To gather prediction performance data, five levels of ray hashing resolution were defined, each with a differing origin hash grid dimensions and direction hash sector counts. These are displayed in table 3.3.

Hash Resolution Level	Origin Hash Grid Dimensions	Direction Hash Sectors
1	128	8
2	256	16
3	512	32
4	1024	64
5	2048	128

Table 3.3 - Ray Hash Resolution Levels

For each geometry detail level and ray hash resolution level combination, each audio source was triggered one hundred times, with a KD-tree depth of 6, and 512 rays originating at the audio source position, using 512 GPU threads. A KD depth of 6 was identified as the most optimal in chapter 4.

Along with the data gathered during space partitioning shader experimentation, the following additional data was gathered here:

- Total predictions attempted in each thread
- Successful predictions in each thread
- Total predictions attempted across all threads
- Successful predictions across all threads
- Mean predictions attempted across all threads
- Mean predictions across all threads

4. RESULTS EVALUATION AND DISCUSSION

This chapter contains the results of the experiments performed on the base, space partitioning, and intersection prediction shaders. An exploratory overview of the base shader results are presented, followed by a discussion of the partitioning and prediction results, including statistical correlation tests to identify statistically significant improvements regarding execution time and intersections performed. Finally, the results of these statistical tests are used to test the research hypothesis and answer the research question.

4.1 Base Shader

In this section the results of experiments performed on the base audio propagation shader are presented.

Figure 4.1 shows the mean execution time, in microseconds, of all audio sources, each executed one hundred times, for each geometry detail level and initial ray count combination. Table 4.1 shows the same data.

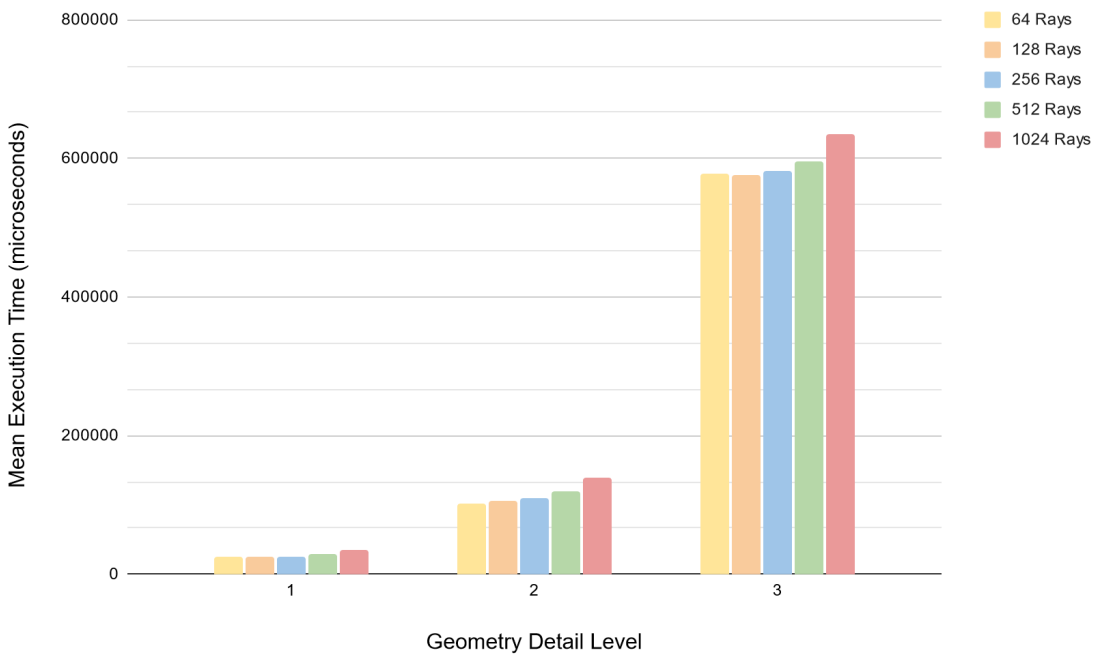

Figure 4.1 - Base Shader - Mean Execution Time Across All Audio Sources (Microseconds)

Geometry Detail Level	64 Rays	128 Rays	256 Rays	512 Rays	1024 Rays
1	25522.704	25640.61	25614.412	29406.824	34703.81
2	102730.218	105286.834	108960.44	119132.51	138515.218
3	578682.59	576009.792	582609.036	596418.998	634608.736

Table 4.1 - Base Shader - Mean Execution Time Across All Audio Sources (Microseconds)

As shown in table 4.1, all executions have a mean execution time between 25522.704 to 634608.736 microseconds, or 25.5227 to 634.6087 milliseconds. Only four of the above geometry/ray detail level combinations currently support the requirement of 30 milliseconds or less laid out in the research question, 64 to 521 rays at geometry detail level 1.

Tables 4.2 to 4.4 show the number of intersection tests performed in the execution shown in table 4.1 above. Table 4.2 shows the mean count of ray-geometry intersection tests performed. Table 4.3 shows the mean count of successful ray-geometry intersection tests performed, the number of intersection tests in which an intersection was found. Table 4.4 shows the percent of successful ray-geometry intersection tests performed.

Geometry Detail Level	64 Rays	128 Rays	256 Rays	512 Rays	1024 Rays
1	2976064	5647600	10219048	22350832	44458960
2	11887232	22618400	40829376	88923296	176494528
3	47943168	90915328	164334464	357935872	711803904

Table 4.2 - Base Shader - Mean Intersection Tests Across All Audio Sources

Geometry Detail Level	64 Rays	128 Rays	256 Rays	512 Rays	1024 Rays
1	42004.8	80085	143566.6	317224.6	630900.6
2	63610.6	123352.6	221833	483441.2	954868
3	74315	141388.8	250748.6	554945	1095721.6

Table 4.3 - Base Shader - Mean Successful Intersection Tests Across All Audio Sources

Geometry Detail Level	64 Rays	128 Rays	256 Rays	512 Rays	1024 Rays
1	1.41142126	1.41803598	1.404892119	1.419296606	1.419062884
2	0.5351170062	0.5453639515	0.543317145	0.5436609097	0.5410184728
3	0.1550064443	0.1555170103	0.1525843051	0.155040342	0.1539358795

Table 4.4 - Base Shader - Percent of Successful Intersection Tests Across All Audio Sources

Given that the base audio propagation shader performs intersection tests against the entire scene geometry for each ray, iterating over the geometry triangles and testing each in turn, a low number of

successful tests relative to the total count is expected. This is expected to be improved by the 3D space partitioning shader, the aim of which is to reduce the number of geometry intersection tests performed.

Figure 4.2 and table 4.5 show the mean transfer time, in microseconds, of the shader execution results from GPU memory to host system memory based on the ray detail level. As expected, the time required increases as the number of initial rays, and therefore results to be transferred, increases.

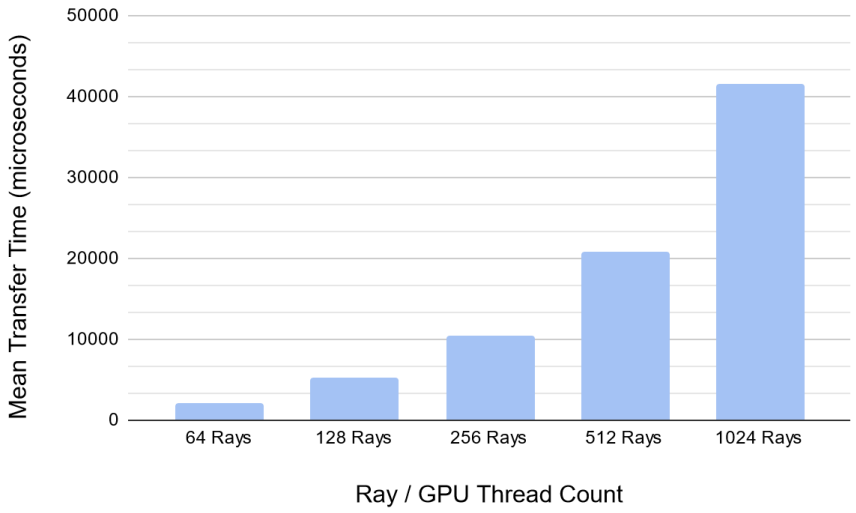

Figure 4.2 - Base Shader - Mean Shader Results Transfer Time

64 Rays	128 Rays	256 Rays	512 Rays	1024 Rays
2158.464667	5282.096667	10446.47533	20840.744	41570.94933

Table 4.5 - Base Shader - Mean Shader Results Transfer Time (Microseconds)

Revisiting the execution time of the base shader, as shown in table 4.1, only 64 to 512 rays at geometry level detail 1 satisfy the requirement of 30 milliseconds laid out in the research question. However, when incorporating the mean data transfer time, only 64 rays at geometry detail level 1 satisfies this requirement.

In this section the results of experimentation on the base shader were presented. It was identified that, when considering both shader execution time and time taken to transfer the shader results data, only one geometry/ray detail level configuration met the requirements of an execution time of 30 milliseconds or less. Metrics on the number of intersection tests performed during shader execution were also presented.

In the next section, the results of experimentation on the 3D space partitioning shader are presented and compared to the base shader results given here.

4.2 3D Space Partitioning Shader

In this section the results of experiments performed on the 3D space partitioning audio propagation shader are presented, and correlation tests are performed between these results and the results of the base audio propagation shader in order to identify any statistically significant improvements as a result of the 3D space partitioning optimisation.

Figure 4.3 shows the mean execution time, in microseconds, of all audio sources, each executed one hundred times, for each KD-tree depth, geometry detail level and ray detail level combination. Tables 4.6 to 4.11 show the same data.

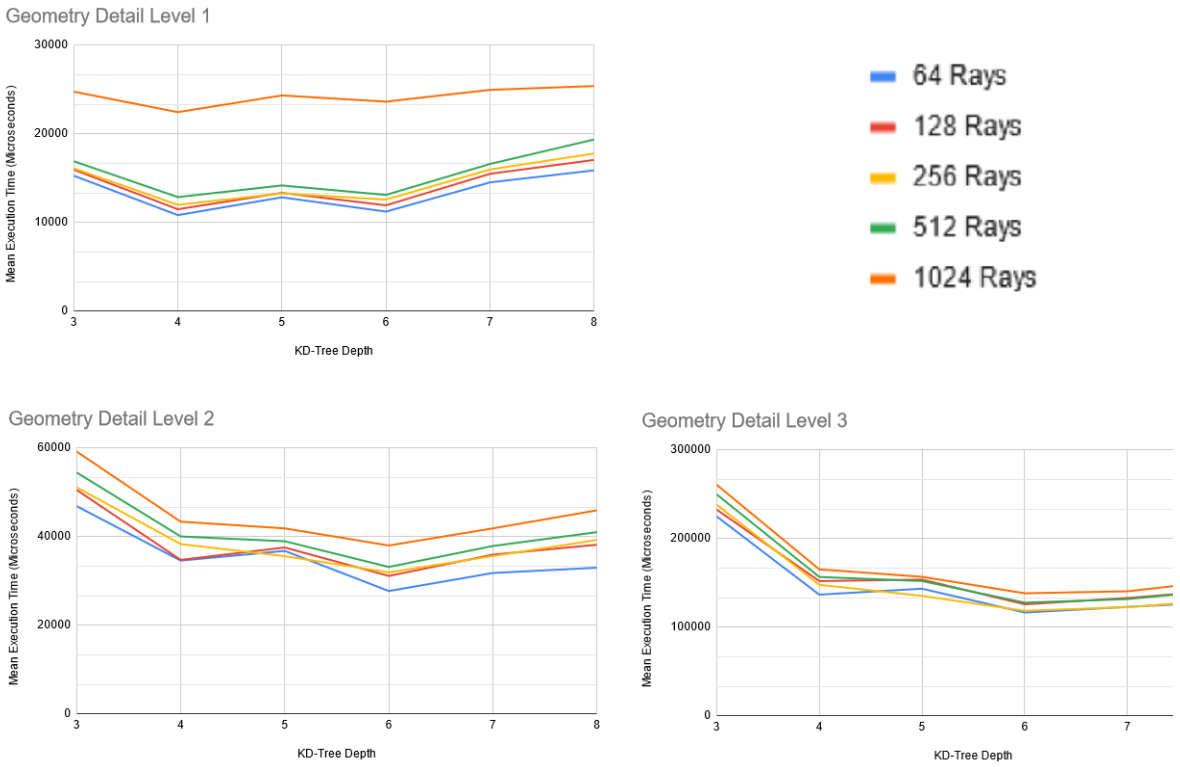

Figure 4.3 - Partition Shader - Mean Execution Time By KD-Tree Depth (Microseconds)

Geometry Detail Level	64 Rays	128 Rays	256 Rays	512 Rays	1024 Rays
1	15272.422	15952.596	16104.776	16904.704	24745.384
2	46860.168	50522.952	51114.916	54448.47	59172.454
3	224977.398	232635.186	238212.874	249878.064	260685.206

Table 4.6 - Partition Shader - Mean Execution Time of KD-Tree Depth 3 (Microseconds)

Geometry Detail Level	64 Rays	128 Rays	256 Rays	512 Rays	1024 Rays
1	10827.018	11490.554	11987.864	12861.326	22435.74
2	34581.014	34741.352	38303.184	40021.858	43356.386
3	136670.734	151949.502	147780.364	156634.746	165200.108

Table 4.7 - Partition Shader - Mean Execution Time of KD-Tree Depth 4 (Microseconds)

Geometry Detail Level	64 Rays	128 Rays	256 Rays	512 Rays	1024 Rays
1	12833.434	13334.89	13277.912	14168.168	24320.346
2	36766.694	37528.97	35576.84	38941.364	41846.668
3	143394.24	153662.11	135289.328	151984.734	156778.59

Table 4.8 - Partition Shader - Mean Execution Time of KD-Tree Depth 5 (Microseconds)

Geometry Detail Level	64 Rays	128 Rays	256 Rays	512 Rays	1024 Rays
1	11223.878	11933.022	12577.294	13104.462	23619.992
2	27719.784	31115.972	31928.618	33146.402	37982.858
3	116555.976	125761.076	118501.608	127645.354	138231.216

Table 4.9 - Partition Shader - Mean Execution Time of KD-Tree Depth 6 (Microseconds)

Geometry Detail Level	64 Rays	128 Rays	256 Rays	512 Rays	1024 Rays
1	14516.812	15484.26	15971.476	16595.932	24950.708
2	31785.228	35928.876	35609.284	37868.886	41849.28
3	122865.508	132945.304	122610.914	131662.572	140438.232

Table 4.10 - Partition Shader - Mean Execution Time of KD-Tree Depth 7 (Microseconds)

Geometry Detail Level	64 Rays	128 Rays	256 Rays	512 Rays	1024 Rays
1	15873.124	17048.876	17773.94	19351.752	25378.31
2	32981.298	38151.98	39225.624	41001.148	45917.938
3	128784.198	142157.878	131085.788	142649.668	153463.76

Table 4.11 - Partition Shader - Mean Execution Time of KD-Tree Depth 8 (Microseconds)

From the above execution data, it is visible that a KD-depth of 6 is the most optimal depth across all geometry/ray detail level combinations. Figure 4.4 shows the AABBs of the virtual environment, with a KD-tree depth of 6, viewed from above. AABBs are visible in yellow.

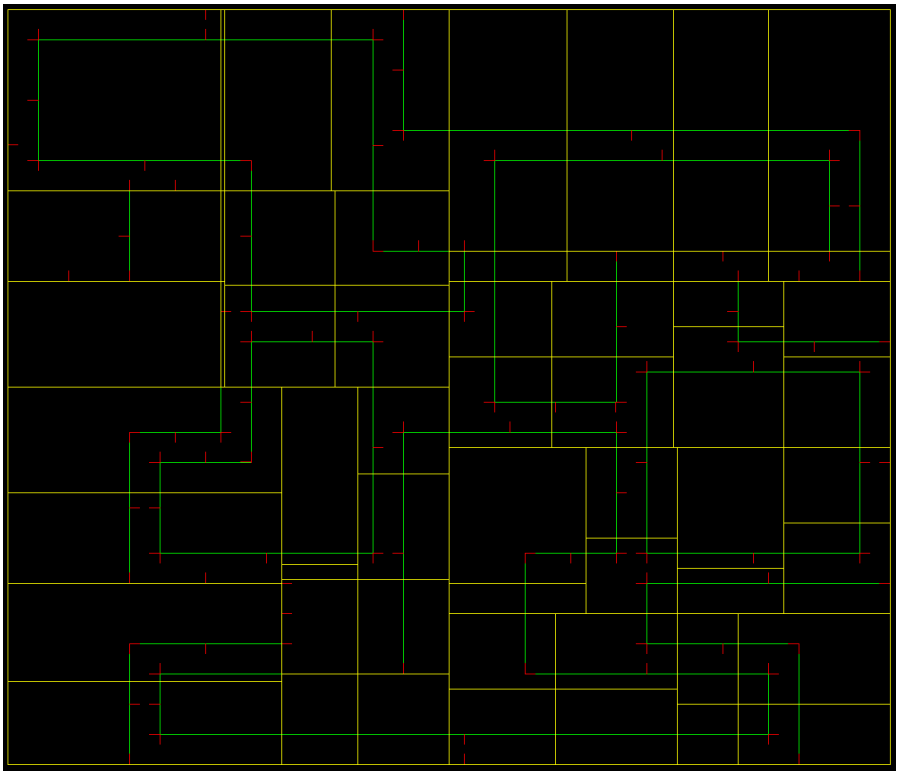

Figure 4.4 - Partition Shader - Virtual Test Environment AABBs

Shown in table 4.9, with a KD-depth of 6, six geometry/ray detail level combinations support the requirement, laid out by the research question, of an execution time under 30 milliseconds. However, when the mean data transfer times shown in table 4.5 are considered, only three of the geometry/ray detail level combinations support the requirement. These significant executions are laid out in table 4.12, and will be referred to, from here on, by their assigned execution ID, and collectively, as the significant executions.

Execution ID	Geometry Detail Level	Ray Detail Level	KD-Tree Depth
1	1	64	6
2	1	128	6
3	1	256	6

Table 4.12 - Partition Shader - Significant Executions

Table 4.13 shows the intersection test metrics gathered during experimentation on the significant executions. AABB intersection metrics are expected to be improved by the ray intersection prediction optimisation, as this optimisation aims to reduce traversal of the KD-tree.

Execution ID	Mean Geometry Intersection Tests	Mean Successful Geometry Intersection Tests	Percent Successful Geometry Intersection Tests	Mean AABB Intersection Tests	Mean Successful AABB Intersection Tests	Percent Successful AABB Intersection Tests
1	815083.6	62632.4	7.684168839	287827.6	195653	67.97576049
2	1553595.4	120735.8	7.771379859	547020	371430.8	67.90077145
3	3081642.6	241503.8	7.836852982	1094966.6	741922.2	67.75751881

Table 4.13 - Partition Shader - Significant Execution Intersection Tests Metrics

From tables 4.9 and 4.13, a reduction in execution time and an increase in intersection test accuracy can be seen when compared to the corresponding results of base shader experimentation. In order to determine if these improvements are statistically significant, correlation tests were performed on the execution time and intersection test distributions of the base and space partitioning shaders for each of the significant executions.

Given that the distributions represent before and after the implementation of the space partitioning optimisation, a paired test was performed on the data. In order to determine which test to perform, a Shapiro-Wilk test was performed to determine the normality of the differences in the distributions. For each, a significant departure from normality, with outliers, was found and so a paired-samples sign test was chosen.

Figure 4.5 shows the histograms and Q-Q plots of the differences in the execution time distributions for the significant executions, while table 4.14 shows the Shapiro-Wilk test results related to each. Table 4.15 shows the sign test results for the significant executions, showing a significant result.

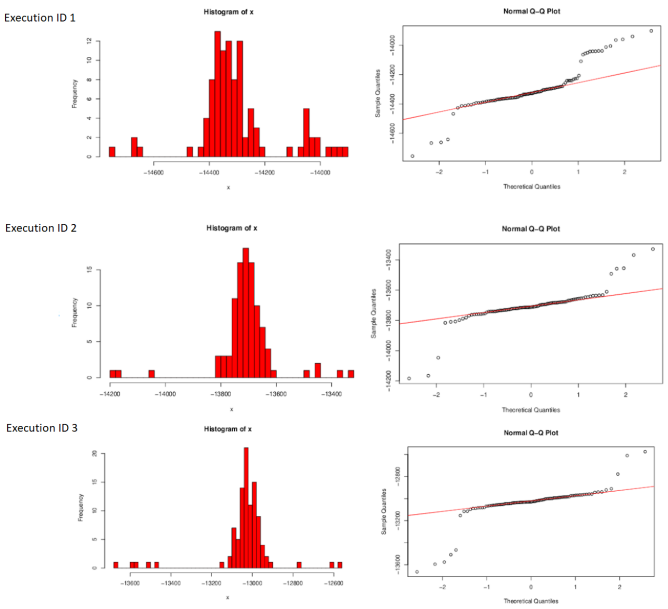

Figure 4.5 - Partition Shader - Execution Time Analysis

Execution ID	Shapiro-Wilk Results
1	W(100) = .87, p < .001
2	W(100) = .736, p < .001
3	W(100) = .65, p < .001

Table 4.14 - Partition Shader - Execution Time Shapiro-Wilk Test Results

Execution ID	Sign Test Results
1	The z-value is 10. The p-value is < .00001. The result is significant at p < 0.05.
2	The z-value is 10. The p-value is < .00001. The result is significant at p < 0.05.
3	The z-value is 10. The p-value is < .00001. The result is significant at p < 0.05.

Table 4.15 - Partition Shader - Execution Time Sign Test Results

Figure 4.6 shows the histograms and Q-Q plots of the differences in the mean intersection test distributions for the significant executions, while table 4.16 shows the Shapiro-Wilk test results related to each. Table 4.17 shows the sign test results for the significant executions, showing a significant result.

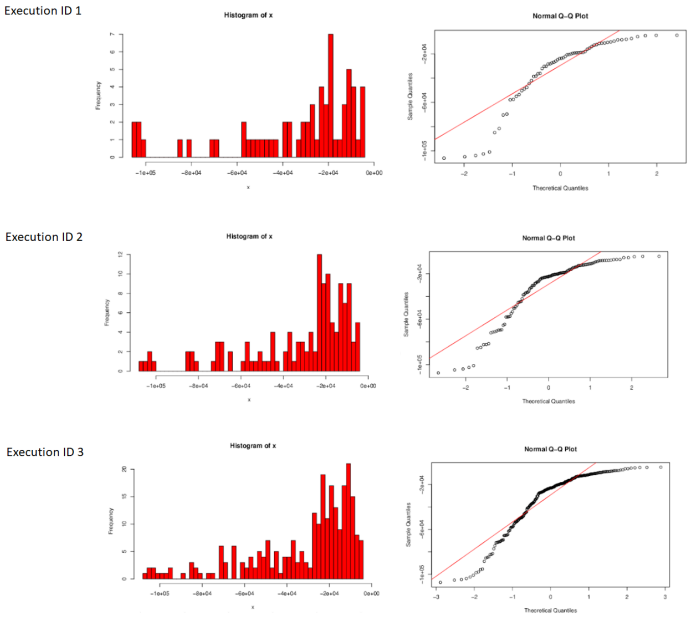

Figure 4.6 - Partition Shader - Mean Intersection Tests Analysis

Execution ID	Shapiro-Wilk Results
1	W(64) = .83, p < .001
2	W(128) = .839, p < .001
3	W(256) = .853, p < .001

Table 4.16 - Partition Shader - Mean Intersection Tests Shapiro-Wilk Test Results

Execution ID	Sign Test Results
1	The z-value is 8. The p-value is < .00001. The result is significant at p < 0.05.
2	The z-value is 11.31371. The p-value is < .00001. The result is significant at p < 0.05.
3	The z-value is 16. The p-value is < .00001. The result is significant at p < 0.05.

Table 4.17 - Partition Shader - Mean Intersection Tests Sign Test Results

Figure 4.7 shows the histograms and Q-Q plots of the differences in the mean successful intersection test distributions for the significant executions, while table 4.18 shows the Shapiro-Wilk test results related to each. Table 4.19 shows the sign test results for the significant executions, showing a significant result.

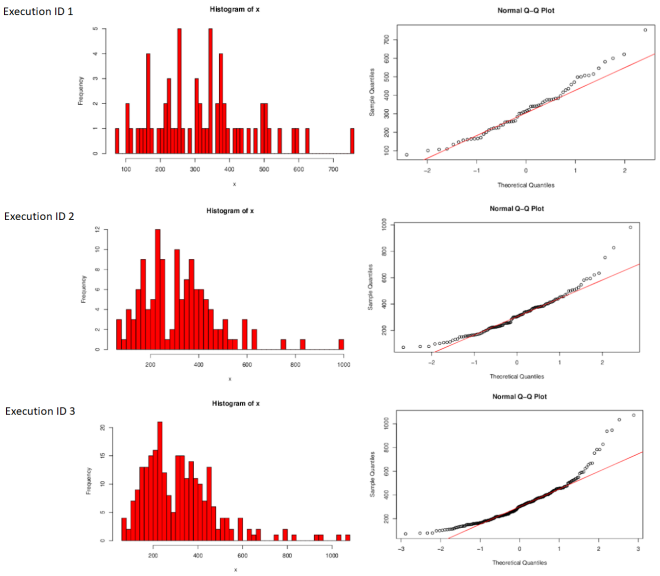

Figure 4.7 - Partition Shader - Mean Successful Intersection Tests Analysis

Execution ID	Shapiro-Wilk Results
1	W(64) = .97, p = .126
2	W(128) = .934, p < .001
3	W(256) = .897, p < .001

Table 4.18 - Partition Shader - Mean Successful Intersection Tests Shapiro-Wilk Test Results

Execution ID	Sign Test Results
1	The z-value is 8. The p-value is < .00001. The result is significant at p < 0.05.
2	The z-value is 11.31371. The p-value is < .00001. The result is significant at p < 0.05.
3	The z-value is 16. The p-value is < .00001. The result is significant at p < 0.05.

Table 4.19 - Partition Shader - Mean Successful Intersection Tests Sign Test Results

In this section, the results of experimentation on the 3D space partitioning shader were presented. It was identified that, when considering both shader execution time and time taken to transfer the shader results data, only the significant executions in table 4.12 met the requirements of an execution time of 30 milliseconds or less. For these executions correlation tests were performed against the base shader on the shader's execution time, mean intersection test and mean successful intersection test

distributions. For each of these distributions, it was found that the implementation of 3D space partitioning resulted in a statistically significant reduction in execution time, and a statistically significant increase in intersection test accuracy.

In the next section, the results of experimentation on the ray intersection prediction shader are presented and compared to the partition shader results given here.

4.3 Ray Intersection Prediction Shader

In this section the results of experiments performed on the two ray intersection prediction shaders are presented, and correlation tests are performed between these results and the results of the 3D space partitioning shader in order to identify any statistically significant improvements as a result of the ray intersection prediction optimisation.

Figure 4.8 shows the mean execution time of the audio propagation shader, in microseconds, of all audio sources, each executed one hundred times, for each ray hash resolution level, geometry detail level and ray detail level combination. Each execution uses a KD-Tree of depth 6. Tables 4.20 to 4.24 show the same data.

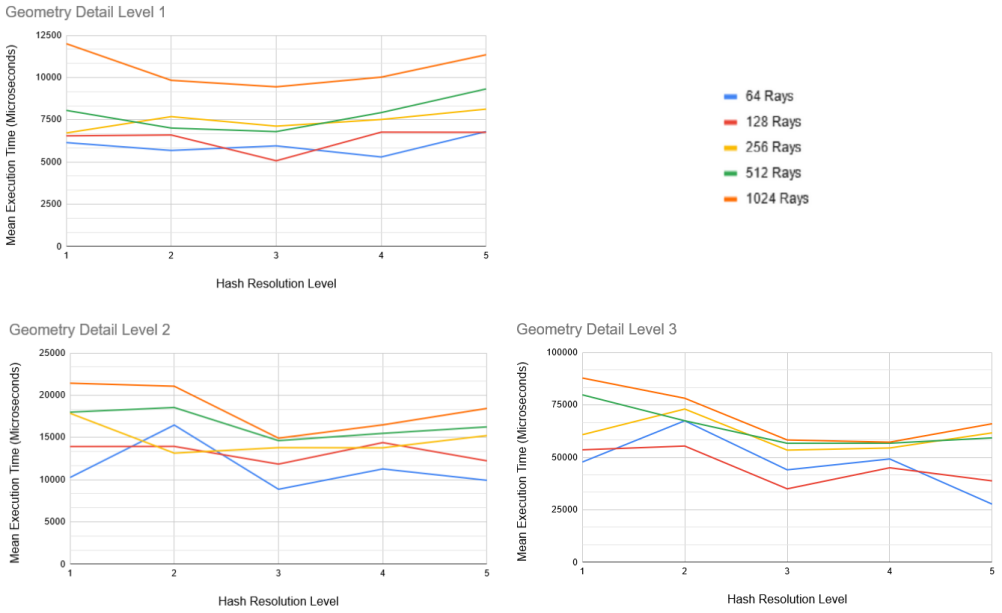

Figure 4.8 - Prediction Shader - Mean Execution Time By Hash Resolution (Microseconds)

Geometry Detail Level	64 Rays	128 Rays	256 Rays	512 Rays	1024 Rays
1	6149.144	6549.446	6718.438	8059.378	12008.212
2	10260.588	13932.544	17870.016	18001.72	21433.32
3	47758.588	53595.748	60781.716	79807.148	87818.748

Table 4.20 - Prediction Shader - Mean Execution Time Ray Hash Resolution 1 (Microseconds)

Geometry Detail Level	64 Rays	128 Rays	256 Rays	512 Rays	1024 Rays
1	5683.934	6599.648	7688.646	7011.094	9838.09
2	16474.712	13948.232	13151.248	18549.214	21071.612
3	67441.744	55394.772	73031.032	67435.868	78171.976

Table 4.21 - Prediction Shader - Mean Execution Time Ray Hash Resolution 2 (Microseconds)

Geometry Detail Level	64 Rays	128 Rays	256 Rays	512 Rays	1024 Rays
1	5956.26	5073.028	7123.624	6799.906	9453.644
2	8870.1	11850.07	13799.212	14628.286	14920.826
3	44050.48	35003.728	53450.036	56708.24	58285.136

Table 4.22 - Prediction Shader - Mean Execution Time Ray Hash Resolution 3 (Microseconds)

Geometry Detail Level	64 Rays	128 Rays	256 Rays	512 Rays	1024 Rays
1	5296.852	6768.812	7518.128	7927.264	10026.652
2	11276.906	14395.836	13782.738	15490.364	16494.128
3	49242.648	45037.348	54470.404	56738.096	57248.184

Table 4.23 - Prediction Shader - Mean Execution Time Ray Hash Resolution 4 (Microseconds)

Geometry Detail Level	64 Rays	128 Rays	256 Rays	512 Rays	1024 Rays
1	6803.302	6760.61	8134.382	9330.344	11352.5
2	9932.654	12235.95	15228.422	16256.258	18446.06
3	27696.104	38781.288	61673.44	59294.008	66027.672

Table 4.24 - Prediction Shader - Mean Execution Time Ray Hash Resolution 5 (Microseconds)

As shown in figure 4.8 and tables 4.20 to 4.24, there is a noticeable decrease in audio propagation shader execution time when compared to the partitioning shader, with many executions supporting the requirement of a 30 millisecond execution time. However a number of issues were identified with the ray intersection prediction implementation, which will be shown below using 512 rays at geometry level 3, with a KD-tree depth of 6 and a ray hash resolution level of 5, referred to from hereon as the prediction example execution.

First, to confirm the execution time improvement using the prediction example execution, a correlation test was performed on the execution time distributions of the prediction and partitioning shaders. Similar to the partition shader correlation tests, a Shapiro-Wilk normality test was first performed to determine what correlation test should be performed. The Shapiro-Wilk tests showed a significant departure from the normality, $W(100) = .126$, $p < .001$. The histogram and Q-Q plots of the differences in the distributions are shown in figure 4.9.

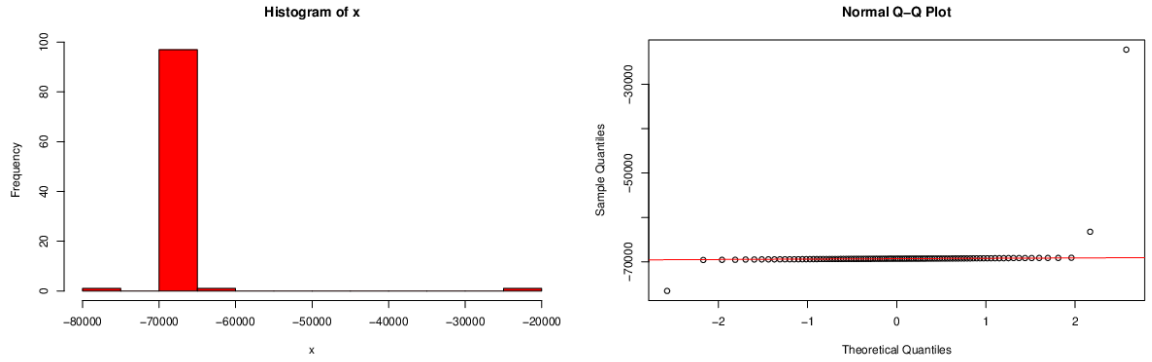

Figure 4.9 - Prediction Shader - Execution time Analysis

Given the Shapiro-Wilk test results, a sign test was performed on the execution time distributions. The results of which show a statistically significant decrease in execution time. Table 4.25 shows the sign test results.

Z Value	P Value	Significance Value
9.8	< .00001	.05

Table 4.25 - Prediction Shader - Execution Time Sign Test Results

However, as mentioned, a number of issues were uncovered with the ray intersection prediction implementation. The first of these is the prediction error rate. Using the prediction example execution, the mean predictions for each hash resolution level, over one hundred shader executions, are shown in figure 4.10. The same data is shown, per audio source, in table 4.26.

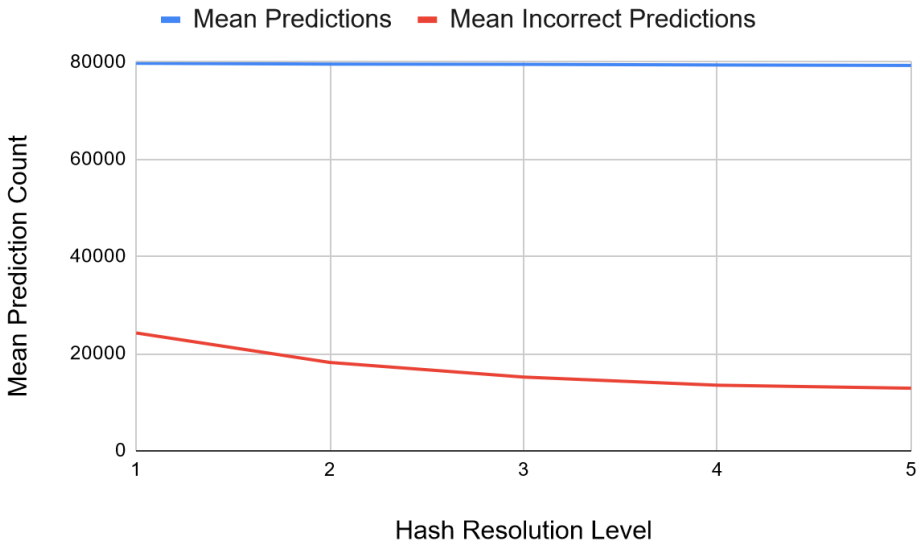

Figure 4.10 - Prediction Accuracy

Prediction Detail Level	Total Predictions	Incorrect Predictions	Mean Predictions	Mean Incorrect Predictions
1	39886047	12130053	79772.094	24260.106
2	39799213	9094072	79598.426	18188.144
3	39774807	7590112	79549.614	15180.224
4	39709729	6751227	79419.458	13502.454
5	39662891	6442806	79325.782	12885.612

Table 4.26 - Prediction accuracy

As shown in table 4.26, there is an error rate of 16.2439% - 30.4118%.

Figure 4.11 shows rays passing through scene geometry and leaving the scene confins as a result of incorrect predictions.

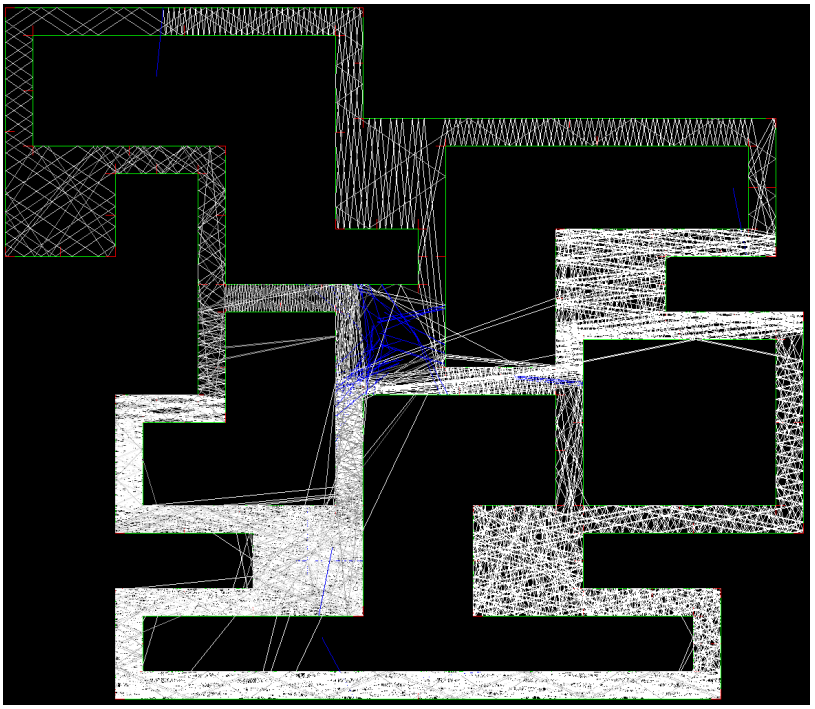

Figure 4.11 - Prediction Shader - Prediction Errors

In order to reduce the error rate, the ray hash resolution must be increased. The second issue however, is an exponential increase in GPU memory usage results in a logarithmic decline in error rate. Figure 4.12 shows the GPU memory requirements (Bytes) of the prediction table for each hash resolution level shown in table 3.3. Table 4.27 shows the same data.

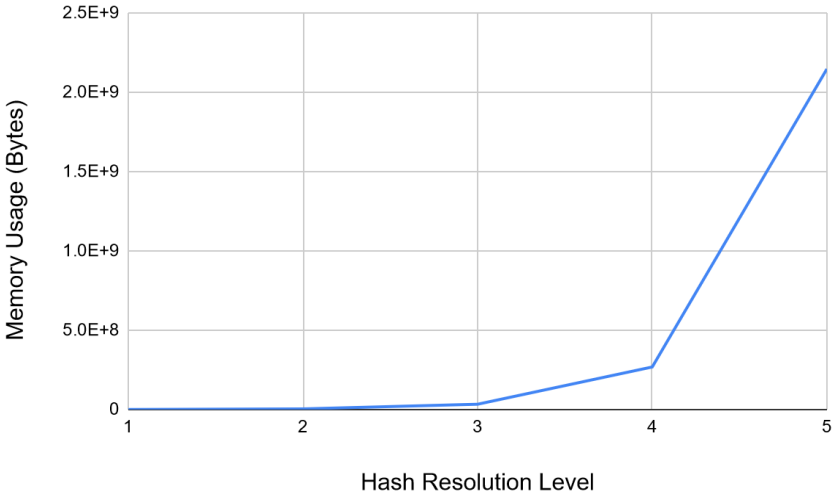

Figure 4.12 - Prediction Table Memory Usage (Bytes)

Hash Resolution Level	Memory Usage (Bytes)
1	524288
2	4194304
3	33554432
4	268435456
5	2147483648

Table 4.27 - Prediction Table Memory Usage (Bytes)

The third and final issue with the current prediction implementation is the execution time of the prediction table generation shader. Figure 4.13 shows the mean execution time (microseconds) of the audio propagation and prediction table generation shaders. Table 4.28 shows the same data. As shown in table 4.28, there is a mean prediction table generation time of 497.6635 to 546.27 milliseconds, which does not support the requirement of an execution time under 30 milliseconds.

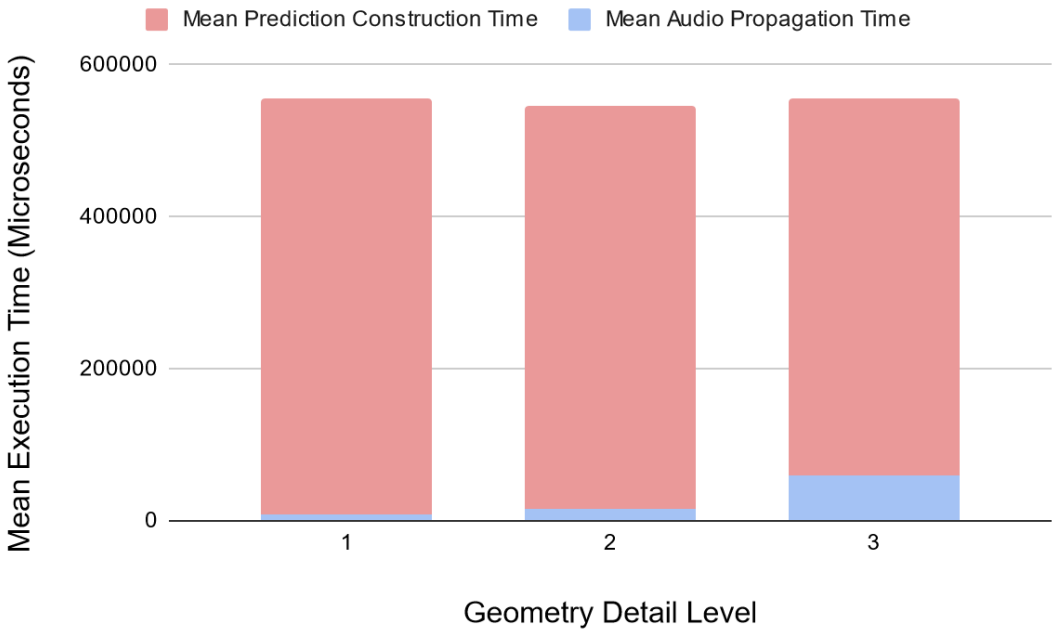

Figure 4.13 - Prediction Execution Time (Microseconds)

Geometry Detail Level	Mean Audio Propagation Time	Mean Prediction Construction Time
1	9302.494	546270.02
2	16497.22	528746.12
3	58785.928	497663.46

Table 4.28 - Prediction Execution Time (Microseconds)

In this section the results of experimentation on the ray intersection prediction shaders were presented. It was identified that, while there is a statistically significant decrease in audio propagation shader execution time, the prediction table generation shader execution time exceeds the 30 millisecond execution time requirement. Also, unacceptable prediction error rates and GPU memory requirements were uncovered. Due to these issues, the current ray intersection prediction implementation is deemed to be infeasible.

4.4 Hypothesis

Research Question: Can spatial acceleration structures and ray intersection prediction be used to enhance real-time ray-traced audio propagation in 3D virtual environments using GLSL Vulkan compute shaders, so that results are returned to the host system within 30 milliseconds of the triggering event?

Null Hypothesis (H0): If 3D space partitioning and/or ray intersection prediction are used to optimise ray traced audio propagation within a GLSL Vulkan compute shader, then audio propagation results will not be returned to the host system within 30 milliseconds of the triggering event.

Alternate Hypothesis (H1): If 3D space partitioning and/or ray intersection prediction are used to optimise ray traced audio propagation within a GLSL Vulkan compute shader, then audio propagation results will be returned to the host system within 30 milliseconds of the triggering event.

As shown in the analysis of the base shader experiment results, a Vulkan GLSL Compute shader can be used to propagate audio in real time using ray tracing. Experiments run on the base shader show this is possible in a scene with up to 560 vertices, and 840 indices. However, these values may be implementation specific and as such, further research into real time ray traced audio is warranted to build a more optimal baseline shader implementation.

Correlation tests performed between the results of the base and partition shaders showed a statistically significant improvement in both shader execution time and ray-geometry intersection test accuracy. As such the results of the experiments performed during this research support the alternate hypothesis in the case of the 3D space partitioning optimisation. Though more research can be undertaken to further optimise this solution.

While correlation tests performed between the partition and prediction shaders showed a statistically significant improvement in both shader execution time and intersection test accuracy, several issues were uncovered regarding the current prediction implementation. It was found that prediction error

rates, GPU memory requirements and prediction table construction time were too great, and as such the current prediction implementation is deemed infeasible. Because of this, in the case of the ray intersection prediction optimisation, the results of the experiments performed during this research support the null hypothesis. Though more research can also be undertaken to further optimise this solution.

5. CONCLUSION

This chapter provides an overview of the research performed in this project, reiterates the findings of the research, discusses the research contribution to the domain, and highlights potential areas of future work.

5.1 Research Overview and Problem Definition

The aim of this research was to determine the extent to which 3D space partitioning and ray intersection prediction could be used to optimise Vulkan GLSL compute shaders so as to be used to propagate audio through a virtual environment in real time using ray tracing, as an alternative to commonly utilised GPGPU methods such as CUDA.

5.2 Design/Experimentation, Evaluation & Results

This research details the implementation of a number of GLSL compute shaders, and a C++ Vulkan application with which to execute them. These shaders facilitate the propagation of audio, using ray tracing, through a virtual environment, and implement space partitioning and intersection prediction in order to gauge the effectiveness of these optimisations for this task.

In order to determine the effectiveness, the compute shaders were executed with a variety of ray counts, geometry vertex counts, KD-tree depths and ray spatial hashing resolutions. Data was gathered during these executions, which was analysed with statistical tests to identify any statistically significant improvements in execution time and ray intersection test accuracy. It was found that the 3D space partitioning optimisation successfully increased shader performance, allowing audio propagation results to be returned to the host system within 30 milliseconds. However, while performance improvements were shown, significantly detailed virtual environments prevented results from being returned in real time. As a result of the experiments performed on the ray intersection prediction optimisation, it was determined that the implementation presented in this research is infeasible. This is due to a high prediction error rate. While this can be addressed by increasing the ray spatial hashing resolution, this requires much more GPU memory to be allocated to storing the prediction table. It was shown that an exponential increase in GPU memory usage resulted in a logarithmic decline in error rate.

5.3 Contributions and impact

This research expands upon the existing body of research within the domains of GPGPU and audio processing. While much research has been done within the GPGPU and audio processing domains using platforms such as CUDA, a lack of research utilising GLSL compute shaders motivated this

research to begin filling that gap. Using 3D space partitioning, this research has shown that GLSL compute shaders are a potential solution to real time ray traced audio tasks. However, virtual environment detail is the primary roadblock to adoption. Further research is warranted regarding ray intersection prediction, as the issues presented arose from the implementation chosen. Other implementations may be more efficient or accurate.

5.4 Future Work & recommendations

This research can be expanded upon in a number of ways.

Base Shader Optimization

As the base shader presented in this research served as the foundations upon which the space partitioning and prediction optimisations were built, the implementation of the base shader may heavily impact the benefits provided by these optimisations. Therefore, research into more optimal baseline shader implementations may be warranted.

Memory Usage and Requirements

This research focused on the algorithmic side of the optimisations implemented, further research into more optimal memory usage is a possible avenue of further research. Particularly regarding the presented ray intersection prediction implementation, as memory usage was one of the major issues with the implementation.

Vulkan Compute Pipeline Optimisation

Beyond the shaders themselve, research into more optimal Vulkan compute pipeline configurations is another possibility for future research.

Results Data Transfer

The current implementation waits for all GPU threads to complete processing before transferring audio propagation results back to the host system. However, each GPU thread is unlikely to finish in equal time, therefore research into the feasibility of transferring results as they are available may be warranted.

BIBLIOGRAPHY

Alfrink, M., Reitz, J., & Roßmann, J. (2021). Improving ray tracing based radio propagation model performance using spatial acceleration structures. *Proceedings of the 17th ACM Symposium on QoS and Security for Wireless and Mobile Networks*, 25–32. https://doi.org/10.1145/3479242.3487318

Bailey, M. (2020). Introduction to the Vulkan computer graphics API. *ACM SIGGRAPH 2020 Courses*, 1-128. https://doi.org/10.1145/3388769.3407508

Beig, M., Kapralos, B., Collins, K., & Mirza-Babaei, P. (2019). G-spar: GPU-based Voxel graph Pathfinding for spatial audio rendering in games and VR. *2019 IEEE Conference on Games (CoG)* (pp. 1-8). https://doi.org/10.1109/cig.2019.8847959

Benthin, C., Woop, S., Vaidyanathan, K. (2019). Wide BVH traversal with a short stack. *Proceedings of the Conference on High-Performance Graphics - HPG '19*, 15–19. https://doi.org/10.2312/hpg.20191190

Benthin, C., Woop, S., Wald, I., & Áfra, A. T. (2017). Improved two-level bvhs using partial re-braiding. *Proceedings of High Performance Graphics*, 1–8. https://doi.org/10.1145/3105762.3105776

Carini, A., Cecchi, S., & Romoli, L. (2016). Robust room impulse response measurement using perfect sequences for Legendre nonlinear filters. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, *24*(11), 1969–1982. https://doi.org/10.1109/taslp.2016.2593803

Chen, M. (2003). A low-latency lip-synchronized videoconferencing system. *Proceedings of the conference on Human factors in computing systems - CHI '03* (pp. 465-471). https://doi.org/10.1145/642611.642692

Chen, S., & Li, X. (2013). A hybrid gpu/Cpu fft library for large FFT problems. *2013 IEEE 32nd International Performance Computing and Communications Conference (IPCCC)* (pp. 1-10). https://doi.org/10.1109/pccc.2013.6742796

Choi, B., Chang, B., & Ihm, I. (2013). Improving memory space efficiency of KD-tree for real-time Ray Tracing. *Computer Graphics Forum*, *32*(7), 335–344. https://doi.org/10.1111/cgf.12241

Cowan, B., & Kapralos, B. (2011). GPU-based acoustical occlusion modeling with acoustical texture maps. *Proceedings of the 6th Audio Mostly Conference on A Conference on Interaction with Sound - AM '11* (pp. 55-61). https://doi.org/10.1145/2095667.2095675

Crawford, L., & O'Boyle, M. (2018). A cross-platform evaluation of graphics Shader compiler optimization. *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (pp. 219-228). https://doi.org/10.1109/ispass.2018.00035

Gribble, C., Fisher, J., Eby, D., Quigley, E., & Ludwig, G. (2012). Ray tracing visualization toolkit. *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games - I3D '12* (pp. 71-78). https://doi.org/10.1145/2159616.2159628

Gu, C., Zhu, M., Lu, H., & Beckers, B. (2014). Room impulse response simulation based on equal-area ray tracing. *2014 International Conference on Audio, Language and Image Processing* (pp. 832-836). https://doi.org/10.1109/icalip.2014.7009911

Hapala, M., Davidovič, T., Wald, I., Havran, V., & Slusallek, P. (2013). Efficient stack-less BVH traversal for Ray Tracing. *Proceedings of the 27th Spring Conference on Computer Graphics - SCCG '11*, 7–12. https://doi.org/10.1145/2461217.2461219

Havran, V., & Bittner, J. (2007). Ray tracing with sparse boxes. *Proceedings of the 23rd Spring Conference on Computer Graphics - SCCG '07*, 49–54. https://doi.org/10.1145/2614348.2614356

He, Y., Foley, T., Hofstee, T., Long, H., & Fatahalian, K. (2017). Shader components. *ACM Transactions on Graphics*, *36*(4), 1-11. https://doi.org/10.1145/3072959.3073648

Hossain, M. M., Tucker, T. M., Kurfess, T. R., & Vuduc, R. W. (2015). A GPU-parallel construction of volumetric tree. *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, 1–4. https://doi.org/10.1145/2833179.2833191

Jack, R. H., Stockman, T., & McPherson, A. (2016). Effect of latency on performer interaction and subjective quality assessment of a digital musical instrument. *Proceedings of the Audio Mostly 2016* (pp. 116-123). https://doi.org/10.1145/2986416.2986428

Junker, A., & Palamas, G. (2020). Real-time interactive snow simulation using compute Shaders in digital environments. *International Conference on the Foundations of Digital Games*, 1-4. https://doi.org/10.1145/3402942.3402995

Junwei He, & Mengyao Zhu. (2013). Simulation of combined head and room impulse response based on sound ray tracing in frequency domain. *IET International Conference on Smart and Sustainable City 2013 (ICSSC 2013)* (pp. 361-365). https://doi.org/10.1049/cp.2013.1957

Kontomichos, F., Tatlas, N., Hatziantoniou, P., & Papadakos, C. (2015). PC-based room correction for audio. *Proceedings of the Audio Mostly 2015 on Interaction With Sound - AM '15*, 1-6. https://doi.org/10.1145/2814895.2814933

Kurtisi, Z., & Wolf, L. (2008). Using wavpack for real-time audio coding in interactive applications. *2008 IEEE International Conference on Multimedia and Expo* (pp. 1381-1384). https://doi.org/10.1109/icme.2008.4607701

Lagae, A., & Dutré, P. (2008). Compact, fast and robust grids for ray tracing. *Computer Graphics Forum*, *27*(4), 1235–1244. https://doi.org/10.1111/j.1467-8659.2008.01262.x

Laine, S. (2010). Restart trail for stackless BVH traversal. *Proceedings of the Conference on High Performance Graphics - HPG '10*, 107–111. https://doi.org/10.5555/1921479.1921496

Liu, L., Chang, W., Demoullin, F., Chou, Y. H., Saed, M., Pankratz, D., Nowicki, T., & Aamodt, T. M. (2021). Intersection prediction for accelerated GPU ray tracing. *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (pp. 709-723). https://doi.org/10.1145/3466752.3480097

Li, Z., Wang, T., & Deng, Y. (2014). Fully parallel KD-tree construction for real-time Ray Tracing. *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games - I3D '14*, 159–159. https://doi.org/10.1145/2556700.2566638

Luebke, D. (2009). Graphics hardware & GPU computing: past, present, and future. *In Proceedings of Graphics Interface 2009 (GI '09)*, 1–1. http://dx.doi.org/10.1145/1555880.1555888

MacDonald, J. D., & Booth, K. S. (1990). Heuristics for ray tracing using space subdivision. *The Visual Computer*, *6*(3), 153–166. https://doi.org/10.1007/bf01911006

Meister, D., Boksansky, J., Guthe, M., & Bittner, J. (2020). On ray reordering techniques for faster GPU ray tracing. *Symposium on Interactive 3D Graphics and Games*, 1-9. https://doi.org/10.1145/3384382.3384534

Möller, T., & Trumbore, B. (1997). Fast, minimum storage ray-triangle intersection. *Journal of Graphics Tools*, *2*(1), 21–28. https://doi.org/10.1080/10867651.1997.10487468

Nabata, K., Iwasaki, K., Dobashi, Y., & Nishita, T. (2013). Efficient divide-and-conquer ray tracing using ray sampling. *Proceedings of the 5th High-Performance Graphics Conference on - HPG '13* (pp. 129-135). https://doi.org/10.1145/2492045.2492059

Nickolls, J., Buck, I., Garland, M., & Skadron, K. (2008). Scalable Parallel Programming with Cuda. *ACM SIGGRAPH 2008 Classes*, 1–14. https://doi.org/10.1145/1401132.1401152

Nickolls, J., & Dally, W. J. (2010). The GPU computing era. *IEEE Micro*, *30*(2), 56–69. https://doi.org/10.1109/mm.2010.41

Nikolov, D. V., Misis, M. J., & Tomasevic, M. V. (2015). GPU-based implementation of reverb effect. *2015 23rd Telecommunications Forum Telfor (TELFOR)* (pp. 990-993). https://doi.org/10.1109/telfor.2015.7377631

Ouali, C., Dumouchel, P., & Gupta, V. (2015). GPU implementation of an audio fingerprints similarity search algorithm. *2015 13th International Workshop on Content-Based Multimedia Indexing (CBMI)* (pp. 1-6). https://doi.org/10.1109/cbmi.2015.7153625

Pankratz, D., Nowicki, T., Eltantawy, A., & Amaral, J. N. (2021). Vulkan vision: Ray tracing workload characterization using automatic graphics instrumentation. *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (pp. 137-149). https://doi.org/10.1109/cgo51591.2021.9370320

Pascuzzi, V. R., & Goli, M. (2022). Benchmarking a proof-of-Concept performance portable SYCL-based fast fourier transformation library. *International Workshop on OpenCL*, 1-9. https://doi.org/10.1145/3529538.3529996

Popov, S., Georgiev, I., Dimov, R., & Slusallek, P. (2009). Object partitioning considered harmful. *Proceedings of the 1st ACM Conference on High Performance Graphics - HPG '09*, 15–22. https://doi.org/10.1145/1572769.1572772

Qi, S., Wang, X., & Shi, S. (2011). Mixed precision method for GPU-based FFT. *2011 14th IEEE International Conference on Computational Science and Engineering* (pp. 580-586). https://doi.org/10.1109/cse.2011.103

Schutz, M., & Wimmer, M. (2019). Rendering point clouds with compute Shaders. *SIGGRAPH Asia 2019 Posters*, 1-2. https://doi.org/10.1145/3355056.3364554

Viitanen, T., Koskela, M., Jääskeläinen, P., & Takala, J. (2016). Multi bounding volume hierarchies for ray tracing pipelines. *SIGGRAPH ASIA 2016 Technical Briefs*, 1-4. https://doi.org/10.1145/3005358.3005384

Vinkler, M., Havran, V., & Bittner, J. (2014). Bounding volume hierarchies versus KD-trees on contemporary many-core architectures. *Proceedings of the 30th Spring Conference on Computer Graphics*, 29–36. https://doi.org/10.1145/2643188.2643196

Woo, A. (1990). Fast ray–box intersection. *Graphics Gems*, 395–396. https://doi.org/10.1016/b978-0-08-050753-8.50084-x

Wu, Z., Zhao, F., & Liu, X. (2011). Sah KD-tree construction on GPU. *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics - HPG '11*, 71–78. https://doi.org/10.1145/2018323.2018335

Xu, Y., Jiang, Y., Zhang, J., Li, K., & Geng, G. (2022). Real-time ray-traced soft shadows of environmental lighting by conical ray culling. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, *5*(1), 1-15. https://doi.org/10.1145/3522617

Ye, D., He, J., Hu, W., & Liu, J. (2018). Measurement and analysis on audio latency for multiple operating systems. *2018 13th IEEE Conference on Industrial Electronics and Applications (ICIEA)* (pp. 2496-2500). https://doi.org/10.1109/iciea.2018.8398130

6 APPENDIX

6.1 Code

6.1.1 Vulkan Instance Creation

```
#ifdef SE_DEBUG
        std::vector<const char*> validationLayers = { "VK_LAYER_KHRONOS_validation" };
#else
        std::vector<const char*> validationLayers = {};
#endif

        instance = VulkanInstance::Builder()
        .withApiVersion(1, 3)
        .withInstanceExtensions(window.getRequiredVulkanInstanceExtensions())
        .withValidationLayers(validationLayers)
#ifdef SE_DEBUG
        .withDebugUtilities()
#endif
        .build();
```

6.1.2 Logical Device Creation

Top level logical device creation

```
device = VulkanDevice::Builder(vulkanQueueBuilder)
    .withApiVersionSupport(1, 3)
    .withQueueFamilySupport(VulkanDeviceQueueFamilies::GRAPHICS |
VulkanDeviceQueueFamilies::PRESENTATION | VulkanDeviceQueueFamilies::TRANSFER |
VulkanDeviceQueueFamilies::COMPUTE)
    .withSurfacePresentationSupport(surface.get())
    .withInstance(instance.get())
    .withValidationLayers(validationLayers)
    .build();
```

Physical Device Selection

```
uint32_t physicalDeviceCount{ 0 };
vkEnumeratePhysicalDevices(instance->getInstance(), &physicalDeviceCount, nullptr);

std::vector<VkPhysicalDevice> physicalDevices(physicalDeviceCount);
vkEnumeratePhysicalDevices(instance->getInstance(), &physicalDeviceCount, physicalDevices.data());

VkPhysicalDevice chosenPhysicalDevice = VK_NULL_HANDLE;
for (const auto& physicalDevice : physicalDevices)
{
    if(!findDeviceQueueFamilyIndicies(physicalDevice, surface).areValid(requiredQueueFamilies) ||
         !checkPhysicalDeviceExtensionSupport(physicalDevice, requiredDeviceExtensions) ||
         !checkPhysicalDeviceSwapchainSupport(physicalDevice, surface))
    {
         continue;
    }

    chosenPhysicalDevice = physicalDevice;
}

if (chosenPhysicalDevice == VK_NULL_HANDLE)
{
```

```
    throw std::runtime_error("No supported physical device found");
}

return chosenPhysicalDevice;
```

Logical Device Creation

```
std::vector<VkDeviceQueueCreateInfo> queueCreateInfos{ };

std::set<int32_t> indicies{ };
if(queueFamilyIdicies.graphics >= 0) { indicies.insert(queueFamilyIdicies.graphics); }
if(queueFamilyIdicies.presentation >= 0) { indicies.insert(queueFamilyIdicies.presentation); }
if(queueFamilyIdicies.transfer >= 0) { indicies.insert(queueFamilyIdicies.transfer); }
if(queueFamilyIdicies.compute >= 0) { indicies.insert(queueFamilyIdicies.compute); }

for (uint32_t const &queueFamilyIndex : indicies)
{
    float priority = 1.0f;
    VkDeviceQueueCreateInfo queueCreateInfo = {};
    queueCreateInfo.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
    queueCreateInfo.queueFamilyIndex = queueFamilyIndex;
    queueCreateInfo.queueCount = 1;
    queueCreateInfo.pQueuePriorities = &priority;

    queueCreateInfos.push_back(queueCreateInfo);
}

VkPhysicalDeviceFeatures deviceFeatures{ };
deviceFeatures.samplerAnisotropy = VK_TRUE;

VkDeviceCreateInfo deviceCreateInfo{ };
deviceCreateInfo.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
deviceCreateInfo.pQueueCreateInfos = queueCreateInfos.data();
deviceCreateInfo.queueCreateInfoCount = static_cast<uint32_t>(queueCreateInfos.size());
deviceCreateInfo.enabledExtensionCount = static_cast<uint32_t>(requiredDeviceExtensions.size());
deviceCreateInfo.ppEnabledExtensionNames = requiredDeviceExtensions.data();
deviceCreateInfo.pEnabledFeatures = &deviceFeatures;

if (validationLayers.size() > 0) {
    deviceCreateInfo.enabledLayerCount = static_cast<uint32_t>(validationLayers.size());
    deviceCreateInfo.ppEnabledLayerNames = validationLayers.data();
}
else {
    deviceCreateInfo.enabledLayerCount = 0;
    deviceCreateInfo.ppEnabledLayerNames = nullptr;
}

VkDevice logicalDevice{ VK_NULL_HANDLE };
VkResult result = vkCreateDevice(physicalDevice, &deviceCreateInfo, nullptr, &logicalDevice);
if (result != VK_SUCCESS)
{
    throw std::runtime_error("Failed to create a logical device");
}

return logicalDevice;
```

6.1.3 Command Buffer Creation

```
computeCommandPool = VulkanCommandPool::Builder()
    .withDevice(device.get())
```

```
    .withFlags(VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT)
    .withQueueFamilyIndex(device->getComputeQueue()->getFamilyIndex())
    .build();


computeCommandBuffer = VulkanCommandBuffer::Builder()
    .withDevice(device.get())
    .withCommandPool(computeCommandPool.get())
    .withLevel(VK_COMMAND_BUFFER_LEVEL_PRIMARY)
    .build();
```

6.1.4 Storage Buffer Creation

Below, computGeometryVertexStagingBuffer, is an array of vertex data which is placed in a host visible transfer buffer, then copied to device local buffer using a command buffer submitted to a graphics/transfer queue.

```
VkDeviceSize bufferSize = sizeof(AudioComputeVertex3) * computGeometryVertexStagingBuffer.size();
computeGeometryVertexBuffer = VulkanBuffer::Builder()
    .withDevice(device.get())
    .withBufferSize(bufferSize)
    .withBufferUsageFlags(VK_BUFFER_USAGE_STORAGE_BUFFER_BIT | VK_BUFFER_USAGE_TRANSFER_DST_BIT)
    .withMemoryPropertyFlags(VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT)
    .build();

auto const stagingBuffer = VulkanBuffer::Builder()
    .withDevice(device.get())
    .withBufferSize(bufferSize)
    .withBufferUsageFlags(VK_BUFFER_USAGE_TRANSFER_SRC_BIT)
    .withMemoryPropertyFlags(VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
VK_MEMORY_PROPERTY_HOST_COHERENT_BIT)
    .build();

stagingBuffer->update(computGeometryVertexStagingBuffer.data(), bufferSize);
stagingBuffer->copyToBuffer(commandBuffer.get(), device->getGraphicsQueue().get(),
computeGeometryVertexBuffer.get(), bufferSize);
```

6.1.5 Descriptor Set Creation

```
computeDescriptorPool = VulkanDescriptorPool::Builder()
    .withDevice(device.get())
    .withMaxSets(2)
    .withPoolSize(VK_DESCRIPTOR_TYPE_STORAGE_BUFFER, 4)
    .build();

computeGeometryDescriptorSetLayout = VulkanDescriptorSetLayout::Builder()
    .withDevice(device.get())
    .withLayoutBinding(0, VK_DESCRIPTOR_TYPE_STORAGE_BUFFER, 1, VK_SHADER_STAGE_COMPUTE_BIT)
    .withLayoutBinding(1, VK_DESCRIPTOR_TYPE_STORAGE_BUFFER, 1, VK_SHADER_STAGE_COMPUTE_BIT)
    .withLayoutBinding(2, VK_DESCRIPTOR_TYPE_STORAGE_BUFFER, 1, VK_SHADER_STAGE_COMPUTE_BIT)
    .build();

computeGeometryDescriptorSet = VulkanDescriptorSet::Builder()
    .withDevice(device.get())
    .withDescriptorPool(computeDescriptorPool.get())
    .withDescriptorSetLayout(computeGeometryDescriptorSetLayout.get())
    .build();
```

6.1.6 Shader Module Creation

Reading Shader File Bytes

```
SDL_RWops * io = SDL_RWFromFile(filePath.c_str(), "rb");
if (io == nullptr) {
    throw std::runtime_error("Failed to open file");
}

size_t const objectsToRead = io->size(io);
size_t const objectSizeBytes = 1;
std::vector<char> content(objectsToRead);
if (io->read(io, content.data(), objectSizeBytes, objectsToRead) == 0) {
    throw std::runtime_error("Failed to read file content");
}
io->close(io);

return content;
```

Shader Module Creation

Below, shaderCode is an array of SPIR-V bytes

```
computeShaderModule = VulkanShaderModule::Builder()
    .withDevice(device.get())
    .withShaderCode(shaderCode)
    .build();
```

The contents of the above build() function

```
VkShaderModuleCreateInfo shaderModuleCreateInfo{ };
shaderModuleCreateInfo.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
shaderModuleCreateInfo.pCode = reinterpret_cast<const uint32_t*>(shaderCode.data());
shaderModuleCreateInfo.codeSize = shaderCode.size();

VkShaderModule shaderModule{ VK_NULL_HANDLE };
if (vkCreateShaderModule(device->getLogicalDevice(), &shaderModuleCreateInfo, nullptr,
&shaderModule) != VK_SUCCESS)
{
    throw std::runtime_error("Failed to create a shader module");
}

return std::make_unique<VulkanShaderModule>(shaderModule, device);
```

6.1.7 Compute Pipeline Creation

```
computePipeline = VulkanPipeline::ComputeBuilder()
    .withDevice(device.get())
    .withDescriptorSetLayouts({ computeGeometryDescriptorSetLayout.get(),
computeOutputDescriptorSetLayout.get() })
    .withPushConstantRange(VK_SHADER_STAGE_COMPUTE_BIT, 0, sizeof(AudioComputeInput))
    .withShaderModule(std::move(VulkanShaderModule::Builder()
         .withDevice(device.get())
         .withShaderCode(shaderCode)
         .build()))
    .build();
```

The contents of the final build() function above

```
std::vector<VkDescriptorSetLayout> descriptorSetLayoutHandles{ };
```

```
for (auto const& descriptorSetLayout : descriptorSetLayouts)
{
    descriptorSetLayoutHandles.push_back(descriptorSetLayout->getDescriptorSetLayout());
}

VkPipelineLayoutCreateInfo pipelineLayoutCreateInfo{ };
pipelineLayoutCreateInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
pipelineLayoutCreateInfo.pSetLayouts = descriptorSetLayoutHandles.data();
pipelineLayoutCreateInfo.setLayoutCount = static_cast<uint32_t>(descriptorSetLayoutHandles.size());
pipelineLayoutCreateInfo.pushConstantRangeCount = static_cast<uint32_t>(pushConstantRanges.size());
pipelineLayoutCreateInfo.pPushConstantRanges = pushConstantRanges.data();

VkPipelineLayout pipelineLayout{ VK_NULL_HANDLE };
if (vkCreatePipelineLayout(device->getLogicalDevice(), &pipelineLayoutCreateInfo, nullptr,
&pipelineLayout) != VK_SUCCESS)
{
    throw std::runtime_error("Failed to create pipeline layout");
}

VkPipelineShaderStageCreateInfo shaderStageCreateInfo = {};
shaderStageCreateInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
shaderStageCreateInfo.stage = VK_SHADER_STAGE_COMPUTE_BIT;
shaderStageCreateInfo.module = shaderModule->getShaderModule();
shaderStageCreateInfo.pName = "main";

VkComputePipelineCreateInfo pipelineCreateInfo = {};
pipelineCreateInfo.sType = VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO;
pipelineCreateInfo.layout = pipelineLayout;
pipelineCreateInfo.stage = shaderStageCreateInfo;

VkPipeline pipeline{ VK_NULL_HANDLE };
if (vkCreateComputePipelines(device->getLogicalDevice(), VK_NULL_HANDLE, 1, &pipelineCreateInfo,
nullptr, &pipeline) != VK_SUCCESS)
{
    throw std::runtime_error("Failed to create pipeline");
}

return std::make_unique<VulkanPipeline>(pipeline, pipelineLayout, device);
```

6.1.8 Fence Creation and Queue Submission

Fence Creation

```
computeFence = VulkanFence::Builder()
    .withDevice(device.get())
    .build();
```

The content of the build() function above

```
VkFenceCreateInfo fenceCreateInfo{ };
fenceCreateInfo.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
fenceCreateInfo.flags = VK_FENCE_CREATE_SIGNALED_BIT;

VkFence fence{ VK_NULL_HANDLE };
if (vkCreateFence(device->getLogicalDevice(), &fenceCreateInfo, nullptr, &fence) != VK_SUCCESS)
{
    throw std::runtime_error("Failed to create fence");
}

return std::make_unique<VulkanFence>(fence, device);
```

Fence Queue Submission

```
VkCommandBuffer commandBufferHandle = commandBuffer->getCommandBuffer();
VkFence fenceHandle = fence->getFence();

VkSubmitInfo submitInfo{ };
submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
submitInfo.pCommandBuffers = &commandBufferHandle;
submitInfo.commandBufferCount = 1;

if (vkQueueSubmit(queue, 1, &submitInfo, fenceHandle) != VK_SUCCESS)
{
    throw std::runtime_error("Failed to submit command buffer to queue");
}
```

6.1.9 Compute Shader Execution

```
// Begin recording command buffer commands
computeCommandBuffer->begin(VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT);
computePipeline->bind(computeCommandBuffer.get(), VK_PIPELINE_BIND_POINT_COMPUTE);

vkCmdPushConstants(computeCommandBuffer->getCommandBuffer(), computePipeline->getPipelineLayout(),
VK_SHADER_STAGE_COMPUTE_BIT, 0, sizeof(AudioComputeInput), &audioComputeInput);

std::vector<VkDescriptorSet> const descriptorSetGroup = {
    computeGeometryDescriptorSet->getDescriptorSet(),
    computeOutputDescriptorSet->getDescriptorSet()
};
vkCmdBindDescriptorSets(
    computeCommandBuffer->getCommandBuffer(),
    VK_PIPELINE_BIND_POINT_COMPUTE,
    computePipeline->getPipelineLayout(),
    0,
    descriptorSetGroup.size(),
    descriptorSetGroup.data(),
    0,
    nullptr);

vkCmdDispatch(computeCommandBuffer->getCommandBuffer(), GPU_AUDIO_THREAD_COUNT, 1, 1);
computeCommandBuffer->end();

// Signal fence
VkFence fenceHandle = computeFence->getFence();
vkResetFences(device->getLogicalDevice(), 1, &fenceHandle);

// Submit command buffer to the compute queue, wait for fence to unsignal meaning compute shader
execution has completed
device->getComputeQueue()->submit(nullptr, nullptr, computeFence.get(), computeCommandBuffer.get(),
VK_PIPELINE_STAGE_NONE);
vkWaitForFences(device->getLogicalDevice(), 1, &fenceHandle, VK_TRUE,
std::numeric_limits<uint64_t>::max());
```

6.1.10 Base Shader Input Buffer Setup

```
AudioComputeGeometryMetadata computeGeometryMetadata{ };
computeGeometryMetadata.vertexCount = computGeometryVertexStagingBuffer.size();
computeGeometryMetadata.indexCount = computGeometryIndexStagingBuffer.size();

auto const commandBuffer = VulkanCommandBuffer::Builder()
        .withDevice(device.get())
```

```
        .withCommandPool(graphicsCommandPool.get())
        .withLevel(VK_COMMAND_BUFFER_LEVEL_PRIMARY)
        .build();

// Copy vertex data to device local buffer, link to a descriptor set
{
        VkDeviceSize bufferSize = sizeof(AudioComputeVertex3) *
computGeometryVertexStagingBuffer.size();
        computeGeometryVertexBuffer = VulkanBuffer::Builder()
        .withDevice(device.get())
        .withBufferSize(bufferSize)
        .withBufferUsageFlags(VK_BUFFER_USAGE_STORAGE_BUFFER_BIT |
VK_BUFFER_USAGE_TRANSFER_DST_BIT)
        .withMemoryPropertyFlags(VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT)
        .build();

        auto const stagingBuffer = VulkanBuffer::Builder()
        .withDevice(device.get())
        .withBufferSize(bufferSize)
        .withBufferUsageFlags(VK_BUFFER_USAGE_TRANSFER_SRC_BIT)
        .withMemoryPropertyFlags(VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
VK_MEMORY_PROPERTY_HOST_COHERENT_BIT)
        .build();

        stagingBuffer->update(computGeometryVertexStagingBuffer.data(), bufferSize);
        stagingBuffer->copyToBuffer(commandBuffer.get(), device->getGraphicsQueue().get(),
computeGeometryVertexBuffer.get(), bufferSize);

        computeGeometryDescriptorSet->updateFromBuffer(computeGeometryVertexBuffer.get(), 0,
bufferSize, 0, VK_DESCRIPTOR_TYPE_STORAGE_BUFFER, device.get());

        computGeometryVertexStagingBuffer.clear();
}

// Copy index data to device local buffer, link to a descriptor set
{
        VkDeviceSize bufferSize = sizeof(std::uint32_t) * computGeometryIndexStagingBuffer.size();
        computeGeometryIndexBuffer = VulkanBuffer::Builder()
        .withDevice(device.get())
        .withBufferSize(bufferSize)
        .withBufferUsageFlags(VK_BUFFER_USAGE_STORAGE_BUFFER_BIT |
VK_BUFFER_USAGE_TRANSFER_DST_BIT)
        .withMemoryPropertyFlags(VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT)
        .build();

        auto const stagingBuffer = VulkanBuffer::Builder()
        .withDevice(device.get())
        .withBufferSize(bufferSize)
        .withBufferUsageFlags(VK_BUFFER_USAGE_TRANSFER_SRC_BIT)
        .withMemoryPropertyFlags(VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
VK_MEMORY_PROPERTY_HOST_COHERENT_BIT)
        .build();

        stagingBuffer->update(computGeometryIndexStagingBuffer.data(), bufferSize);
        stagingBuffer->copyToBuffer(commandBuffer.get(), device->getGraphicsQueue().get(),
computeGeometryIndexBuffer.get(), bufferSize);

        computeGeometryDescriptorSet->updateFromBuffer(computeGeometryIndexBuffer.get(), 0,
bufferSize, 1, VK_DESCRIPTOR_TYPE_STORAGE_BUFFER, device.get());

        computGeometryIndexStagingBuffer.clear();
}
```

```
// Copy metadata data to device local buffer, link to a descriptor set
{
        VkDeviceSize bufferSize = sizeof(AudioComputeGeometryMetadata);
        computeGeometryMetadataBuffer = VulkanBuffer::Builder()
        .withDevice(device.get())
        .withBufferSize(bufferSize)
        .withBufferUsageFlags(VK_BUFFER_USAGE_STORAGE_BUFFER_BIT |
VK_BUFFER_USAGE_TRANSFER_DST_BIT)
        .withMemoryPropertyFlags(VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT)
        .build();

        auto const stagingBuffer = VulkanBuffer::Builder()
        .withDevice(device.get())
        .withBufferSize(bufferSize)
        .withBufferUsageFlags(VK_BUFFER_USAGE_TRANSFER_SRC_BIT)
        .withMemoryPropertyFlags(VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
VK_MEMORY_PROPERTY_HOST_COHERENT_BIT)
        .build();

        stagingBuffer->update(&computeGeometryMetadata, bufferSize);
        stagingBuffer->copyToBuffer(commandBuffer.get(), device->getGraphicsQueue().get(),
computeGeometryMetadataBuffer.get(), bufferSize);

        computeGeometryDescriptorSet->updateFromBuffer(computeGeometryMetadataBuffer.get(), 0,
bufferSize, 2, VK_DESCRIPTOR_TYPE_STORAGE_BUFFER, device.get());

        computeGeometryMetadata.vertexCount = 0;
        computeGeometryMetadata.indexCount = 0;
}

// Create output buffer, link to a descriptor set
const uint32_t outputBufferSize = GPU_AUDIO_THREAD_COUNT * sizeof(AudioComputeOutput);
computeOutBuffer = VulkanBuffer::Builder()
        .withDevice(device.get())
        .withBufferSize(outputBufferSize)
        .withBufferUsageFlags(VK_BUFFER_USAGE_STORAGE_BUFFER_BIT)
        .withMemoryPropertyFlags(VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
VK_MEMORY_PROPERTY_HOST_COHERENT_BIT)
        .build();

computeOutputDescriptorSet->updateFromBuffer(computeOutBuffer.get(), 0, outputBufferSize, 0,
VK_DESCRIPTOR_TYPE_STORAGE_BUFFER, device.get());
```

6.1.11 Base Shader GLSL Code

```
#version 450


////////////////////////////////////
////////////// Defines /////////////
////////////////////////////////////
#define FLT_MAX 3.402823466e+38
#define FLT_MIN 1.175494351e-38
#define FLT_EPSILON 1.192092896e-07


////////////////////////////////////
////////////// Constants ///////////
////////////////////////////////////
const int MAX_RAYS = 500;
const int AUDIO_DIRECTION_COUNT = 256;
```

```
////////////////////////////////////
/////////////// Structs //////////////
////////////////////////////////////
struct Ray
{
        vec3 origin;
        vec3 direction;
        float totalDistance;
};

struct RayQueue
{
        int current;
        int end;
        Ray rays[MAX_RAYS];
};

struct AudioComputeVertex3
{
        vec3 position;
        vec3 normal;
};

struct AudioComputeGeometryMetadata
{
        int vertexCount;
        int indexCount;
};

struct Intersection
{
        bool isFound;
        bool isDestination;
        float t;
        vec3 normal;
};

struct AudioComputeInput
{
        vec3 audioSourcePosition;
        vec3 listenerPosition;
};

struct AudioComputeOutput
{
        bool intersectionFound;
        Ray finalRay;

// Debug output
        float intersectionT;
        vec3 intersectionNormal;
        vec3 audioSourcePosition;
        vec3 audioDirection;
        int successfulGeometryIntersectionTests;
        int totalGeometryIntersectionTests;
        Ray rays[MAX_RAYS];
};


////////////////////////////////////
////////// Shader Inputs //////////
```

```
////////////////////////////////////
layout(push_constant) uniform PushModel {
	AudioComputeInput audioComputeInput;
} pushConstants;

layout(set = 0, binding = 0) buffer GeometryVertexBuffer
{
	AudioComputeVertex3 data[];
} geometryVertexBuffer;

layout(set = 0, binding = 1) buffer GeometryIndexBuffer
{
	int data[];
} geometryIndexBuffer;

layout(set = 0, binding = 2) buffer GeometryMetadataBuffer
{
	AudioComputeGeometryMetadata data;
} geometryMetadataBuffer;

layout(set = 1, binding = 0) buffer OutputBuffer
{
	AudioComputeOutput data[];
} outputBuffer;


////////////////////////////////////
////////////// Functions //////////////
////////////////////////////////////

/**
 * @brief Initialize an intersection with default values
 *
 * @returns The intersection
 */
Intersection defaultIntersection()
{
	Intersection intersection;
	intersection.isFound = false;
	intersection.isDestination = false;
	intersection.t = FLT_MAX;
	intersection.normal = vec3(0.0);

	return intersection;
}

/**
 * @brief Test for a ray-triangle intersection using the Möller-Trumbore algorithm
 * @see https://en.wikipedia.org/wiki/M%C3%B6ller%E2%80%93Trumbore_intersection_algorithm
 *
 * @param ray The ray to test
 * @param vertex0 The first vertex describing the triangle
 * @param vertex1 The second vertex describing the triangle
 * @param vertex2 The third vertex describing the triangle
 * @param minT The minimum distance along the ray which can intersect
 * @param maxT The maximum distance along the ray which can intersect
 *
 * @returns The intersection, if found
 */
Intersection testRayTriangleIntersection(Ray ray, vec3 vertex0, vec3 vertex1, vec3 vertex2, float
minT, float maxT)
{
	uint gID = gl_GlobalInvocationID.x;
```

```
        outputBuffer.data[gID].totalGeometryIntersectionTests++;

        Intersection intersection = defaultIntersection();

        vec3 edge1 = vec3(0.0);
        vec3 edge2 = vec3(0.0);
        vec3 h = vec3(0.0);
        vec3 s = vec3(0.0);
        vec3 q = vec3(0.0);

        float a = 0.0;
        float f = 0.0;
        float u = 0.0;
        float v = 0.0;

        edge1 = vertex1 - vertex0;
        edge2 = vertex2 - vertex0;

        h = cross(ray.direction, edge2);
        a = dot(edge1, h);
        if (a > -FLT_EPSILON && a < FLT_EPSILON)
        {
        return intersection;
        }

        f = 1.0f / a;
        s = ray.origin - vertex0;
        u = f * dot(s, h);
        if (u < 0.0f || u > 1.0f)
        {
        return intersection;
        }

        q = cross(s, edge1);
        v = f * dot(ray.direction, q);
        if (v < 0.0f || u + v > 1.0f)
        {
        return intersection;
        }

        float t = f * dot(edge2, q);
        if (t > FLT_EPSILON && t > minT && t < maxT)
        {
        outputBuffer.data[gID].successfulGeometryIntersectionTests++;

        intersection.t = t;
        intersection.normal = normalize(cross(edge1, edge2));
        intersection.isFound = true;
        return intersection;
        }

        return intersection;
}

/**
 * @brief Test for a ray-sphere intersection using the quadratic formula
 *
 * @param ray The ray to test
 * @param spherePosition The center point of the sphere
 * @param radius The sphere radius
 * @param minT The minimum distance along the ray which can intersect
 * @param maxT The maximum distance along the ray which can intersect
 *
```

```
 * @returns The intersection, if found
 */
Intersection testRaySphereIntersection(Ray ray, vec3 spherePosition, float sphereRadius, float minT,
float maxT)
{
        Intersection intersection = defaultIntersection();

        float a = dot(ray.direction, ray.direction);
        float b = dot(ray.origin - spherePosition, ray.direction) * 2;
        float c = dot(ray.origin - spherePosition, ray.origin - spherePosition) - (sphereRadius *
sphereRadius);

        float discriminant = (b * b) - (4.0f * a * c);
        if (discriminant < 0.0f)
        {
        return intersection;
        }

        float intersectionA = (-b + sqrt(discriminant)) / (2.0f * a);
        float intersectionB = (-b - sqrt(discriminant)) / (2.0f * a);
        if ((intersectionA > minT && intersectionA < maxT) ||
        (intersectionB > minT && intersectionB < maxT))
        {
        intersection.t = min(intersectionA, intersectionB);
        intersection.normal = normalize((ray.origin + ray.direction * intersection.t) -
spherePosition);
        intersection.isFound = true;
        return intersection;
        }

        return intersection;
}

/**
 * @brief Reflects a ray around a given normal
 *
 * @param rayDirection The direction of the ray to reflect
 * @param normal The normal to reflect around
 *
 * @returns The reflected ray direction
 */
vec3 reflectRay(vec3 rayDirection, vec3 normal)
{
        normal = normalize(normal);
        rayDirection = normalize(rayDirection);

        mat3 rayReversalMatrix;
        rayReversalMatrix[0] = vec3(-1.0, 0.0, 0.0);
        rayReversalMatrix[1] = vec3(0.0, -1.0, 0.0);
        rayReversalMatrix[2] = vec3(0.0, 0.0, -1.0);

        vec3 raySourceDirection = rayReversalMatrix * rayDirection;
        return (normal * dot(normal, raySourceDirection) * 2) - raySourceDirection;
}

/**
 * @brief Find the closest intersection for a ray
 *
 * @param ray The ray to test
 * @param minT The minimum distance along the ray which can intersect
 * @param maxT The maximum distance along the ray which can intersect
 *
 * @returns The closest intersection, if found
```

```
 */
Intersection findClosestIntersection(Ray ray, float minT, float maxT)
{
        Intersection closestIntersection = defaultIntersection();

        float sphereRadius = 1.0;
        Intersection sphereIntersection = testRaySphereIntersection(
        ray,
        pushConstants.audioComputeInput.listenerPosition,
        sphereRadius,
        minT,
        maxT);

        if(sphereIntersection.t < closestIntersection.t)
        {
        closestIntersection.isFound = true;
        closestIntersection.isDestination = true;
        closestIntersection.t = sphereIntersection.t;
        closestIntersection.normal = sphereIntersection.normal;
        }

        for(int i = 0; i < geometryMetadataBuffer.data.indexCount; i+=3)
        {
        Intersection intersection = testRayTriangleIntersection(
        ray,
        geometryVertexBuffer.data[geometryIndexBuffer.data[i + 0]].position,
        geometryVertexBuffer.data[geometryIndexBuffer.data[i + 1]].position,
        geometryVertexBuffer.data[geometryIndexBuffer.data[i + 2]].position,
        minT,
        maxT);

        if(intersection.t < closestIntersection.t)
        {
        closestIntersection.isFound = true;
        closestIntersection.isDestination = false;
        closestIntersection.t = intersection.t;
        closestIntersection.normal = intersection.normal;
        }
        else if(intersection.t == closestIntersection.t)
        {
        closestIntersection.normal = (intersection.normal + closestIntersection.normal) / 2;
        }
        }

        return closestIntersection;
}

/**
 * @brief Shader entry point
 */
void main()
{
        uint gID = gl_GlobalInvocationID.x;
        outputBuffer.data[gID].successfulGeometryIntersectionTests = 0;
        outputBuffer.data[gID].totalGeometryIntersectionTests = 0;

        float angle = radians(360.0 / AUDIO_DIRECTION_COUNT);
        vec3 direction = vec3(0.0, 0.0, 1.0);
        for(int i = 0; i < gID; ++i)
        {
        direction = vec3(
        (direction.x * cos(angle)) - (direction.z * sin(angle)),
        0.0,
```

```
        (direction.z * cos(angle)) + (direction.x * sin(angle))
        );
        }

        Ray initialRay;
        initialRay.origin = pushConstants.audioComputeInput.audioSourcePosition;
        initialRay.direction = direction;
        initialRay.totalDistance = 0.0;

        RayQueue rayQueue;
        rayQueue.rays[0] = initialRay;
        rayQueue.current = 0;
        rayQueue.end = 1;

        Intersection intersection = defaultIntersection();

        // For each ray in the queue of rays to process
        while(rayQueue.current < rayQueue.end)
        {
        // Find the closest intersection between the ray and the environment
        intersection = findClosestIntersection(
        rayQueue.rays[rayQueue.current],
        FLT_MIN,
        FLT_MAX);

        if(intersection.isFound)
        {
        if(!intersection.isDestination && rayQueue.end < MAX_RAYS)
        {
                // If the ray has not reached it's destination, and the maximum number of rays has
not been computed,
                // reflect the ray around the intersection normal and add the reflected ray to the
queue of rays to process
                Ray reflectedRay;
                reflectedRay.origin = rayQueue.rays[rayQueue.current].origin +
(rayQueue.rays[rayQueue.current].direction * intersection.t);
                reflectedRay.direction = reflectRay(rayQueue.rays[rayQueue.current].direction,
intersection.normal);
                reflectedRay.totalDistance = rayQueue.rays[rayQueue.current].totalDistance +
intersection.t;

                // Move the reflected ray origin along the intersection normal to avoid incorrectly
intersecting the same object
                reflectedRay.origin = reflectedRay.origin + (intersection.normal * 0.0001);

                rayQueue.rays[rayQueue.end] = reflectedRay;
                rayQueue.end++;
        }
        else if(intersection.isDestination)
        {
                // Destination found
                break;
        }
        }

        ++rayQueue.current;
        }

        outputBuffer.data[gID].intersectionFound = intersection.isDestination;
        outputBuffer.data[gID].finalRay = rayQueue.rays[rayQueue.current];

        //////////////////////
        // Debugging Output //
```

```
        //////////////////////
        if(rayQueue.current == rayQueue.end)
        {
        --rayQueue.current;
        }

        outputBuffer.data[gID].intersectionT = intersection.t;
        outputBuffer.data[gID].intersectionNormal = intersection.normal;
        outputBuffer.data[gID].audioSourcePosition =
pushConstants.audioComputeInput.audioSourcePosition;
        outputBuffer.data[gID].audioDirection = direction;

        for(int i = 0; i < MAX_RAYS; ++i)
        {
        outputBuffer.data[gID].rays[i] = rayQueue.rays[i];
        }
}
```

6.1.12 3D Space Partitioned Shader GLSL Code

```
#version 450


////////////////////////////////////
////////////// Defines /////////////
////////////////////////////////////
#define FLT_MAX 3.402823466e+38
#define FLT_MIN 1.175494351e-38
#define FLT_EPSILON 1.192092896e-07
//#define DEBUG_OUTPUT


////////////////////////////////////
////////////// Constants ///////////
////////////////////////////////////
const int MAX_RAYS = 500;
const int AUDIO_DIRECTION_COUNT = 1024;
const int INVALID_INDEX = 4294967295;


////////////////////////////////////
////////////// Structs /////////////
////////////////////////////////////
struct Ray
{
        vec3 origin;
        vec3 direction;
        float totalDistance;
};

struct RayQueue
{
        int current;
        int end;
        Ray rays[MAX_RAYS];
};

struct AudioComputeVertex3
{
        vec3 position;
        vec3 normal;
```

```
};

struct AudioComputeGeometryMetadata
{
        int vertexCount;
        int indexCount;
};

struct Intersection
{
        bool isFound;
        bool isDestination;
        float t;
        vec3 normal;
};

struct AudioComputeInput
{
        vec3 audioSourcePosition;
        vec3 listenerPosition;
};

struct AudioComputeOutput
{
        bool intersectionFound;
        float intersectionT;
        vec3 intersectionNormal;
        vec3 audioSourcePosition;
        vec3 audioDirection;
        Ray finalRay;
        int successfulGeometryIntersectionTests;
        int totalGeometryIntersectionTests;
        int successfulAabbIntersectionTests;
        int totalAabbIntersectionTests;
        Ray rays[MAX_RAYS];
};

struct AABB
{
        vec3 position;
        vec3 extents;
};

struct KdNode
{
        AABB aabb;
        int nearChild;
        int farChild;
        int indicesStart;
        int indicesEnd;
};


////////////////////////////////////
////////// Shader Inputs //////////
////////////////////////////////////
layout(push_constant) uniform PushModel {
        AudioComputeInput audioComputeInput;
} pushConstants;

layout(set = 0, binding = 0) buffer GeometryVertexBuffer
{
        AudioComputeVertex3 data[];
```

```
} geometryVertexBuffer;

layout(set = 0, binding = 1) buffer GeometryIndexBuffer
{
        int data[];
} geometryIndexBuffer;

layout(set = 0, binding = 2) buffer GeometryMetadataBuffer
{
        AudioComputeGeometryMetadata data;
} geometryMetadataBuffer;

layout(set = 0, binding = 3) buffer GeometryKdTreeNodeBuffer
{
        KdNode data[];
} geometryKdTreeNodeBuffer;

layout(set = 0, binding = 4) buffer GeometryKdTreeLeafNodeIndexBuffer
{
        int data[];
} geometryKdTreeLeafNodeIndexBuffer;

layout(set = 1, binding = 0) buffer OutputBuffer
{
        AudioComputeOutput data[];
} outputBuffer;


////////////////////////////////////
/////////// Functions //////////////
////////////////////////////////////

/**
 * @brief Initialize an intersection with default values
 *
 * @returns The intersection
 */
Intersection defaultIntersection()
{
        Intersection intersection;
        intersection.isFound = false;
        intersection.isDestination = false;
        intersection.t = FLT_MAX;
        intersection.normal = vec3(0.0);

        return intersection;
}

/**
 * @brief Test for a ray-AABB intersection using the Fast Ray-Box Intersection
 * @see Woo (1990)
 * @see Graphics Gems, 1990, pp. 395-396
 *
 * @param ray The ray to test
 * @param aabb The axis aligned bounding box to test
 *
 * @returns True if an intersection is found, otherwise false
 */
bool testRayAabbIntersection(Ray ray, AABB aabb)
{
        #ifdef DEBUG_OUTPUT
        uint gID = gl_GlobalInvocationID.x;
        outputBuffer.data[gID].totalAabbIntersectionTests++;
```

```
#endif

bool xInside = true;
bool yInside = true;
bool zInside = true;

vec3 candidatePlane = vec3(0.0, 0.0, 0.0);
vec3 maxT = vec3(0.0, 0.0, 0.0);

vec3 boxMin = vec3(
aabb.position.x - aabb.extents.x,
aabb.position.y - aabb.extents.y,
aabb.position.z - aabb.extents.z
);
vec3 boxMax = vec3(
aabb.position.x + aabb.extents.x,
aabb.position.y + aabb.extents.y,
aabb.position.z + aabb.extents.z
);

// Find candidate planes, determine if the ray origin is inside the AABB
if (ray.origin.x < boxMin.x)
{
candidatePlane.x = boxMin.x;
xInside = false;
}
else if (ray.origin.x > boxMax.x)
{
candidatePlane.x = boxMax.x;
xInside = false;
}

if (ray.origin.y < boxMin.y)
{
candidatePlane.y = boxMin.y;
yInside = false;
}
else if (ray.origin.y > boxMax.y)
{
candidatePlane.y = boxMax.y;
yInside = false;
}

if (ray.origin.z < boxMin.z)
{
candidatePlane.z = boxMin.z;
zInside = false;
}
else if (ray.origin.z > boxMax.z)
{
candidatePlane.z = boxMax.z;
zInside = false;
}

if (xInside && yInside && zInside) {
#ifdef DEBUG_OUTPUT
outputBuffer.data[gID].successfulAabbIntersectionTests++;
#endif
return true;
}

// Find candidate plane T values
maxT.x = !xInside && ray.direction.x != 0.0
```

```
        ? (candidatePlane.x - ray.origin.x) / ray.direction.x
        : -1.0;

        maxT.y = !yInside && ray.direction.y != 0.0
        ? (candidatePlane.y - ray.origin.y) / ray.direction.y
        : -1.0;

        maxT.z = !zInside && ray.direction.z != 0.0
        ? (candidatePlane.z - ray.origin.z) / ray.direction.z
        : -1.0;

        // For the largest T value, confirm the intersection
        if (max(maxT.x, max(maxT.y, maxT.z)) < 0.0)
        {
        return false;
        }

        if (maxT.x > maxT.y && maxT.x > maxT.z)
        {
        // Check Y & Z
        float y = ray.origin.y + maxT.x * ray.direction.y;
        float z = ray.origin.z + maxT.x * ray.direction.z;
        if ((y < boxMin.y || y > boxMax.y) ||
        (z < boxMin.z || z > boxMax.z))
        {
        return false;
        }
        }
        else if (maxT.y > maxT.x && maxT.y > maxT.z)
        {
        // Check X & Z
        float x = ray.origin.x + maxT.y * ray.direction.x;
        float z = ray.origin.z + maxT.y * ray.direction.z;
        if ((x < boxMin.x || x > boxMax.x) ||
        (z < boxMin.z || z > boxMax.z))
        {
        return false;
        }
        }
        else if (maxT.z > maxT.x && maxT.z > maxT.y)
        {
        // Check X & Y
        float x = ray.origin.x + maxT.z * ray.direction.x;
        float y = ray.origin.y + maxT.z * ray.direction.y;
        if ((x < boxMin.x || x > boxMax.x) ||
        (y < boxMin.y || y > boxMax.y))
        {
        return false;
        }
        }

        #ifdef DEBUG_OUTPUT
        outputBuffer.data[gID].successfulAabbIntersectionTests++;
        #endif
        return true;
}

/**
 * @brief Test for a ray-triangle intersection using the Möller-Trumbore algorithm
 * @see Möller and Trumbore (1997)
 * @see https://en.wikipedia.org/wiki/M%C3%B6ller%E2%80%93Trumbore_intersection_algorithm
 *
 * @param ray The ray to test
```

```
 * @param vertex0 The first vertex describing the triangle
 * @param vertex1 The second vertex describing the triangle
 * @param vertex2 The third vertex describing the triangle
 * @param minT The minimum distance along the ray which can intersect
 * @param maxT The maximum distance along the ray which can intersect
 *
 * @returns The intersection, if found
 */
Intersection testRayTriangleIntersection(Ray ray, vec3 vertex0, vec3 vertex1, vec3 vertex2, float
minT, float maxT)
{
        #ifdef DEBUG_OUTPUT
        uint gID = gl_GlobalInvocationID.x;
        outputBuffer.data[gID].totalGeometryIntersectionTests++;
        #endif

        Intersection intersection = defaultIntersection();

        vec3 edge1 = vec3(0.0);
        vec3 edge2 = vec3(0.0);
        vec3 h = vec3(0.0);
        vec3 s = vec3(0.0);
        vec3 q = vec3(0.0);

        float a = 0.0;
        float f = 0.0;
        float u = 0.0;
        float v = 0.0;

        edge1 = vertex1 - vertex0;
        edge2 = vertex2 - vertex0;

        h = cross(ray.direction, edge2);
        a = dot(edge1, h);
        if (a > -FLT_EPSILON && a < FLT_EPSILON)
        {
        return intersection;
        }

        f = 1.0f / a;
        s = ray.origin - vertex0;
        u = f * dot(s, h);
        if (u < 0.0f || u > 1.0f)
        {
        return intersection;
        }

        q = cross(s, edge1);
        v = f * dot(ray.direction, q);
        if (v < 0.0f || u + v > 1.0f)
        {
        return intersection;
        }

        float t = f * dot(edge2, q);
        if (t > FLT_EPSILON && t > minT && t < maxT)
        {
        #ifdef DEBUG_OUTPUT
        outputBuffer.data[gID].successfulGeometryIntersectionTests++;
        #endif

        intersection.t = t;
        intersection.normal = normalize(cross(edge1, edge2));
```

```
        intersection.isFound = true;
        return intersection;
        }

        return intersection;
}

/**
 * @brief Test for a ray-sphere intersection using the quadratic formula
 *
 * @param ray The ray to test
 * @param spherePosition The center point of the sphere
 * @param radius The sphere radius
 * @param minT The minimum distance along the ray which can intersect
 * @param maxT The maximum distance along the ray which can intersect
 *
 * @returns The intersection, if found
 */
Intersection testRaySphereIntersection(Ray ray, vec3 spherePosition, float sphereRadius, float minT,
float maxT)
{
        Intersection intersection = defaultIntersection();

        float a = dot(ray.direction, ray.direction);
        float b = dot(ray.origin - spherePosition, ray.direction) * 2;
        float c = dot(ray.origin - spherePosition, ray.origin - spherePosition) - (sphereRadius *
sphereRadius);

        float discriminant = (b * b) - (4.0f * a * c);
        if (discriminant < 0.0f)
        {
        return intersection;
        }

        float intersectionA = (-b + sqrt(discriminant)) / (2.0f * a);
        float intersectionB = (-b - sqrt(discriminant)) / (2.0f * a);
        if ((intersectionA > minT && intersectionA < maxT) ||
        (intersectionB > minT && intersectionB < maxT))
        {
        intersection.t = min(intersectionA, intersectionB);
        intersection.normal = normalize((ray.origin + ray.direction * intersection.t) -
spherePosition);
        intersection.isFound = true;
        return intersection;
        }

        return intersection;
}

/**
 * @brief Reflects a ray around a given normal
 *
 * @param rayDirection The direction of the ray to reflect
 * @param normal The normal to reflect around
 *
 * @returns The reflected ray direction
 */
vec3 reflectRay(vec3 rayDirection, vec3 normal)
{
        normal = normalize(normal);
        rayDirection = normalize(rayDirection);

        mat3 rayReversalMatrix;
```

```
        rayReversalMatrix[0] = vec3(-1.0, 0.0, 0.0);
        rayReversalMatrix[1] = vec3(0.0, -1.0, 0.0);
        rayReversalMatrix[2] = vec3(0.0, 0.0, -1.0);

        vec3 raySourceDirection = rayReversalMatrix * rayDirection;
        return (normal * dot(normal, raySourceDirection) * 2) - raySourceDirection;
}

/**
 * @brief Find the closest intersection for a ray
 *
 * @param ray The ray to test
 * @param minT The minimum distance along the ray which can intersect
 * @param maxT The maximum distance along the ray which can intersect
 *
 * @returns The closest intersection, if found
 */
Intersection findClosestIntersection(Ray ray, float minT, float maxT)
{
        Intersection closestIntersection = defaultIntersection();

        float sphereRadius = 1.0;
        Intersection sphereIntersection = testRaySphereIntersection(
        ray,
        pushConstants.audioComputeInput.listenerPosition,
        sphereRadius,
        minT,
        maxT);

        if(sphereIntersection.t < closestIntersection.t)
        {
        closestIntersection.isFound = true;
        closestIntersection.isDestination = true;
        closestIntersection.t = sphereIntersection.t;
        closestIntersection.normal = sphereIntersection.normal;
        }

        int nodeStack[50];
        int nodeStackSize = 1;
        nodeStack[0] = 0;

        while(nodeStackSize > 0)
        {
        // Pop top KD tree node off of the stack
        KdNode currentNode = geometryKdTreeNodeBuffer.data[nodeStack[nodeStackSize-1]];
        nodeStackSize--;

        // Test AABB intersection
        bool aabbIntersection = testRayAabbIntersection(ray, currentNode.aabb);
        if(aabbIntersection)
        {
        // If inner node, add children to the stack of nodes to process
        if(currentNode.nearChild != INVALID_INDEX && currentNode.farChild != INVALID_INDEX)
        {
                nodeStack[nodeStackSize] = currentNode.farChild;
                nodeStackSize++;
                nodeStack[nodeStackSize] = currentNode.nearChild;
                nodeStackSize++;
        }
        // Leaf node, test for geometry intersection
        else
        {
                for(int i = currentNode.indicesStart; i < currentNode.indicesEnd; ++i)
```

```
                {
                Intersection intersection = testRayTriangleIntersection(
                ray,

geometryVertexBuffer.data[geometryIndexBuffer.data[geometryKdTreeLeafNodeIndexBuffer.data[i] +
0]].position,

geometryVertexBuffer.data[geometryIndexBuffer.data[geometryKdTreeLeafNodeIndexBuffer.data[i] +
1]].position,

geometryVertexBuffer.data[geometryIndexBuffer.data[geometryKdTreeLeafNodeIndexBuffer.data[i] +
2]].position,
                minT,
                maxT);

                if(intersection.t < closestIntersection.t)
                {
                        closestIntersection.isFound = true;
                        closestIntersection.isDestination = false;
                        closestIntersection.t = intersection.t;
                        closestIntersection.normal = intersection.normal;
                }
                else if(intersection.t == closestIntersection.t)
                {
                        closestIntersection.normal = (intersection.normal +
closestIntersection.normal) / 2;
                }
                }
        }
        }
        }

        // for(int i = 0; i < geometryMetadataBuffer.data.indexCount; i+=3)
        // {
        //      Intersection intersection = testRayTriangleIntersection(
        //      ray,
        //      geometryVertexBuffer.data[geometryIndexBuffer.data[i + 0]].position,
        //      geometryVertexBuffer.data[geometryIndexBuffer.data[i + 1]].position,
        //      geometryVertexBuffer.data[geometryIndexBuffer.data[i + 2]].position,
        //      minT,
        //      maxT);

        //      if(intersection.t < closestIntersection.t)
        //      {
        //      closestIntersection.isFound = true;
        //      closestIntersection.isDestination = false;
        //      closestIntersection.t = intersection.t;
        //      closestIntersection.normal = intersection.normal;
        //      }
        //      else if(intersection.t == closestIntersection.t)
        //      {
        //      closestIntersection.normal = (intersection.normal + closestIntersection.normal) /
2;
        //      }
        // }

        return closestIntersection;
}

/**
 * @brief Shader entry point
 */
void main()
```

```
{
        uint gID = gl_GlobalInvocationID.x;
        #ifdef DEBUG_OUTPUT
        outputBuffer.data[gID].successfulGeometryIntersectionTests = 0;
        outputBuffer.data[gID].totalGeometryIntersectionTests = 0;
        outputBuffer.data[gID].successfulAabbIntersectionTests = 0;
        outputBuffer.data[gID].totalAabbIntersectionTests = 0;
        #endif

        float angle = radians(360.0 / AUDIO_DIRECTION_COUNT);
        vec3 direction = vec3(0.0, 0.0, 1.0);
        for(int i = 0; i < gID; ++i)
        {
        direction = vec3(
        (direction.x * cos(angle)) - (direction.z * sin(angle)),
        0.0,
        (direction.z * cos(angle)) + (direction.x * sin(angle))
        );
        }

        Ray initialRay;
        initialRay.origin = pushConstants.audioComputeInput.audioSourcePosition;
        initialRay.direction = direction;
        initialRay.totalDistance = 0.0;

        RayQueue rayQueue;
        rayQueue.rays[0] = initialRay;
        rayQueue.current = 0;
        rayQueue.end = 1;

        Intersection intersection = defaultIntersection();

        // For each ray in the queue of rays to process
        while(rayQueue.current < rayQueue.end)
        {
        // Find the closest intersection between the ray and the environment
        intersection = findClosestIntersection(
        rayQueue.rays[rayQueue.current],
        FLT_MIN,
        FLT_MAX);

        if(intersection.isFound)
        {
        if(!intersection.isDestination && rayQueue.end < MAX_RAYS)
        {
                // If the ray has not reached it's destination, and the maximum number of rays has
not been computed,
                // reflect the ray around the intersection normal and add the reflected ray to the
queue of rays to process
                Ray reflectedRay;
                reflectedRay.origin = rayQueue.rays[rayQueue.current].origin +
(rayQueue.rays[rayQueue.current].direction * intersection.t);
                reflectedRay.direction = reflectRay(rayQueue.rays[rayQueue.current].direction,
intersection.normal);
                reflectedRay.totalDistance = rayQueue.rays[rayQueue.current].totalDistance +
intersection.t;

                // Move the reflected ray origin along the intersection normal to avoid incorrectly
intersecting the same object
                reflectedRay.origin = reflectedRay.origin + (intersection.normal * 0.0001);

                rayQueue.rays[rayQueue.end] = reflectedRay;
                rayQueue.end++;
```

```
        }
        else if(intersection.isDestination)
        {
                // Destination found
                break;
        }
        }

        ++rayQueue.current;
        }

        outputBuffer.data[gID].intersectionFound = intersection.isDestination;
        outputBuffer.data[gID].finalRay = rayQueue.rays[rayQueue.current];

        ///////////////////////
        // Debugging Output //
        ///////////////////////
        if(rayQueue.current == rayQueue.end)
        {
        --rayQueue.current;
        }

        outputBuffer.data[gID].intersectionT = intersection.t;
        outputBuffer.data[gID].intersectionNormal = intersection.normal;
        outputBuffer.data[gID].audioSourcePosition =
pushConstants.audioComputeInput.audioSourcePosition;
        outputBuffer.data[gID].audioDirection = direction;

        // for(int i = 0; i < MAX_RAYS; ++i)
        // {
        //      outputBuffer.data[gID].rays[i] = rayQueue.rays[i];
        // }
}
```

6.1.13 Fast Ray-Box Intersection GLSL Code

```
/**
 * @brief Test for a ray-AABB intersection using the Fast Ray-Box Intersection
 * @see Woo (1990)
 * @see Graphics Gems, 1990, pp. 395-396
 *
 * @param ray The ray to test
 * @param aabb The axis aligned bounding box to test
 *
 * @returns True if an intersection is found, otherwise false
 */
bool testRayAabbIntersection(Ray ray, AABB aabb)
{
        #ifdef DEBUG_OUTPUT
        uint gID = gl_GlobalInvocationID.x;
        outputBuffer.data[gID].totalAabbIntersectionTests++;
        #endif

        bool xInside = true;
        bool yInside = true;
        bool zInside = true;

        vec3 candidatePlane = vec3(0.0, 0.0, 0.0);
        vec3 maxT = vec3(0.0, 0.0, 0.0);

        vec3 boxMin = vec3(
```

```
aabb.position.x - aabb.extents.x,
aabb.position.y - aabb.extents.y,
aabb.position.z - aabb.extents.z
);
vec3 boxMax = vec3(
aabb.position.x + aabb.extents.x,
aabb.position.y + aabb.extents.y,
aabb.position.z + aabb.extents.z
);

// Find candidate planes, determine if the ray origin is inside the AABB
if (ray.origin.x < boxMin.x)
{
candidatePlane.x = boxMin.x;
xInside = false;
}
else if (ray.origin.x > boxMax.x)
{
candidatePlane.x = boxMax.x;
xInside = false;
}

if (ray.origin.y < boxMin.y)
{
candidatePlane.y = boxMin.y;
yInside = false;
}
else if (ray.origin.y > boxMax.y)
{
candidatePlane.y = boxMax.y;
yInside = false;
}

if (ray.origin.z < boxMin.z)
{
candidatePlane.z = boxMin.z;
zInside = false;
}
else if (ray.origin.z > boxMax.z)
{
candidatePlane.z = boxMax.z;
zInside = false;
}

if (xInside && yInside && zInside) {
#ifdef DEBUG_OUTPUT
outputBuffer.data[gID].successfulAabbIntersectionTests++;
#endif
return true;
}

// Find candidate plane T values
maxT.x = !xInside && ray.direction.x != 0.0
? (candidatePlane.x - ray.origin.x) / ray.direction.x
: -1.0;

maxT.y = !yInside && ray.direction.y != 0.0
? (candidatePlane.y - ray.origin.y) / ray.direction.y
: -1.0;

maxT.z = !zInside && ray.direction.z != 0.0
? (candidatePlane.z - ray.origin.z) / ray.direction.z
: -1.0;
```

```
        // For the largest T value, confirm the intersection
        if (max(maxT.x, max(maxT.y, maxT.z)) < 0.0)
        {
        return false;
        }

        if (maxT.x > maxT.y && maxT.x > maxT.z)
        {
        // Check Y & Z
        float y = ray.origin.y + maxT.x * ray.direction.y;
        float z = ray.origin.z + maxT.x * ray.direction.z;
        if ((y < boxMin.y || y > boxMax.y) ||
        (z < boxMin.z || z > boxMax.z))
        {
        return false;
        }
        }
        else if (maxT.y > maxT.x && maxT.y > maxT.z)
        {
        // Check X & Z
        float x = ray.origin.x + maxT.y * ray.direction.x;
        float z = ray.origin.z + maxT.y * ray.direction.z;
        if ((x < boxMin.x || x > boxMax.x) ||
        (z < boxMin.z || z > boxMax.z))
        {
        return false;
        }
        }
        else if (maxT.z > maxT.x && maxT.z > maxT.y)
        {
        // Check X & Y
        float x = ray.origin.x + maxT.z * ray.direction.x;
        float y = ray.origin.y + maxT.z * ray.direction.y;
        if ((x < boxMin.x || x > boxMax.x) ||
        (y < boxMin.y || y > boxMax.y))
        {
        return false;
        }
        }

        #ifdef DEBUG_OUTPUT
        outputBuffer.data[gID].successfulAabbIntersectionTests++;
        #endif
        return true;
}
```

6.1.14 Partition Shader Updated Closest Intersection GLSL Code

```
/**
 * @brief Find the closest intersection for a ray
 *
 * @param ray The ray to test
 * @param minT The minimum distance along the ray which can intersect
 * @param maxT The maximum distance along the ray which can intersect
 *
 * @returns The closest intersection, if found
 */
Intersection findClosestIntersection(Ray ray, float minT, float maxT)
{
        Intersection closestIntersection = defaultIntersection();
```

```
        float sphereRadius = 1.0;
        Intersection sphereIntersection = testRaySphereIntersection(
        ray,
        pushConstants.audioComputeInput.listenerPosition,
        sphereRadius,
        minT,
        maxT);

        if(sphereIntersection.t < closestIntersection.t)
        {
        closestIntersection.isFound = true;
        closestIntersection.isDestination = true;
        closestIntersection.t = sphereIntersection.t;
        closestIntersection.normal = sphereIntersection.normal;
        }

        int nodeStack[50];
        int nodeStackSize = 1;
        nodeStack[0] = 0;

        while(nodeStackSize > 0)
        {
        // Pop top KD tree node off of the stack
        KdNode currentNode = geometryKdTreeNodeBuffer.data[nodeStack[nodeStackSize-1]];
        nodeStackSize--;

        // Test AABB intersection
        bool aabbIntersection = testRayAabbIntersection(ray, currentNode.aabb);
        if(aabbIntersection)
        {
        // If inner node, add children to the stack of nodes to process
        if(currentNode.nearChild != INVALID_INDEX && currentNode.farChild != INVALID_INDEX)
        {
                nodeStack[nodeStackSize] = currentNode.farChild;
                nodeStackSize++;
                nodeStack[nodeStackSize] = currentNode.nearChild;
                nodeStackSize++;
        }
        // Leaf node, test for geometry intersection
        else
        {
                for(int i = currentNode.indicesStart; i < currentNode.indicesEnd; ++i)
                {
                Intersection intersection = testRayTriangleIntersection(
                ray,

geometryVertexBuffer.data[geometryIndexBuffer.data[geometryKdTreeLeafNodeIndexBuffer.data[i] +
0]].position,

geometryVertexBuffer.data[geometryIndexBuffer.data[geometryKdTreeLeafNodeIndexBuffer.data[i] +
1]].position,

geometryVertexBuffer.data[geometryIndexBuffer.data[geometryKdTreeLeafNodeIndexBuffer.data[i] +
2]].position,
                minT,
                maxT);

                if(intersection.t < closestIntersection.t)
                {
                        closestIntersection.isFound = true;
                        closestIntersection.isDestination = false;
                        closestIntersection.t = intersection.t;
```

```
                        closestIntersection.normal = intersection.normal;
                }
                else if(intersection.t == closestIntersection.t)
                {
                        closestIntersection.normal = (intersection.normal +
closestIntersection.normal) / 2;
                }
                }
        }
        }
        }

        return closestIntersection;
}
```

6.1.15 KD-Tree Construction C++ Code

```
void VulkanRenderer::buildGeometryKdTree(std::uint32_t desiredDepth)
{
        // Determine final node count of binary tree
        int finalNodeCount = 0;
        for (int i = desiredDepth; i > 0; i--)
        {
        finalNodeCount += i * 2;
        }
        finalNodeCount++;

        // Get the first index of each triangle
        std::set<uint32_t> geometryIndices{ };
        for (size_t i = 0; i < computGeometryIndexStagingBuffer.size(); i+=3)
        {
        geometryIndices.insert(i);
        }

        // Create list of tree nodes
        std::vector<KdNode> kdTreeNodes{ };
        std::vector<KdNodeMetaData> kdTreeNodeMetaData{ };

        // Create list of indices to reference in leaf nodes
        std::vector<std::uint32_t> leafNodeGeometryIndices{ };

        // Seed the list with the root node
        KdNode kdNode{ };
        kdNode.aabb = buildKdNodeAabb(geometryIndices);

        kdTreeNodes.push_back(kdNode);
        kdTreeNodeMetaData.push_back(
        {
        { 1.0f, 0.0f, 0.0f},
        0,
        false,
        geometryIndices
        }
        );

        // Create queue of nodes to split (indices into the node list)
        std::queue<std::uint32_t> nodesToSplit{ };

        // Seed queue with the root node
        nodesToSplit.push(0);
```

```
        // While queue not empty, and desired depth not reached, split on the next queued node
        while (!nodesToSplit.empty())
        {
        // Get next queued node
        std::uint32_t currentNodeIndex = nodesToSplit.front();
        nodesToSplit.pop();

        if (!kdTreeNodeMetaData[currentNodeIndex].isLeafNode)
        {
        // Split the node
        auto children = splitKdNode(kdTreeNodes[currentNodeIndex],
kdTreeNodeMetaData[currentNodeIndex]);

        // Add split children to the node list
        kdTreeNodes.push_back(children.first.first);
        kdTreeNodes.push_back(children.second.first);
        std::uint32_t nearChildIndex = kdTreeNodes.size() - 2;
        std::uint32_t farChildIndex = kdTreeNodes.size() - 1;

        // Push metadata for each child node
        children.first.second.isLeafNode = children.first.second.nodeDepth == desiredDepth;
        children.second.second.isLeafNode = children.second.second.nodeDepth == desiredDepth;
        kdTreeNodeMetaData.push_back(children.first.second);
        kdTreeNodeMetaData.push_back(children.second.second);

        // Add nodes to list to be processed
        nodesToSplit.push(nearChildIndex);
        nodesToSplit.push(farChildIndex);

        // Set child indices on the parent node
        kdTreeNodes[currentNodeIndex].nearChild = nearChildIndex;
        kdTreeNodes[currentNodeIndex].farChild = farChildIndex;

        // Clear geometry from the parent node
        kdTreeNodeMetaData[currentNodeIndex].GeometryIndices.clear();
        }
        else
        {
        kdTreeNodes[currentNodeIndex].indicesStart = leafNodeGeometryIndices.size();
        leafNodeGeometryIndices.insert(leafNodeGeometryIndices.end(),
kdTreeNodeMetaData[currentNodeIndex].GeometryIndices.begin(),
kdTreeNodeMetaData[currentNodeIndex].GeometryIndices.end());
        kdTreeNodes[currentNodeIndex].indicesEnd = leafNodeGeometryIndices.size();
        }
        }
}



std::pair<std::pair<KdNode, KdNodeMetaData>, std::pair<KdNode, KdNodeMetaData>>
VulkanRenderer::splitKdNode(KdNode & kdNode, KdNodeMetaData& kdNodeMetaData)
{
        KdNodeMetaData commonMetadata{ };
        commonMetadata.nodeDepth = kdNodeMetaData.nodeDepth + 1;
        commonMetadata.splitAxis = kdNodeMetaData.splitAxis.x == 1.0f
        // If current node was split on the x-axis, then children must be split on the z-axis
        ? math::Vector3{ 0.0f, 0.0f, 1.0f }
        // Otherwise, current node was split on the z-axis, children must be split on the x-axis
        : math::Vector3{ 1.0f, 0.0f, 0.0f };

        KdNodeMetaData nearChildMetadata{ };
        nearChildMetadata.nodeDepth = commonMetadata.nodeDepth;
        nearChildMetadata.splitAxis = commonMetadata.splitAxis;
```

```
        KdNodeMetaData farChildMetadata{ };
        farChildMetadata.nodeDepth = commonMetadata.nodeDepth;
        farChildMetadata.splitAxis = commonMetadata.splitAxis;

        // Determine split point
        float splitPoint = determineAabbSplitPlane(kdNode.aabb, kdNodeMetaData.GeometryIndices,
kdNodeMetaData.splitAxis);

        // Split vertex indices amongst the children
        math::Vector3 aabbMin
        {
        kdNode.aabb.position.x - kdNode.aabb.extents.x,
        kdNode.aabb.position.y - kdNode.aabb.extents.y,
        kdNode.aabb.position.z - kdNode.aabb.extents.z
        };
        math::Vector3 aabbMax
        {
        kdNode.aabb.position.x + kdNode.aabb.extents.x,
        kdNode.aabb.position.y + kdNode.aabb.extents.y,
        kdNode.aabb.position.z + kdNode.aabb.extents.z
        };
        for (auto index : kdNodeMetaData.GeometryIndices)
        {
        for (size_t i = 0; i < 3; ++i)
        {
        auto vertex = computGeometryVertexStagingBuffer[computGeometryIndexStagingBuffer[index +
i]];

        // If the vertex is within the parent AABB
        if ((vertex.position.x >= aabbMin.x && vertex.position.x <= aabbMax.x) &&
                (vertex.position.y >= aabbMin.y && vertex.position.y <= aabbMax.y) &&
                (vertex.position.z >= aabbMin.z && vertex.position.z <= aabbMax.z))
        {
                float vertexAxisPoint = kdNodeMetaData.splitAxis.x == 1.0f
                ? vertex.position.x
                : vertex.position.z;

                if (vertexAxisPoint <= splitPoint)
                {
                nearChildMetadata.GeometryIndices.insert(index);
                }

                if (vertexAxisPoint >= splitPoint)
                {
                farChildMetadata.GeometryIndices.insert(index);
                }
        }
        }
        }

        // Build child node AABBs
        std::pair<math::AABB, math::AABB> childAabbs = splitAabb(kdNode.aabb, splitPoint,
kdNodeMetaData.splitAxis);
        KdNode nearChild{ };
        nearChild.aabb = childAabbs.first;

        KdNode farChild{ };
        farChild.aabb = childAabbs.second;

        // Check for geometry that overlaps the child AABBs, but may not be contained within them
        for (auto index : kdNodeMetaData.GeometryIndices)
        {
```

```
        auto vertex0 = computGeometryVertexStagingBuffer[computGeometryIndexStagingBuffer[index +
0]].position;
        auto vertex1 = computGeometryVertexStagingBuffer[computGeometryIndexStagingBuffer[index +
1]].position;
        auto vertex2 = computGeometryVertexStagingBuffer[computGeometryIndexStagingBuffer[index +
2]].position;

        if (math::triangleAabbInterection(vertex0, vertex1, vertex2, nearChild.aabb))
        {
        nearChildMetadata.GeometryIndices.insert(index);
        }

        if (math::triangleAabbInterection(vertex0, vertex1, vertex2, farChild.aabb))
        {
        farChildMetadata.GeometryIndices.insert(index);
        }
        }

        return std::make_pair(
        std::make_pair(nearChild, nearChildMetadata),
        std::make_pair(farChild, farChildMetadata)
        );
}



std::pair<math::AABB, math::AABB> VulkanRenderer::splitAabb(math::AABB aabb, float splitPoint,
math::Vector3 splitAxis)
{
        math::Vector3 nearMin{ };
        math::Vector3 nearMax{ };
        math::Vector3 farMin{ };
        math::Vector3 farMax{ };

        if (splitAxis.x == 1.0f)
        {
        nearMin = { aabb.position.x - aabb.extents.x, aabb.position.y - aabb.extents.y,
aabb.position.z - aabb.extents.z };
        nearMax = { splitPoint, aabb.position.y + aabb.extents.y, aabb.position.z + aabb.extents.z
};

        farMin = { splitPoint, aabb.position.y - aabb.extents.y, aabb.position.z - aabb.extents.z
};
        farMax = { aabb.position.x + aabb.extents.x, aabb.position.y + aabb.extents.y,
aabb.position.z + aabb.extents.z };
        }
        else
        {
        nearMin = { aabb.position.x - aabb.extents.x, aabb.position.y - aabb.extents.y,
aabb.position.z - aabb.extents.z };
        nearMax = { aabb.position.x + aabb.extents.x, aabb.position.y + aabb.extents.y, splitPoint
};

        farMin = { aabb.position.x - aabb.extents.x, aabb.position.y - aabb.extents.y, splitPoint
};
        farMax = { aabb.position.x + aabb.extents.x, aabb.position.y + aabb.extents.y,
aabb.position.z + aabb.extents.z };
        }

        math::AABB nearAabb{ };
        nearAabb.position =
        {
        (nearMin.x + nearMax.x) / 2,
```

```
        (nearMin.y + nearMax.y) / 2,
        (nearMin.z + nearMax.z) / 2
        };
        nearAabb.extents =
        {
        nearMax.x - nearAabb.position.x,
        nearMax.y - nearAabb.position.y,
        nearMax.z - nearAabb.position.z
        };

        math::AABB farAabb{ };
        farAabb.position =
        {
        (farMin.x + farMax.x) / 2,
        (farMin.y + farMax.y) / 2,
        (farMin.z + farMax.z) / 2
        };
        farAabb.extents =
        {
        farMax.x - farAabb.position.x,
        farMax.y - farAabb.position.y,
        farMax.z - farAabb.position.z
        };

        return std::make_pair(nearAabb, farAabb);
}



float VulkanRenderer::determineAabbSplitPlane(math::AABB aabb, std::set<std::uint32_t>
geometryIndices, math::Vector3 splitAxis)
{
        float vertMin = FLT_MAX,
        vertMax = FLT_MIN;

        math::Vector3 aabbMin
        {
        aabb.position.x - aabb.extents.x,
        aabb.position.y - aabb.extents.y,
        aabb.position.z - aabb.extents.z
        };
        math::Vector3 aabbMax
        {
        aabb.position.x + aabb.extents.x,
        aabb.position.y + aabb.extents.y,
        aabb.position.z + aabb.extents.z
        };

        for (auto index : geometryIndices)
        {
        for (size_t i = 0; i < 3; ++i)
        {
        auto vertex = computGeometryVertexStagingBuffer[computGeometryIndexStagingBuffer[index +
i]];

        // If the vertex is within the AABB
        if ((vertex.position.x >= aabbMin.x && vertex.position.x <= aabbMax.x) &&
                (vertex.position.y >= aabbMin.y && vertex.position.y <= aabbMax.y) &&
                (vertex.position.z >= aabbMin.z && vertex.position.z <= aabbMax.z))
        {
                if (splitAxis.x == 1.0f)
                {
                vertMin = std::min(vertMin, vertex.position.x);
```

```
                vertMax = std::max(vertMax, vertex.position.x);
                }
                else
                {
                vertMin = std::min(vertMin, vertex.position.z);
                vertMax = std::max(vertMax, vertex.position.z);
                }
        }
        }
        }

        return (vertMin + vertMax) / 2;
}



math::AABB VulkanRenderer::buildKdNodeAabb(std::set<std::uint32_t> geometryIndices)
{
        float minX = FLT_MAX, minY = FLT_MAX, minZ = FLT_MAX,
        maxX = FLT_MIN, maxY = FLT_MIN, maxZ = FLT_MIN;

        for (auto index : geometryIndices)
        {
        for (size_t i = 0; i < 3; ++i)
        {
        auto vertex = computGeometryVertexStagingBuffer[computGeometryIndexStagingBuffer[index +
i]];

        minX = std::min(minX, vertex.position.x);
        minY = std::min(minY, vertex.position.y);
        minZ = std::min(minZ, vertex.position.z);

        maxX = std::max(maxX, vertex.position.x);
        maxY = std::max(maxY, vertex.position.y);
        maxZ = std::max(maxZ, vertex.position.z);
        }
        }

        math::AABB aabb = {};
        aabb.position =
        {
        (minX + maxX) / 2,
        (minY + maxY) / 2,
        (minZ + maxZ) / 2
        };
        aabb.extents =
        {
        maxX - aabb.position.x,
        maxY - aabb.position.y,
        maxZ - aabb.position.z
        };

        return aabb;
}
```

6.1.16 Copy KD-Tree To GPU Storage Buffers C++ Code

```
auto const commandBuffer = VulkanCommandBuffer::Builder()
        .withDevice(device.get())
        .withCommandPool(graphicsCommandPool.get())
        .withLevel(VK_COMMAND_BUFFER_LEVEL_PRIMARY)
```

```
        .build();

// Copy KD tree data to device local buffer
{
        VkDeviceSize bufferSize = sizeof(KdNode) * kdTreeNodes.size();
        computeGeometryKdTreeBuffer = VulkanBuffer::Builder()
        .withDevice(device.get())
        .withBufferSize(bufferSize)
        .withBufferUsageFlags(VK_BUFFER_USAGE_STORAGE_BUFFER_BIT |
VK_BUFFER_USAGE_TRANSFER_DST_BIT)
        .withMemoryPropertyFlags(VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT)
        .build();

        auto const stagingBuffer = VulkanBuffer::Builder()
        .withDevice(device.get())
        .withBufferSize(bufferSize)
        .withBufferUsageFlags(VK_BUFFER_USAGE_TRANSFER_SRC_BIT)
        .withMemoryPropertyFlags(VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
VK_MEMORY_PROPERTY_HOST_COHERENT_BIT)
        .build();

        stagingBuffer->update(kdTreeNodes.data(), bufferSize);
        stagingBuffer->copyToBuffer(commandBuffer.get(), device->getGraphicsQueue().get(),
computeGeometryKdTreeBuffer.get(), bufferSize);

        computeGeometryDescriptorSet->updateFromBuffer(computeGeometryKdTreeBuffer.get(), 0,
bufferSize, 3, VK_DESCRIPTOR_TYPE_STORAGE_BUFFER, device.get());
}

// Copy KD tree leaf node indices data to device local buffer
{
        VkDeviceSize bufferSize = sizeof(std::uint32_t) * leafNodeGeometryIndices.size();
        computeGeometryKdTreeLeafIndicesBuffer = VulkanBuffer::Builder()
        .withDevice(device.get())
        .withBufferSize(bufferSize)
        .withBufferUsageFlags(VK_BUFFER_USAGE_STORAGE_BUFFER_BIT |
VK_BUFFER_USAGE_TRANSFER_DST_BIT)
        .withMemoryPropertyFlags(VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT)
        .build();

        auto const stagingBuffer = VulkanBuffer::Builder()
        .withDevice(device.get())
        .withBufferSize(bufferSize)
        .withBufferUsageFlags(VK_BUFFER_USAGE_TRANSFER_SRC_BIT)
        .withMemoryPropertyFlags(VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
VK_MEMORY_PROPERTY_HOST_COHERENT_BIT)
        .build();

        stagingBuffer->update(leafNodeGeometryIndices.data(), bufferSize);
        stagingBuffer->copyToBuffer(commandBuffer.get(), device->getGraphicsQueue().get(),
computeGeometryKdTreeLeafIndicesBuffer.get(), bufferSize);


computeGeometryDescriptorSet->updateFromBuffer(computeGeometryKdTreeLeafIndicesBuffer.get(), 0,
bufferSize, 4, VK_DESCRIPTOR_TYPE_STORAGE_BUFFER, device.get());
}

VkCommandBuffer const commandBufferHandle = commandBuffer->getCommandBuffer();
vkFreeCommandBuffers(device->getLogicalDevice(), graphicsCommandPool->getCommandPool(), 1,
&commandBufferHandle);
```

6.1.17 Ray Intersection Prediction Audio Propagation GLSL Code

```
#version 450


////////////////////////////////////
////////////// Defines /////////////
////////////////////////////////////
#define FLT_MAX 3.402823466e+38
#define FLT_MIN 1.175494351e-38
#define FLT_EPSILON 1.192092896e-07
#define PI 3.1415926538
#define DEBUG_OUTPUT


////////////////////////////////////
////////////// Constants ///////////
////////////////////////////////////
const int MAX_RAYS = 500;
const int AUDIO_DIRECTION_COUNT = 512;
const int INVALID_INDEX = 4294967295;
const int HASH_GRID_DIMENSIONS = 2048;
const int DIRECTION_HASH_RESOLUTION = 128;


////////////////////////////////////
////////////// Structs /////////////
////////////////////////////////////
struct Ray
{
        vec3 origin;
        vec3 direction;
        float totalDistance;
};

struct RayQueue
{
        int current;
        int end;
};

struct AudioComputeVertex3
{
        vec3 position;
        vec3 normal;
};

struct AudioComputeGeometryMetadata
{
        int vertexCount;
        int indexCount;
        int isPredictionAvailable;
};

struct Intersection
{
        bool isFound;
        bool isDestination;
        float t;
        vec3 normal;
        int leafNodeIndex;
};

struct AudioComputeInput
```

```
{
	vec3 audioSourcePosition;
	vec3 listenerPosition;
};

struct AudioComputeOutput
{
	bool intersectionFound;
	float intersectionT;
	vec3 intersectionNormal;
	vec3 audioSourcePosition;
	vec3 audioDirection;
	Ray finalRay;
	int successfulGeometryIntersectionTests;
	int totalGeometryIntersectionTests;
	int successfulAabbIntersectionTests;
	int totalAabbIntersectionTests;
	int predictions;
	int predictionMisses;
	Ray rays[MAX_RAYS];
};

struct AABB
{
	vec3 position;
	vec3 extents;
};

struct KdNode
{
	AABB aabb;
	int nearChild;
	int farChild;
	int indicesStart;
	int indicesEnd;
};

struct RayGeometryCollision
{
	Ray ray;
	int leafNodeIndex;
};

struct RayGeometryCollisions
{
	RayGeometryCollision collisions[MAX_RAYS];
};

struct RayDirectionPredictions
{
	int directions[4];
};


////////////////////////////////
////////// Shader Inputs //////////
////////////////////////////////
layout(push_constant) uniform PushModel {
	AudioComputeInput audioComputeInput;
} pushConstants;

layout(set = 0, binding = 0) buffer GeometryVertexBuffer
{
```

```
        AudioComputeVertex3 data[];
} geometryVertexBuffer;

layout(set = 0, binding = 1) buffer GeometryIndexBuffer
{
        int data[];
} geometryIndexBuffer;

layout(set = 0, binding = 2) buffer GeometryMetadataBuffer
{
        AudioComputeGeometryMetadata data;
} geometryMetadataBuffer;

layout(set = 0, binding = 3) buffer GeometryKdTreeNodeBuffer
{
        KdNode data[];
} geometryKdTreeNodeBuffer;

layout(set = 0, binding = 4) buffer GeometryKdTreeLeafNodeIndexBuffer
{
        int data[];
} geometryKdTreeLeafNodeIndexBuffer;

layout(set = 0, binding = 5) buffer RayGeometryCollisionsBuffer
{
        RayGeometryCollisions data[];
} rayGeometryCollisionsBuffer;

layout(set = 0, binding = 6) buffer RayGeometryCollisionPredictionBuffer
{
        RayDirectionPredictions data[];
} rayGeometryCollisionPredictionBuffer;

layout(set = 1, binding = 0) buffer OutputBuffer
{
        AudioComputeOutput data[];
} outputBuffer;


////////////////////////////////////
////////////// Functions //////////
////////////////////////////////////

/**
 * @brief Initialize an intersection with default values
 *
 * @returns The intersection
 */
Intersection defaultIntersection()
{
        Intersection intersection;
        intersection.isFound = false;
        intersection.isDestination = false;
        intersection.t = FLT_MAX;
        intersection.normal = vec3(0.0);
        intersection.leafNodeIndex = INVALID_INDEX;

        return intersection;
}

/**
 * @brief Spatially hash a ray origin into an integer
 *
```

```
 * @param ray The ray to hash
 * @param aabb The aabb defining the space with which to hash the ray (treated as an NxN grid)
 *
 * @returns The hashed origin
 */
int hashRayOrigin(Ray ray, AABB aabb)
{
        int minGridIndex = 0;
        int maxGridIndex = HASH_GRID_DIMENSIONS - 1;
        int gridRange = (maxGridIndex - minGridIndex);

        int minX = int(aabb.position.x - aabb.extents.x);
        int maxX = int(aabb.position.x + aabb.extents.x);
        int xRange = (maxX - minX);

        int minZ = int(aabb.position.z - aabb.extents.z);
        int maxZ = int(aabb.position.z + aabb.extents.z);
        int zRange = (maxZ - minZ);

        int x = (((int(ray.origin.x) - minX) * gridRange) / xRange) + minGridIndex;
        int z = (((int(ray.origin.z) - minZ) * gridRange) / zRange) + minGridIndex;

        return (z * HASH_GRID_DIMENSIONS) + x;
}

/**
 * @brief Hash a ray direction into an integer describing one of N directions
 *
 * @param ray The ray to hash
 *
 * @returns The hashed direction
 */
int hashRayDirection(Ray ray)
{
        float stepRadians = (2 * PI) / DIRECTION_HASH_RESOLUTION;
        float hashAngle = stepRadians;
        for (int i = 0; i < DIRECTION_HASH_RESOLUTION; ++i)
        {
        float vAngle = atan(ray.direction.z, ray.direction.x);
        if (vAngle < 0)
        {
        vAngle += 2 * PI;
        }
        if (vAngle <= hashAngle)
        {
        return i;
        }

        hashAngle += stepRadians;
        }
}

/**
 * @brief Test for a ray-AABB intersection using the Fast Ray-Box Intersection
 * @see Graphics Gems, 1990, pp. 395-396
 *
 * @param ray The ray to test
 * @param aabb The axis aligned bounding box to test
 *
 * @returns True if an intersection is found, otherwise false
 */
bool testRayAabbIntersection(Ray ray, AABB aabb)
{
```

```
#ifdef DEBUG_OUTPUT
uint gID = gl_GlobalInvocationID.x;
outputBuffer.data[gID].totalAabbIntersectionTests++;
#endif

bool xInside = true;
bool yInside = true;
bool zInside = true;

vec3 candidatePlane = vec3(0.0, 0.0, 0.0);
vec3 maxT = vec3(0.0, 0.0, 0.0);

vec3 boxMin = vec3(
aabb.position.x - aabb.extents.x,
aabb.position.y - aabb.extents.y,
aabb.position.z - aabb.extents.z
);
vec3 boxMax = vec3(
aabb.position.x + aabb.extents.x,
aabb.position.y + aabb.extents.y,
aabb.position.z + aabb.extents.z
);

// Find candidate planes, determine if the ray origin is inside the AABB
if (ray.origin.x < boxMin.x)
{
candidatePlane.x = boxMin.x;
xInside = false;
}
else if (ray.origin.x > boxMax.x)
{
candidatePlane.x = boxMax.x;
xInside = false;
}

if (ray.origin.y < boxMin.y)
{
candidatePlane.y = boxMin.y;
yInside = false;
}
else if (ray.origin.y > boxMax.y)
{
candidatePlane.y = boxMax.y;
yInside = false;
}

if (ray.origin.z < boxMin.z)
{
candidatePlane.z = boxMin.z;
zInside = false;
}
else if (ray.origin.z > boxMax.z)
{
candidatePlane.z = boxMax.z;
zInside = false;
}

if (xInside && yInside && zInside) {
#ifdef DEBUG_OUTPUT
outputBuffer.data[gID].successfulAabbIntersectionTests++;
#endif
return true;
}
```

```
        // Find candidate plane T values
        maxT.x = !xInside && ray.direction.x != 0.0
        ? (candidatePlane.x - ray.origin.x) / ray.direction.x
        : -1.0;

        maxT.y = !yInside && ray.direction.y != 0.0
        ? (candidatePlane.y - ray.origin.y) / ray.direction.y
        : -1.0;

        maxT.z = !zInside && ray.direction.z != 0.0
        ? (candidatePlane.z - ray.origin.z) / ray.direction.z
        : -1.0;

        // For the largest T value, confirm the intersection
        if (max(maxT.x, max(maxT.y, maxT.z)) < 0.0)
        {
        return false;
        }

        if (maxT.x > maxT.y && maxT.x > maxT.z)
        {
        // Check Y & Z
        float y = ray.origin.y + maxT.x * ray.direction.y;
        float z = ray.origin.z + maxT.x * ray.direction.z;
        if ((y < boxMin.y || y > boxMax.y) ||
        (z < boxMin.z || z > boxMax.z))
        {
        return false;
        }
        }
        else if (maxT.y > maxT.x && maxT.y > maxT.z)
        {
        // Check X & Z
        float x = ray.origin.x + maxT.y * ray.direction.x;
        float z = ray.origin.z + maxT.y * ray.direction.z;
        if ((x < boxMin.x || x > boxMax.x) ||
        (z < boxMin.z || z > boxMax.z))
        {
        return false;
        }
        }
        else if (maxT.z > maxT.x && maxT.z > maxT.y)
        {
        // Check X & Y
        float x = ray.origin.x + maxT.z * ray.direction.x;
        float y = ray.origin.y + maxT.z * ray.direction.y;
        if ((x < boxMin.x || x > boxMax.x) ||
        (y < boxMin.y || y > boxMax.y))
        {
        return false;
        }
        }

        #ifdef DEBUG_OUTPUT
        outputBuffer.data[gID].successfulAabbIntersectionTests++;
        #endif
        return true;
}

/**
 * @brief Test for a ray-triangle intersection using the Möller-Trumbore algorithm
 * @see https://en.wikipedia.org/wiki/M%C3%B6ller%E2%80%93Trumbore_intersection_algorithm
```

```
 *
 * @param ray The ray to test
 * @param vertex0 The first vertex describing the triangle
 * @param vertex1 The second vertex describing the triangle
 * @param vertex2 The third vertex describing the triangle
 * @param minT The minimum distance along the ray which can intersect
 * @param maxT The maximum distance along the ray which can intersect
 *
 * @returns The intersection, if found
 */
Intersection testRayTriangleIntersection(Ray ray, vec3 vertex0, vec3 vertex1, vec3 vertex2, float
minT, float maxT)
{
        #ifdef DEBUG_OUTPUT
        uint gID = gl_GlobalInvocationID.x;
        outputBuffer.data[gID].totalGeometryIntersectionTests++;
        #endif

        Intersection intersection = defaultIntersection();

        vec3 edge1 = vec3(0.0);
        vec3 edge2 = vec3(0.0);
        vec3 h = vec3(0.0);
        vec3 s = vec3(0.0);
        vec3 q = vec3(0.0);

        float a = 0.0;
        float f = 0.0;
        float u = 0.0;
        float v = 0.0;

        edge1 = vertex1 - vertex0;
        edge2 = vertex2 - vertex0;

        h = cross(ray.direction, edge2);
        a = dot(edge1, h);
        if (a > -FLT_EPSILON && a < FLT_EPSILON)
        {
        return intersection;
        }

        f = 1.0f / a;
        s = ray.origin - vertex0;
        u = f * dot(s, h);
        if (u < 0.0f || u > 1.0f)
        {
        return intersection;
        }

        q = cross(s, edge1);
        v = f * dot(ray.direction, q);
        if (v < 0.0f || u + v > 1.0f)
        {
        return intersection;
        }

        float t = f * dot(edge2, q);
        if (t > FLT_EPSILON && t > minT && t < maxT)
        {
        #ifdef DEBUG_OUTPUT
        outputBuffer.data[gID].successfulGeometryIntersectionTests++;
        #endif
```

```
        intersection.t = t;
        intersection.normal = normalize(cross(edge1, edge2));
        intersection.isFound = true;
        return intersection;
        }

        return intersection;
}

/**
 * @brief Test for a ray-sphere intersection using the quadratic formula
 *
 * @param ray The ray to test
 * @param spherePosition The center point of the sphere
 * @param radius The sphere radius
 * @param minT The minimum distance along the ray which can intersect
 * @param maxT The maximum distance along the ray which can intersect
 *
 * @returns The intersection, if found
 */
Intersection testRaySphereIntersection(Ray ray, vec3 spherePosition, float sphereRadius, float minT,
float maxT)
{
        Intersection intersection = defaultIntersection();

        float a = dot(ray.direction, ray.direction);
        float b = dot(ray.origin - spherePosition, ray.direction) * 2;
        float c = dot(ray.origin - spherePosition, ray.origin - spherePosition) - (sphereRadius *
sphereRadius);

        float discriminant = (b * b) - (4.0f * a * c);
        if (discriminant < 0.0f)
        {
        return intersection;
        }

        float intersectionA = (-b + sqrt(discriminant)) / (2.0f * a);
        float intersectionB = (-b - sqrt(discriminant)) / (2.0f * a);
        if ((intersectionA > minT && intersectionA < maxT) ||
        (intersectionB > minT && intersectionB < maxT))
        {
        intersection.t = min(intersectionA, intersectionB);
        intersection.normal = normalize((ray.origin + ray.direction * intersection.t) -
spherePosition);
        intersection.isFound = true;
        return intersection;
        }

        return intersection;
}

/**
 * @brief Reflects a ray around a given normal
 *
 * @param rayDirection The direction of the ray to reflect
 * @param normal The normal to reflect around
 *
 * @returns The reflected ray direction
 */
vec3 reflectRay(vec3 rayDirection, vec3 normal)
{
        normal = normalize(normal);
        rayDirection = normalize(rayDirection);
```

```
        mat3 rayReversalMatrix;
        rayReversalMatrix[0] = vec3(-1.0, 0.0, 0.0);
        rayReversalMatrix[1] = vec3(0.0, -1.0, 0.0);
        rayReversalMatrix[2] = vec3(0.0, 0.0, -1.0);

        vec3 raySourceDirection = rayReversalMatrix * rayDirection;
        return (normal * dot(normal, raySourceDirection) * 2) - raySourceDirection;
}

/**
 * @brief Find the closest intersection for a ray
 *
 * @param ray The ray to test
 * @param minT The minimum distance along the ray which can intersect
 * @param maxT The maximum distance along the ray which can intersect
 *
 * @returns The closest intersection, if found
 */
Intersection findClosestIntersection(Ray ray, float minT, float maxT)
{
        Intersection closestIntersection = defaultIntersection();

        float sphereRadius = 1.0;
        Intersection sphereIntersection = testRaySphereIntersection(
        ray,
        pushConstants.audioComputeInput.listenerPosition,
        sphereRadius,
        minT,
        maxT);

        if(sphereIntersection.t < closestIntersection.t)
        {
        closestIntersection.isFound = true;
        closestIntersection.isDestination = true;
        closestIntersection.t = sphereIntersection.t;
        closestIntersection.normal = sphereIntersection.normal;
        }

        ////////////////////////////////////////////
        ////////////////// Prediction //////////////
        ////////////////////////////////////////////
        int x = hashRayOrigin(ray, geometryKdTreeNodeBuffer.data[0].aabb);
        int z = hashRayDirection(ray);

        bool madePrediction = false;
        Intersection predictedClosestIntersection = defaultIntersection();
        int predictedKdNodeIndex = rayGeometryCollisionPredictionBuffer.data[x].directions[z];
        if(predictedKdNodeIndex != INVALID_INDEX)
        {
        madePrediction = true;
        #ifdef DEBUG_OUTPUT
        uint gID = gl_GlobalInvocationID.x;
        outputBuffer.data[gID].predictions++;
        #endif

        KdNode predictedNode = geometryKdTreeNodeBuffer.data[predictedKdNodeIndex];
        for(int i = predictedNode.indicesStart; i < predictedNode.indicesEnd; ++i)
        {
        Intersection intersection = testRayTriangleIntersection(
                ray,

geometryVertexBuffer.data[geometryIndexBuffer.data[geometryKdTreeLeafNodeIndexBuffer.data[i] +
```

```
0]].position,

geometryVertexBuffer.data[geometryIndexBuffer.data[geometryKdTreeLeafNodeIndexBuffer.data[i] +
1]].position,

geometryVertexBuffer.data[geometryIndexBuffer.data[geometryKdTreeLeafNodeIndexBuffer.data[i] +
2]].position,
                minT,
                maxT);

        if(intersection.t < predictedClosestIntersection.t)
        {
                predictedClosestIntersection.isFound = true;
                predictedClosestIntersection.isDestination = false;
                predictedClosestIntersection.t = intersection.t;
                predictedClosestIntersection.normal = intersection.normal;
                predictedClosestIntersection.leafNodeIndex = predictedKdNodeIndex;
        }
        else if(intersection.t == predictedClosestIntersection.t)
        {
                predictedClosestIntersection.normal = (intersection.normal +
predictedClosestIntersection.normal) / 2;
        }
        }

        #ifndef DEBUG_OUTPUT
        if(predictedClosestIntersection.isFound == true &&
                predictedClosestIntersection.t < closestIntersection.t)
        {
                return predictedClosestIntersection;
        }
        #endif
        }
        ////////////////////////////////////////////////


        ////////////////////////////////////////////////
        ////////////// KD Tree Traversal ///////////////
        ////////////////////////////////////////////////
        int nodeStack[50];
        int nodeStackSize = 1;
        nodeStack[0] = 0;

        while(nodeStackSize > 0)
        {
        // Pop top KD tree node off of the stack
        int currentKdNodeIndex = nodeStack[nodeStackSize-1];
        KdNode currentNode = geometryKdTreeNodeBuffer.data[currentKdNodeIndex];
        nodeStackSize--;

        // Test AABB intersection
        bool aabbIntersection = testRayAabbIntersection(ray, currentNode.aabb);
        if(aabbIntersection)
        {
        // If inner node, add children to the stack of nodes to process
        if(currentNode.nearChild != INVALID_INDEX && currentNode.farChild != INVALID_INDEX)
        {
                nodeStack[nodeStackSize] = currentNode.farChild;
                nodeStackSize++;
                nodeStack[nodeStackSize] = currentNode.nearChild;
                nodeStackSize++;
        }
        // Leaf node, test for geometry intersection
```

```
        else
        {
                for(int i = currentNode.indicesStart; i < currentNode.indicesEnd; ++i)
                {
                Intersection intersection = testRayTriangleIntersection(
                        ray,

geometryVertexBuffer.data[geometryIndexBuffer.data[geometryKdTreeLeafNodeIndexBuffer.data[i] +
0]].position,

geometryVertexBuffer.data[geometryIndexBuffer.data[geometryKdTreeLeafNodeIndexBuffer.data[i] +
1]].position,

geometryVertexBuffer.data[geometryIndexBuffer.data[geometryKdTreeLeafNodeIndexBuffer.data[i] +
2]].position,
                        minT,
                        maxT);

                if(intersection.t < closestIntersection.t)
                {
                        closestIntersection.isFound = true;
                        closestIntersection.isDestination = false;
                        closestIntersection.t = intersection.t;
                        closestIntersection.normal = intersection.normal;
                        closestIntersection.leafNodeIndex = currentKdNodeIndex;
                }
                else if(intersection.t == closestIntersection.t)
                {
                        closestIntersection.normal = (intersection.normal +
closestIntersection.normal) / 2;
                }
                }
        }
        }
        }
        ////////////////////////////////////////////

        #ifdef DEBUG_OUTPUT
        if(madePrediction)
        {
        if(predictedClosestIntersection.leafNodeIndex != closestIntersection.leafNodeIndex)
        {
                uint gID = gl_GlobalInvocationID.x;
                outputBuffer.data[gID].predictionMisses++;
        }
        }
        #endif

        return closestIntersection;
}

/**
 * @brief Shader entry point
 */
void main()
{
        uint gID = gl_GlobalInvocationID.x;
        #ifdef DEBUG_OUTPUT
        outputBuffer.data[gID].successfulGeometryIntersectionTests = 0;
        outputBuffer.data[gID].totalGeometryIntersectionTests = 0;
        outputBuffer.data[gID].successfulAabbIntersectionTests = 0;
        outputBuffer.data[gID].totalAabbIntersectionTests = 0;
        outputBuffer.data[gID].predictions = 0;
```

```
        outputBuffer.data[gID].predictionMisses = 0;
        #endif

        float angle = radians(360.0 / AUDIO_DIRECTION_COUNT);
        vec3 direction = vec3(0.0, 0.0, 1.0);
        for(int i = 0; i < gID; ++i)
        {
        direction = vec3(
        (direction.x * cos(angle)) - (direction.z * sin(angle)),
        0.0,
        (direction.z * cos(angle)) + (direction.x * sin(angle))
        );
        }

        Ray initialRay;
        initialRay.origin = pushConstants.audioComputeInput.audioSourcePosition;
        initialRay.direction = direction;
        initialRay.totalDistance = 0.0;

        RayQueue rayQueue;
        rayQueue.current = 0;
        rayQueue.end = 1;

        RayGeometryCollision initialCollision;
        initialCollision.ray = initialRay;
        initialCollision.leafNodeIndex = INVALID_INDEX;
        rayGeometryCollisionsBuffer.data[gID].collisions[0] = initialCollision;

        Intersection intersection = defaultIntersection();

        // For each ray in the queue of rays to process
        while(rayQueue.current < rayQueue.end)
        {
        Ray currentRay = rayGeometryCollisionsBuffer.data[gID].collisions[rayQueue.current].ray;

        // Find the closest intersection between the ray and the environment
        intersection = findClosestIntersection(
        currentRay,
        FLT_MIN,
        FLT_MAX);

        if(intersection.isFound)
        {
        if(!intersection.isDestination && rayQueue.end < MAX_RAYS)
        {
                // If the ray has not reached it's destination, and the maximum number of rays has
not been computed,
                // reflect the ray around the intersection normal and add the reflected ray to the
queue of rays to process
                Ray reflectedRay;
                reflectedRay.origin = currentRay.origin + (currentRay.direction * intersection.t);
                reflectedRay.direction = reflectRay(currentRay.direction, intersection.normal);
                reflectedRay.totalDistance = currentRay.totalDistance + intersection.t;

                // Move the reflected ray origin along the intersection normal to avoid incorrectly
intersecting the same object
                reflectedRay.origin = reflectedRay.origin + (intersection.normal * 0.0001);

                // Set the leaf node index where the collision was found (For prediction)
                rayGeometryCollisionsBuffer.data[gID].collisions[rayQueue.current].leafNodeIndex =
intersection.leafNodeIndex;

                // Initialise the next ray to be tested
```

```
                RayGeometryCollision nextCollision;
                nextCollision.ray = reflectedRay;
                nextCollision.leafNodeIndex = INVALID_INDEX;
                rayGeometryCollisionsBuffer.data[gID].collisions[rayQueue.end] = nextCollision;

                rayQueue.end++;
        }
        else if(intersection.isDestination)
        {
                // Destination found
                break;
        }
        }

        ++rayQueue.current;
        }

        outputBuffer.data[gID].intersectionFound = intersection.isDestination;
        outputBuffer.data[gID].finalRay =
rayGeometryCollisionsBuffer.data[gID].collisions[rayQueue.current].ray;

        //////////////////////
        // Debugging Output //
        //////////////////////
        if(rayQueue.current == rayQueue.end)
        {
        --rayQueue.current;
        }

        outputBuffer.data[gID].intersectionT = intersection.t;
        outputBuffer.data[gID].intersectionNormal = intersection.normal;
        outputBuffer.data[gID].audioSourcePosition =
pushConstants.audioComputeInput.audioSourcePosition;
        outputBuffer.data[gID].audioDirection = direction;

        for(int i = 0; i < MAX_RAYS; ++i)
        {
        outputBuffer.data[gID].rays[i] = rayGeometryCollisionsBuffer.data[gID].collisions[i].ray;
        }
}
```

6.1.18 Ray Intersection Prediction Updated Closest Intersection GLSL Code

```
/**
 * @brief Find the closest intersection for a ray
 *
 * @param ray The ray to test
 * @param minT The minimum distance along the ray which can intersect
 * @param maxT The maximum distance along the ray which can intersect
 *
 * @returns The closest intersection, if found
 */
Intersection findClosestIntersection(Ray ray, float minT, float maxT)
{
        Intersection closestIntersection = defaultIntersection();

        float sphereRadius = 1.0;
        Intersection sphereIntersection = testRaySphereIntersection(
        ray,
        pushConstants.audioComputeInput.listenerPosition,
        sphereRadius,
```

```
        minT,
        maxT);

        if(sphereIntersection.t < closestIntersection.t)
        {
        closestIntersection.isFound = true;
        closestIntersection.isDestination = true;
        closestIntersection.t = sphereIntersection.t;
        closestIntersection.normal = sphereIntersection.normal;
        }

        ////////////////////////////////////////////
        ///////////////// Prediction /////////////////
        ////////////////////////////////////////////
        int x = hashRayOrigin(ray, geometryKdTreeNodeBuffer.data[0].aabb);
        int z = hashRayDirection(ray);

        bool madePrediction = false;
        Intersection predictedClosestIntersection = defaultIntersection();
        int predictedKdNodeIndex = rayGeometryCollisionPredictionBuffer.data[x].directions[z];
        if(predictedKdNodeIndex != INVALID_INDEX)
        {
        madePrediction = true;
        #ifdef DEBUG_OUTPUT
        uint gID = gl_GlobalInvocationID.x;
        outputBuffer.data[gID].predictions++;
        #endif

        KdNode predictedNode = geometryKdTreeNodeBuffer.data[predictedKdNodeIndex];
        for(int i = predictedNode.indicesStart; i < predictedNode.indicesEnd; ++i)
        {
        Intersection intersection = testRayTriangleIntersection(
                ray,

geometryVertexBuffer.data[geometryIndexBuffer.data[geometryKdTreeLeafNodeIndexBuffer.data[i] +
0]].position,

geometryVertexBuffer.data[geometryIndexBuffer.data[geometryKdTreeLeafNodeIndexBuffer.data[i] +
1]].position,

geometryVertexBuffer.data[geometryIndexBuffer.data[geometryKdTreeLeafNodeIndexBuffer.data[i] +
2]].position,
                minT,
                maxT);

        if(intersection.t < predictedClosestIntersection.t)
        {
                predictedClosestIntersection.isFound = true;
                predictedClosestIntersection.isDestination = false;
                predictedClosestIntersection.t = intersection.t;
                predictedClosestIntersection.normal = intersection.normal;
                predictedClosestIntersection.leafNodeIndex = predictedKdNodeIndex;
        }
        else if(intersection.t == predictedClosestIntersection.t)
        {
                predictedClosestIntersection.normal = (intersection.normal +
predictedClosestIntersection.normal) / 2;
        }
        }

        #ifndef DEBUG_OUTPUT
        if(predictedClosestIntersection.isFound == true &&
                predictedClosestIntersection.t < closestIntersection.t)
```

```
        {
                return predictedClosestIntersection;
        }
        #endif
        }
        ////////////////////////////////////////////


        ////////////////////////////////////////////
        ////////////// KD Tree Traversal //////////////
        ////////////////////////////////////////////
        int nodeStack[50];
        int nodeStackSize = 1;
        nodeStack[0] = 0;

        while(nodeStackSize > 0)
        {
        // Pop top KD tree node off of the stack
        int currentKdNodeIndex = nodeStack[nodeStackSize-1];
        KdNode currentNode = geometryKdTreeNodeBuffer.data[currentKdNodeIndex];
        nodeStackSize--;

        // Test AABB intersection
        bool aabbIntersection = testRayAabbIntersection(ray, currentNode.aabb);
        if(aabbIntersection)
        {
        // If inner node, add children to the stack of nodes to process
        if(currentNode.nearChild != INVALID_INDEX && currentNode.farChild != INVALID_INDEX)
        {
                nodeStack[nodeStackSize] = currentNode.farChild;
                nodeStackSize++;
                nodeStack[nodeStackSize] = currentNode.nearChild;
                nodeStackSize++;
        }
        // Leaf node, test for geometry intersection
        else
        {
                for(int i = currentNode.indicesStart; i < currentNode.indicesEnd; ++i)
                {
                Intersection intersection = testRayTriangleIntersection(
                        ray,

geometryVertexBuffer.data[geometryIndexBuffer.data[geometryKdTreeLeafNodeIndexBuffer.data[i] +
0]].position,

geometryVertexBuffer.data[geometryIndexBuffer.data[geometryKdTreeLeafNodeIndexBuffer.data[i] +
1]].position,

geometryVertexBuffer.data[geometryIndexBuffer.data[geometryKdTreeLeafNodeIndexBuffer.data[i] +
2]].position,
                        minT,
                        maxT);

                if(intersection.t < closestIntersection.t)
                {
                        closestIntersection.isFound = true;
                        closestIntersection.isDestination = false;
                        closestIntersection.t = intersection.t;
                        closestIntersection.normal = intersection.normal;
                        closestIntersection.leafNodeIndex = currentKdNodeIndex;
                }
                else if(intersection.t == closestIntersection.t)
                {
```

```
                        closestIntersection.normal = (intersection.normal +
closestIntersection.normal) / 2;
                }
                }
        }
        }
        }
        ////////////////////////////////////////////

        #ifdef DEBUG_OUTPUT
        if(madePrediction)
        {
        if(predictedClosestIntersection.leafNodeIndex != closestIntersection.leafNodeIndex)
        {
                uint gID = gl_GlobalInvocationID.x;
                outputBuffer.data[gID].predictionMisses++;
        }
        }
        #endif

        return closestIntersection;
}
```

6.1.19 Ray Intersection Prediction Updated To Populate Ray Collision Buffer GLSL Code

```
/**
 * @brief Shader entry point
 */
void main()
{
        uint gID = gl_GlobalInvocationID.x;
        #ifdef DEBUG_OUTPUT
        outputBuffer.data[gID].successfulGeometryIntersectionTests = 0;
        outputBuffer.data[gID].totalGeometryIntersectionTests = 0;
        outputBuffer.data[gID].successfulAabbIntersectionTests = 0;
        outputBuffer.data[gID].totalAabbIntersectionTests = 0;
        outputBuffer.data[gID].predictions = 0;
        outputBuffer.data[gID].predictionMisses = 0;
        #endif

        float angle = radians(360.0 / AUDIO_DIRECTION_COUNT);
        vec3 direction = vec3(0.0, 0.0, 1.0);
        for(int i = 0; i < gID; ++i)
        {
        direction = vec3(
        (direction.x * cos(angle)) - (direction.z * sin(angle)),
        0.0,
        (direction.z * cos(angle)) + (direction.x * sin(angle))
        );
        }

        Ray initialRay;
        initialRay.origin = pushConstants.audioComputeInput.audioSourcePosition;
        initialRay.direction = direction;
        initialRay.totalDistance = 0.0;

        RayQueue rayQueue;
        rayQueue.current = 0;
        rayQueue.end = 1;

        RayGeometryCollision initialCollision;
```

```
        initialCollision.ray = initialRay;
        initialCollision.leafNodeIndex = INVALID_INDEX;
        rayGeometryCollisionsBuffer.data[gID].collisions[0] = initialCollision;

        Intersection intersection = defaultIntersection();

        // For each ray in the queue of rays to process
        while(rayQueue.current < rayQueue.end)
        {
        Ray currentRay = rayGeometryCollisionsBuffer.data[gID].collisions[rayQueue.current].ray;

        // Find the closest intersection between the ray and the environment
        intersection = findClosestIntersection(
        currentRay,
        FLT_MIN,
        FLT_MAX);

        if(intersection.isFound)
        {
        if(!intersection.isDestination && rayQueue.end < MAX_RAYS)
        {
                // If the ray has not reached it's destination, and the maximum number of rays has
not been computed,
                // reflect the ray around the intersection normal and add the reflected ray to the
queue of rays to process
                Ray reflectedRay;
                reflectedRay.origin = currentRay.origin + (currentRay.direction * intersection.t);
                reflectedRay.direction = reflectRay(currentRay.direction, intersection.normal);
                reflectedRay.totalDistance = currentRay.totalDistance + intersection.t;

                // Move the reflected ray origin along the intersection normal to avoid incorrectly
intersecting the same object
                reflectedRay.origin = reflectedRay.origin + (intersection.normal * 0.0001);

                // Set the leaf node index where the collision was found (For prediction)
                rayGeometryCollisionsBuffer.data[gID].collisions[rayQueue.current].leafNodeIndex =
intersection.leafNodeIndex;

                // Initialise the next ray to be tested
                RayGeometryCollision nextCollision;
                nextCollision.ray = reflectedRay;
                nextCollision.leafNodeIndex = INVALID_INDEX;
                rayGeometryCollisionsBuffer.data[gID].collisions[rayQueue.end] = nextCollision;

                rayQueue.end++;
        }
        else if(intersection.isDestination)
        {
                // Destination found
                break;
        }
        }

        ++rayQueue.current;
        }

        outputBuffer.data[gID].intersectionFound = intersection.isDestination;
        outputBuffer.data[gID].finalRay =
rayGeometryCollisionsBuffer.data[gID].collisions[rayQueue.current].ray;

        //////////////////////
        // Debugging Output //
        //////////////////////
```

```
        if(rayQueue.current == rayQueue.end)
        {
        --rayQueue.current;
        }

        outputBuffer.data[gID].intersectionT = intersection.t;
        outputBuffer.data[gID].intersectionNormal = intersection.normal;
        outputBuffer.data[gID].audioSourcePosition =
pushConstants.audioComputeInput.audioSourcePosition;
        outputBuffer.data[gID].audioDirection = direction;

        for(int i = 0; i < MAX_RAYS; ++i)
        {
        outputBuffer.data[gID].rays[i] = rayGeometryCollisionsBuffer.data[gID].collisions[i].ray;
        }
}
```

6.1.20 Ray Intersection Prediction Table Generation GLSL Code

```
#version 450


////////////////////////////////////
////////////// Defines /////////////
////////////////////////////////////
#define FLT_MAX 3.402823466e+38
#define FLT_MIN 1.175494351e-38
#define FLT_EPSILON 1.192092896e-07
#define PI 3.1415926538
//#define DEBUG_OUTPUT


////////////////////////////////////
////////////// Constants ///////////
////////////////////////////////////
const int MAX_RAYS = 500;
const int AUDIO_DIRECTION_COUNT = 512;
const int INVALID_INDEX = 4294967295;
const int HASH_GRID_DIMENSIONS = 2048;
const int DIRECTION_HASH_RESOLUTION = 128;


////////////////////////////////////
////////////// Structs /////////////
////////////////////////////////////
struct Ray
{
        vec3 origin;
        vec3 direction;
        float totalDistance;
};

struct AudioComputeGeometryMetadata
{
        int vertexCount;
        int indexCount;
        int isPredictionAvailable;
};

struct AABB
{
```

```
        vec3 position;
        vec3 extents;
};

struct KdNode
{
        AABB aabb;
        int nearChild;
        int farChild;
        int indicesStart;
        int indicesEnd;
};

struct RayGeometryCollision
{
        Ray ray;
        int leafNodeIndex;
};

struct RayGeometryCollisions
{
        RayGeometryCollision collisions[MAX_RAYS];
};

struct RayDirectionPredictions
{
        int directions[4];
};


////////////////////////////////////
////////// Shader Inputs //////////
////////////////////////////////////
layout(set = 0, binding = 2) buffer GeometryMetadataBuffer
{
        AudioComputeGeometryMetadata data;
} geometryMetadataBuffer;

layout(set = 0, binding = 3) buffer GeometryKdTreeNodeBuffer
{
        KdNode data[];
} geometryKdTreeNodeBuffer;

layout(set = 0, binding = 5) buffer RayGeometryCollisionsBuffer
{
        RayGeometryCollisions data[];
} rayGeometryCollisionsBuffer;

layout(set = 0, binding = 6) buffer RayGeometryCollisionPredictionBuffer
{
        RayDirectionPredictions data[];
} rayGeometryCollisionPredictionBuffer;


////////////////////////////////////
////////// Functions //////////
////////////////////////////////////

/**
 * @brief Spatially hash a ray origin into an integer
 *
 * @param ray The ray to hash
 * @param aabb The aabb defining the space with which to hash the ray (treated as an NxN grid)
```

```
 *
 * @returns The hashed origin
 */
int hashRayOrigin(Ray ray, AABB aabb)
{
        int minGridIndex = 0;
        int maxGridIndex = HASH_GRID_DIMENSIONS - 1;
        int gridRange = (maxGridIndex - minGridIndex);

        int minX = int(aabb.position.x - aabb.extents.x);
        int maxX = int(aabb.position.x + aabb.extents.x);
        int xRange = (maxX - minX);

        int minZ = int(aabb.position.z - aabb.extents.z);
        int maxZ = int(aabb.position.z + aabb.extents.z);
        int zRange = (maxZ - minZ);

        int x = (((int(ray.origin.x) - minX) * gridRange) / xRange) + minGridIndex;
        int z = (((int(ray.origin.z) - minZ) * gridRange) / zRange) + minGridIndex;

        return (z * HASH_GRID_DIMENSIONS) + x;
}

/**
 * @brief Hash a ray direction into an integer describing one of N directions
 *
 * @param ray The ray to hash
 *
 * @returns The hashed direction
 */
int hashRayDirection(Ray ray)
{
        float stepRadians = (2 * PI) / DIRECTION_HASH_RESOLUTION;
        float hashAngle = stepRadians;
        for (int i = 0; i < DIRECTION_HASH_RESOLUTION; ++i)
        {
        float vAngle = atan(ray.direction.z, ray.direction.x);
        if (vAngle < 0)
        {
        vAngle += 2 * PI;
        }
        if (vAngle <= hashAngle)
        {
        return i;
        }

        hashAngle += stepRadians;
        }
}

/**
 * @brief Shader entry point
 */
void main()
{
        for(int i = 0; i < AUDIO_DIRECTION_COUNT; ++i)
        {
        for(int j = 0; j < MAX_RAYS; ++j)
        {
        RayGeometryCollision collision = rayGeometryCollisionsBuffer.data[i].collisions[j];
        if(collision.leafNodeIndex == INVALID_INDEX)
        {
                break;
```

```
        }

        int x = hashRayOrigin(collision.ray, geometryKdTreeNodeBuffer.data[0].aabb);
        int z = hashRayDirection(collision.ray);

        rayGeometryCollisionPredictionBuffer.data[x].directions[z] = collision.leafNodeIndex;
        }
        }

        geometryMetadataBuffer.data.isPredictionAvailable = 1;
}
```

6.1.21 Ray Intersection Prediction Buffer Creation GLSL Code

```
const uint32_t rayGeometryCollisionsBufferSize = GPU_AUDIO_THREAD_COUNT *
sizeof(RayGeometryCollisions);
rayGeometryCollisionsBuffer = VulkanBuffer::Builder()
        .withDevice(device.get())
        .withBufferSize(rayGeometryCollisionsBufferSize)
        .withBufferUsageFlags(VK_BUFFER_USAGE_STORAGE_BUFFER_BIT)
        .withMemoryPropertyFlags(VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
VK_MEMORY_PROPERTY_HOST_COHERENT_BIT)
        .build();
computeGeometryDescriptorSet->updateFromBuffer(rayGeometryCollisionsBuffer.get(), 0,
rayGeometryCollisionsBufferSize, 5, VK_DESCRIPTOR_TYPE_STORAGE_BUFFER, device.get());

const uint32_t rayGeometryCollisionPredictionBufferSize = RAY_GEOMETRY_PREDICTION_COUNT *
sizeof(RayDirectionPredictions);
rayGeometryCollisionPredictionBuffer = VulkanBuffer::Builder()
        .withDevice(device.get())
        .withBufferSize(rayGeometryCollisionPredictionBufferSize)
        .withBufferUsageFlags(VK_BUFFER_USAGE_STORAGE_BUFFER_BIT)
        .withMemoryPropertyFlags(VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
VK_MEMORY_PROPERTY_HOST_COHERENT_BIT)
        .build();
computeGeometryDescriptorSet->updateFromBuffer(rayGeometryCollisionPredictionBuffer.get(), 0,
rayGeometryCollisionPredictionBufferSize, 6, VK_DESCRIPTOR_TYPE_STORAGE_BUFFER, device.get());

// Initialize predictor table data with invalid leaf node IDs, indicating no predictions
std::vector<RayDirectionPredictions> predictionTable(RAY_GEOMETRY_PREDICTION_COUNT);
for (int i = 0; i < predictionTable.size(); ++i)
{
        for (int j = 0; j < math::DIRECTION_HASH_RESOLUTION; ++j)
        {
        predictionTable[i].directions[j] = std::numeric_limits<std::uint32_t>::max();
        }
}

auto const stagingBuffer = VulkanBuffer::Builder()
        .withDevice(device.get())
        .withBufferSize(rayGeometryCollisionPredictionBufferSize)
        .withBufferUsageFlags(VK_BUFFER_USAGE_TRANSFER_SRC_BIT)
        .withMemoryPropertyFlags(VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
VK_MEMORY_PROPERTY_HOST_COHERENT_BIT)
        .build();
stagingBuffer->update(predictionTable.data(), rayGeometryCollisionPredictionBufferSize);

auto const commandBuffer = VulkanCommandBuffer::Builder()
        .withDevice(device.get())
        .withCommandPool(graphicsCommandPool.get())
        .withLevel(VK_COMMAND_BUFFER_LEVEL_PRIMARY)
```

```
        .build();

stagingBuffer->copyToBuffer(commandBuffer.get(), device->getGraphicsQueue().get(),
rayGeometryCollisionPredictionBuffer.get(), rayGeometryCollisionPredictionBufferSize);

VkCommandBuffer const commandBufferHandle = commandBuffer->getCommandBuffer();
vkFreeCommandBuffers(device->getLogicalDevice(), graphicsCommandPool->getCommandPool(), 1,
&commandBufferHandle);
```

6.1.22 Ray Spatial Hashing GLSL Code

```
/**
 * @brief Spatially hash a ray origin into an integer
 *
 * @param ray The ray to hash
 * @param aabb The aabb defining the space with which to hash the ray (treated as an NxN grid)
 *
 * @returns The hashed origin
 */
int hashRayOrigin(Ray ray, AABB aabb)
{
        int minGridIndex = 0;
        int maxGridIndex = HASH_GRID_DIMENSIONS - 1;
        int gridRange = (maxGridIndex - minGridIndex);

        int minX = int(aabb.position.x - aabb.extents.x);
        int maxX = int(aabb.position.x + aabb.extents.x);
        int xRange = (maxX - minX);

        int minZ = int(aabb.position.z - aabb.extents.z);
        int maxZ = int(aabb.position.z + aabb.extents.z);
        int zRange = (maxZ - minZ);

        int x = (((int(ray.origin.x) - minX) * gridRange) / xRange) + minGridIndex;
        int z = (((int(ray.origin.z) - minZ) * gridRange) / zRange) + minGridIndex;

        return (z * HASH_GRID_DIMENSIONS) + x;
}

/**
 * @brief Hash a ray direction into an integer describing one of N directions
 *
 * @param ray The ray to hash
 *
 * @returns The hashed direction
 */
int hashRayDirection(Ray ray)
{
        float stepRadians = (2 * PI) / DIRECTION_HASH_RESOLUTION;
        float hashAngle = stepRadians;
        for (int i = 0; i < DIRECTION_HASH_RESOLUTION; ++i)
        {
        float vAngle = atan(ray.direction.z, ray.direction.x);
        if (vAngle < 0)
        {
        vAngle += 2 * PI;
        }
        if (vAngle <= hashAngle)
        {
        return i;
        }
```

```
        hashAngle += stepRadians;
        }
}
```

6.1.23 Ray Prediction Table Generator Pipeline Creation C++ Code

```
predictionPipeline = VulkanPipeline::ComputeBuilder()
        .withDevice(device.get())
        .withDescriptorSetLayouts({ computeGeometryDescriptorSetLayout.get() })
        .withPushConstantRange(VK_SHADER_STAGE_COMPUTE_BIT, 0, sizeof(AudioComputeInput))
        .withShaderModule(std::move(VulkanShaderModule::Builder()
        .withDevice(device.get())
        .withShaderCode(shaderCode)
        .build()))
        .build();

predictionFence = VulkanFence::Builder()
        .withDevice(device.get())
        .build();
```

6.1.24 Ray Prediction Table Generator Pipeline Execution C++ Code

```
startTimer(std::format("record-prediction-commands-{}", eventName));
computeCommandBuffer->begin(VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT);
predictionPipeline->bind(computeCommandBuffer.get(), VK_PIPELINE_BIND_POINT_COMPUTE);

std::vector<VkDescriptorSet> const predictionDescriptorSetGroup = {
        computeGeometryDescriptorSet->getDescriptorSet()
};
vkCmdBindDescriptorSets(
        computeCommandBuffer->getCommandBuffer(),
        VK_PIPELINE_BIND_POINT_COMPUTE,
        predictionPipeline->getPipelineLayout(),
        0,
        predictionDescriptorSetGroup.size(),
        predictionDescriptorSetGroup.data(),
        0,
        nullptr);

vkCmdDispatch(computeCommandBuffer->getCommandBuffer(), 1, 1, 1);
computeCommandBuffer->end();
endTimer(std::format("record-prediction-commands-{}", eventName));

VkFence predictionFenceHandle = predictionFence->getFence();
vkResetFences(device->getLogicalDevice(), 1, &predictionFenceHandle);

startTimer(std::format("generate-prediction-table-{}", eventName));
device->getComputeQueue()->submit(nullptr, nullptr, predictionFence.get(),
computeCommandBuffer.get(), VK_PIPELINE_STAGE_NONE);
vkWaitForFences(device->getLogicalDevice(), 1, &predictionFenceHandle, VK_TRUE,
std::numeric_limits<uint64_t>::max());
endTimer(std::format("generate-prediction-table-{}", eventName));
```