Scalable Immersive Audio for Virtual Environments

By

Mirza Beig

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Science in Computer Science

University of Ontario Institute of Technology

Supervisors:

Dr. Bill Kapralos and Dr. Pejman Mirza-Babaei

May 2018
Copyright © Mirza Beig, 2018
Abstract

This thesis discusses the history of game sound and how it developed to become increasingly more immersive and realistic by accounting for the spatial aspects of sound. It then compares several 3D sound technologies before continuing on to present a custom spatial sound system that uses the highly parallel nature of modern graphics processing units (GPUs) to process large amounts of data for simulated sound propagation. In a real-time virtual reality game environment, the system is able to handle dynamic geometry and movable sound sources in full 3D at interactive rates. To test the implementation, I developed a VR-compatible video game for desktop and mobile platforms with an emphasis on play-by-sound gameplay mechanics. Results demonstrated low overhead on all devices tested when used with the appropriate pathfinding backend (CPU/GPU). Additional testing yielded that the GPU part of the system is capable of processing several thousand 3D paths at interactive rates.
Acknowledgements

I would like to gratefully acknowledge the support of both my parents who helped me not only all the way through my masters, but all throughout my life as well. I would also like to thank my exceptionally helpful supervisors, Dr. Bill Kapralos and Dr. Pejman Mirza-Babaei, who constantly gave me direct input and were available for advice when I needed it. I would not have gotten into this program and had the remarkable support that I did if it wasn’t also for additional professors such as Dr. Andrew Hogue, Dr. Lennart Nacke, Dr. Karen Collins, Dr. Richard Pazzi, and James Robb. I’d also like to thank the committee members Dr. Alvarro Joffre Uribe Quevedo and Dr. Loutfouz Zaman who gave me feedback after reviewing my thesis. I’d like to thank all my friends who made my journey more fun. A special shout-out goes to Naeem Moosajee, who advised me to get involved with doing my masters early on, as well as Albert Bushara, Lucas Lupinski, and Oliver Pollard. They all showed up when I was sick with yummy chocolates. That was pretty cool. Finally, the financial support of the Social Sciences and Humanities Research of Council of Canada (Insight grant led by Dr. Karen Collins) is gratefully acknowledged.
# Table of Contents

Abstract .......................................................................................................................... 2
Acknowledgements ......................................................................................................... 3
Table of Contents ............................................................................................................ 4
List of Figures ................................................................................................................ 8
List of Tables .................................................................................................................. 10
List of Code ................................................................................................................... 11
Acronyms ....................................................................................................................... 12
Chapter 1: Introduction ............................................................................................... 14
  1.1 Overview .................................................................................................................. 14
  1.2 Motivation ............................................................................................................... 14
  1.3 Thesis Statement .................................................................................................... 16
  1.4 Thesis Outline ......................................................................................................... 17
  1.5 Summary ................................................................................................................. 17
Chapter 2: Literature Review ....................................................................................... 19
  2.1 Overview ................................................................................................................ 19
  2.2 Video Games ......................................................................................................... 19
  2.3 The Importance of Game State Feedback ............................................................... 20
  2.4 Localizing Sounds and Establishing Presence in Virtual Environments ............. 22
  2.5 Sound Source Localization: Direction .................................................................... 24
     2.5.1 Overview .......................................................................................................... 24
     2.5.2 Binaural Cues: ITD and ILD ........................................................................... 24
     2.5.3 Monaural Cue: Spectral Cue / HRTFs .............................................................. 26
  2.6 Sound Source Localization: Distance ..................................................................... 27
  2.7 Sound Source Localization: Acuity ......................................................................... 28
  2.8 Sound Source Localization: The Environment ....................................................... 29
     2.8.1 Reflections and Reverb .................................................................................... 29
     2.8.2 Parametric Reverb ......................................................................................... 32
     2.8.3 Convolution Reverb ........................................................................................ 33
     2.8.4 Occlusion, Obstruction, Transmission, and Diffraction .................................. 35
  2.9 Acoustical Modelling .............................................................................................. 37
7.3 Future Work .................................................................................................................. 169
7.4 Conclusions .................................................................................................................. 170
References .......................................................................................................................... 172
List of Figures

Figure 1: The ITD is caused by the difference in time between sound energy reaching the two ears, whereas the ILD is the difference in intensity. ................................................................. 26
Figure 2: The spectral cue mapped using HRTFs. Notice the subtle differences in the frequency of the sound between front and back positioning (using the Oculus Spatializer VST in FLStudio)...................... 27
Figure 3: A visualization of the Doppler effect, where sound waves ahead of the object are compressed and the reverse happens for sound waves opposite the direction of motion. .................................................. 28
Figure 4: Sound waves propagating as direct and reflected sound energy................................................. 30
Figure 5: A virtual image of a sound source created through the reflection of sound energy. .................. 31
Figure 6: Parametric reverb as seen in FLStudio (top image) and Unity (bottom image). ......................... 33
Figure 7: Impulse response waveform and spectrum from the "Fruity Convolver" plugin in FLStudio. ....... 34
Figure 8: The impulse spectrum, covering a broad range of frequencies ............................................... 35
Figure 9: Partial occlusion (obstruction) produces an area of decreased intensity known as a "sound shadow". The sound energy colliding with the obstruction is partially reflected and absorbed. Note that diffraction, a wave effect, is not demonstrated in the image. In reality, when the wavelength of propagated sound is greater than the width of an obstructing surface, sound waves will diffract, or “bend” into the shadow region. ........................................................................................................ 37
Figure 10: Virtual volume rolloff curve visualization in the Unity game engine, where X is the distance to the listener, and Y is the volume. ........................................................................................ 39
Figure 11: EAX as an option in the PC/Windows version of Halo: Combat Evolved [77]. ....................... 54
Figure 12: Surround sound support as an optional feature in Kingdom Hearts 2 [86] for the PlayStation 2. ................................................................................................................................. 56
Figure 13: A simple scene rendered real-time in Unity with and without high-quality baked lighting. The bottom image features only direct illumination. The top image features pre-computed global illumination, where propagated light diffusion is apparent throughout the scene (including ambient lighting provided by a procedural day time skybox and colour bleeding by the emissive blue cube)..... 61
Figure 14: A scene from the 1st District of "Traverse Town" in Kingdom Hearts, featuring many false light sources (this scene is unlit, and lighting is baked into the textures). ..................................................... 69
Figure 15: Direct path rendering with no geometry (red line). Source: A3D 2.0 Technology Brief [81]. ... 69
Figure 16: Simple occlusion with a single occluding wall between the source and listener (faint red line). Source: A3D 2.0 Technology Brief [81]. ......................................................................................... 70
Figure 17: Simple reflection case against three walls (green lines) without any occlusion. Source: A3D 2.0 Technology Brief [81]. ........................................................................................................... 71
Figure 18: Simple occlusion/reflection with two walls reflecting the sound (green lines) and another occluding the direct path (faint red line). Source: A3D 2.0 Technology Brief [81]. ......................... 72
Figure 19: More complex reflection/occlusion with six walls reflecting the sound (green lines), where one wall’s reflection is occluded (faint green lines). Source: A3D 2.0 Technology Brief [81]. ................. 73
Figure 20: EAX reverb presets defined in the EAX.H header file. Source: EAX v1.0 User Manual [133]. ... 76
Figure 21: A "complex" 3D level rendered real-time in Unity where each sound source (represented as a loudspeaker/megaphone icon) is also an offline point light rendered to a texture. .......................... 90
Figure 22: A high-quality baked lighting texture, known as a "lightmap". It's possible to sample the grayscale values to use as an attenuation factor for sound occlusion if each light source is positioned with a fixed sound source. .......................................................... 92

Figure 23: An orthographic top-down view from an area of my game in Unity's editor. Using pathfinding, the sound will appear to come through the doorway, rather than from behind the player where it actually is – a rough approximation of diffraction (red line) and reflected (blue line) sound. ..................... 94

Figure 24: A high-level flow diagram showing the overall structure of my spatial audio system. NavMesh obstacles are handled directly by Unity's system, whereas GPU pathfinder obstacles update node states on the CPU which are then re-sent to the GPU. .......................................................... 100

Figure 25: The GPU pathfinder component as seen configured in Unity's editor in the performance benchmarking scene. ........................................................................... 102

Figure 26: The "DSP processor" component as seen in Unity's editor attached to an audio source. ..................... 122

Figure 27: Unity's built-in profiler. ........................................................................... 134

Figure 28: A sample profiler output with all the recorded and calculated measurements. ..................... 142

Figure 29: The pathfinder manager in-game UI used for touch/mouse-based control over the global spatial audio pathfinding settings. ........................................................................... 143

Figure 30: My benchmarking level in Unity's editor with 10 sound sources. Sound sources are visible as small solid spheres inside larger, yellow-tinted wireframe spheres. ......................................................... 144

Figure 31: Part of the level in which most of the game takes place. .......................................................... 153

Figure 32: The fully activated teleportation monolith. ........................................................................... 154

Figure 33: Interacting with a physics-based object (small crate). ........................................................................... 155

Figure 34: The game's touchscreen control system, featuring a move pad and interaction buttons. .......... 156

Figure 35: Being pursued by the enemy AI. ................................................................................................. 160

Figure 36: A visualization of Unity's NavMesh on the second level of my game (highlighted above the scene geometry). ................................................................................................. 163
List of Tables

Table 1: DirectSound and A3D feature comparison. Source: A3D 2.0 Technology Brief [81]. .......................... 68
Table 2: All systems used for benchmarking. ........................................................................................................ 145
Table 3: A summary of all performance tests in the benchmark environment, where \( \text{RT} = \) Runtime Total, \( \text{SA} = \) Spatial Audio. Lower is better (less time to execute equates to higher FPS). .................................. 146
Table 4: A reference performance table measuring total runtimes with only the GPU pathfinder’s path requests. .................................................................................................................................................. 150
Table 5: A summary of all performance tests in the game where \( \text{RT} = \) Runtime Total, \( \text{SA} = \) Spatial Audio. Lower is better (less time to execute equates to a higher FPS). .......................................................... 162
List of Code

Code 1: An implementation of using pre-computed lighting information to control a DSP filter. ........... 91
Code 2: The node struct which is mirrored between the CPU and GPU. ........................................... 103
Code 3: The CreateGrid function called from the GPU pathfinder component on awake. This initializes
the voxel-node graph for pathfinding as a bounding volume in the scene. The basis of this function is
from Sebastian Lague’s 2D A* pathfinding tutorial [174]. .................................................................. 106
Code 4: Start is called after Awake in Unity. The scene graph has already been created at this point and
the compute shader is initialized with buffer/kernel bindings. Many of the elements that will be used on
the CPU are also initialized here. ........................................................................................................ 111
Code 5: The GPU pathfinder update function (called every application frame update). ....................... 116
Code 6: The frontier propagation kernel in the compute shader....................................................... 118
Code 7: The kernel that checks to see if frontier propagation has stopped flowing into new cells of the
graph. .................................................................................................................................................. 118
Code 8: The final kernel in the GPU pathfinder system that sums up path data and records its state. ... 121
Code 9: A double linecast (a form of raycasting defined as a line) to check for obstructions between
source and listener................................................................................................................................. 122
Code 10: If path to listener is blocked, request a path from the current pathfinder. ............................... 123
Code 11: The main DSP processor update function which controls virtual image source projection and
DSP filters............................................................................................................................................... 132
Code 12: The pathfinder manager used to assert global control over the spatial audio pathfinding
configuration. Features an integrated profiler for the system................................................................. 141
Code 13: The enemy AI’s listening behaviour function. ........................................................................ 158
Code 14: The world sound source monitor, used to log global sound source events by other components
(physics-based collisions, player footsteps, etc...). .............................................................................. 159
<table>
<thead>
<tr>
<th>Acronyms</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR</td>
<td>Virtual Reality</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>O&amp;M</td>
<td>Orientation and Mobility</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processing</td>
</tr>
<tr>
<td>NPC</td>
<td>Non-Player Character</td>
</tr>
<tr>
<td>ITD</td>
<td>Interaural Time Difference</td>
</tr>
<tr>
<td>ILD</td>
<td>Interaural Level Difference</td>
</tr>
<tr>
<td>HRTF</td>
<td>Head-Related Transfer Function</td>
</tr>
<tr>
<td>IR</td>
<td>Impulse Response</td>
</tr>
<tr>
<td>UTD</td>
<td>Uniform Theory of Diffraction</td>
</tr>
<tr>
<td>EAX</td>
<td>Environmental Audio Extensions</td>
</tr>
<tr>
<td>EFX</td>
<td>Effects Extension</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Full Form</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
</tr>
<tr>
<td>SDK</td>
<td>Software Development Kit</td>
</tr>
<tr>
<td>BFS</td>
<td>Breadth-First Search</td>
</tr>
<tr>
<td>HLSL</td>
<td>High Level Shading Language</td>
</tr>
<tr>
<td>FPS</td>
<td>Frames Per Second</td>
</tr>
</tbody>
</table>
Chapter 1: Introduction

1.1 Overview

Game visuals have primarily evolved much faster than sound. It is only through the sudden resurgence in virtual reality (VR) technology that we are finally seeing a renewed interest for more immersive audio in video games as a means to establish a greater sense of presence in virtual environments. This chapter will highlight the motivation behind creating a new cross-platform system for approximated sound propagation (part of the contribution of this thesis), as well an outline of subsequent chapters and what they discuss.

1.2 Motivation

Video games account for a large portion of the entertainment market. In 2016, the global games market was estimated to have generated over $100 billion in revenue [1]. Currently, video games and video game-based technologies are seeing a large application in areas beyond entertainment, including training, education, research, and being applied as promotional media for commercial products. Over the years, video games have undergone a rapid evolution as newer and more powerful technology has made way for cutting-edge, real-time interactivity, immersion, and graphical fidelity. In the real-world, we rely on more than just our vision, where sound also plays an important role as a source of information for everyday orientation and mobility. However, sound has often been a neglected aspect of game development and games in general, and is only just beginning to make major advances given the recent increasing popularity with VR and the desire to increase immersion and engagement.
Many topics related to game development can benefit from increased presence through *spatial sound*, which provides additional localization and environmental cues to better represent the virtual environment in which sound sources are simulated. Some of these topics include serious games for health and rehabilitation, such as orientation and mobility (O&M) training for the visually impaired, exposure therapy, and simulations for high-risk occupations (pilots, surgeons, and firefighters, amongst others). As an example, Seki and Sato developed a virtual reality training simulation for blind individuals that employs spatial sound [2]. Seki and Sato’s system realized the head-related transfer function (HRTF), that is, the location-dependent filtering of sound (primarily by the pinna), and environmental cues such as reflection and occlusion, to aid in the disambiguation of not only sound source localization, but also obstacle perception (reflections were modelled without energy loss against walls and sounds were insulated completely when occluded by walls). Their results suggested that some O&M metrics were as good as or better than traditional training methods that rely on experiential auditory training in real-world environments (as opposed to virtual simulations). While spatialized sound for games is becoming increasingly accessible through third-party audio middleware and integration into popular game engines, additional processing for environmental cues such as occlusion and propagation with dynamic geometry is still computationally expensive for real-time use. One potential and mostly untapped source for greater immersive audio in video games is the use of general-purpose GPU (GPGPU) computing to take advantage of today’s powerful massively parallel desktop graphics processors to assist in the approximation of fully 3D sound propagation in real-time while still taking advantage of the central processing unit (CPU) for “lighter” implementations of similar techniques. Although the GPU is, at its core, a specialized co-
processor for computer graphics rendering, GPGPU has been applied to accelerate computationally exhaustive tasks such as fluid dynamics, physically animated hair, particle simulations [3] [4], digital signal processing (DSP) for sound [5], acoustical occlusion/diffraction modelling for 2D virtual environments [6] and more.

1.3 Thesis Statement

This thesis provides a comparison between current and previous implementations/approximations of 3D spatialized sound in video games. Through identifying the features and trends of the most recent spatial sound technologies, I created a new system that draws on the history of game sound and the power of modern hardware to produce a new method of approximating sound propagation in video games using 3D pathfinding with dynamic geometry and movable source/listener pairs. As a demonstration of how the system is capable of performing at interactive rates and its scalability across a wide range of hardware, I created a complete cross-platform game developed using the Unity game engine that uses my new system and strongly emphasizes play-by-sound mechanics. The system is capable of exploiting the massively parallel nature of modern GPUs and faster host-device transfer rates to pre-process large amounts of 3D level data (stored as voxels) which is then interpreted by the CPU for digital signal processing applying a low-pass filter and parametric reverb to simulate sound occlusion, diffraction, and reflection. As a fallback for lower-end hardware, the system can revert to handling 2.5D data using only the CPU where the pathfinding is performed using a navigation mesh (or “NavMesh”) projected over the surface of the scene geometry.
1.4 Thesis Outline

In this chapter I discussed the motivation behind the contributions of the thesis. Chapter Two provides a review of video games and highlights the use of feedback to aid the player in making intelligent and informed decisions. It then goes on to focus on how sound is perceived by humans to localize sources and establish presence, as well as how sound interacts physically with the environment in the real-world as opposed to approximations used in video games (which is further expanded upon in another chapter). Chapter Three summarizes the most important changes to game sound throughout the history of video games and provides a high-level overview of emerging trends as technology evolved. In Chapter Four, I analyze and compare the features of some of the most popular implementations and technology behind spatialized 3D sound against each other, as well as how they compare to real-world acoustics. Drawing on the analysis and information presented in Chapters One to Four, Chapter Five introduces a new system for scalable, cross-platform approximation of sound propagation using pathfinding. Chapter Six discusses the design and gameplay mechanics of the cross-platform (mobile, PC, VR) game I developed to provide a demonstration of my new system capable of running on a wide range of hardware. Finally, Chapter Seven delivers some conclusions and discussions while detailing some of the possibilities and benefits of future work.

1.5 Summary

This introductory chapter has provided an overview of the motivation behind the contributions of the thesis as well as an outline of the content to follow. I have touched upon the largely untapped use of today’s powerful GPU hardware for the general-purpose processing of 3D audio and the need for more immersive sound in video games – especially those involving VR. Although
third-party solutions for spatialized 3D audio are becoming increasingly prevalent, real-time environmental spatialization involving dynamic geometry is still computationally expensive, creating a noticeable gap in many of these existing implementations. One of the highlighted outcomes of this thesis is a new system that is capable of rendering an approximation of sound propagation using pathfinding either fully in 3D for high-end systems (by exploiting the highly parallel nature of modern desktop GPUs) or in 2.5D for lower-end hardware (using only the CPU).
Chapter 2: Literature Review

2.1 Overview

In this chapter, I review video games and how players commonly interact with virtual environments. I then discuss the importance of sound for game state feedback and how it can be used to add depth to virtual environments. Lastly, I cover how humans perceive sounds and decode directionality, as well as how sound behaves in real-world environments and some of the similarities it possess to light.

2.2 Video Games

Play is any activity that is done primarily for recreation and enjoyment, rather than for any serious or practical purpose [7]. Games are an organized form of play, and since this is subjective, any activity with rules could be a “game” [8]. Playing checkers, snakes and ladders, soccer, tennis, bowling, and billiards are all examples of activities that are games. Games have been part of human history for a very long time, and are about as old as civilization itself [8]. A blurred transition begins to develop from the real-world to an artificial one when the domain in which these games are played becomes increasingly synthetic. An example of this is comparing soccer, a loose mechanical variation of the sport such as table soccer (foosball), and then soccer video games.

Video games can be loosely defined as games represented visually on a continuously updated video display that respond to player input [9], although exceptions exist for games based entirely on sound, even though they may still be referred to as “video” games. Through the evolution and
advancement of increasingly more powerful hardware, video games circumvent and overcome the limitation of complexity that mechanical games face. Although the input into the artificial domain is still limited (gamepads, keyboard, mice, etc.), the macro-interactions and mechanics in modern video games can be as diverse and arbitrarily complex as they need to be, up to the point where it’s possible to simulate entirely autonomous and reactive worlds. Unlike mechanical games, simple external input can map to some very complex interactions internally within the game (e.g., conversing with non-player characters (NPCs), moving objects in three dimensions, and solving elaborate puzzles), given that computerized games can easily account for the context of the input based on the game state up and into many deeper levels of interaction, even if the same few inputs from the player may be used throughout the lifetime of the game session. While the thumbstick of a gamepad may control player movement in one sub-state of a game’s input system, in another it may be used to rotate and organize the elements of a puzzle piece.

2.3 The Importance of Game State Feedback

The vast majority of videos games are multi-sensory displays with two or more channels used to provide game state feedback to players. The concept of states extends to the feedback that a game provides both visually and aurally (i.e. through sound), as this is usually the primary means by which a player continues to interact and respond intelligently to the two. Simple haptics (to simulate the sense of touch), and other sensory channels are also occasionally used, but the majority of games are designed such that the audio-visual feed is the most reliable and accurate source of information about the game state. By seeing and hearing how their input changes the game state, players are able to continue engaging with the artificial domain where the game persists. Although providing a higher volume of information is not necessarily better, accurately
conveying the state of the game with regards to the context of the experience that the game is meant to provide affords the player better, and more informed choices to make without inducing sensory overload, which may otherwise hinder player focus and processing [10]. When video games provide intentionally deceptive feedback (e.g., as many horror games often do using misdirection), it may not always be viewed as “unfair” because it would fall into the context of what the game is trying to provide players as an experience. While the video game software can easily provide accurate information to players for them to react to, such as the location of puzzle items, competing players behind corridors, enemies, etc., doing so could fall outside the context of giving the player a challenge, or guiding them through some engineered experience (horror, absurd, realistic, etc.).

*Unintentional* deceptive feedback can also result in misinformed choices made in video games. An example would be errors in sound localization due to low fidelity spatial audio processing in a first-person shooter. While most video games in this genre have stereo and distance spatialization of audio positioned in a 3D virtual environment, they often lack detailed processing for environmental obstruction and propagation for sounds. Examples of such games include Doom [11], and Homefront [12]. Here, the intent of the developers isn’t to specifically have unrealistic acoustics in their game, but several reasons may hold them back from investing time and resources into a better implementation. Some of these reasons may include limited computational resources, lack of knowledge on how to best add such features, or the belief that it’s “just not that important” for that particular game (which may or may not be true). This can lead to situations where a player may hear a sound source as being next to them without any obstruction, even though that sound may actually be on the other side of a wall, or worse, several/
walls, with only a fall-off based on distance to determine how loud it should be. An example of this is in Left 4 Dead 2 [13]. In this case, the player may even adapt to these cues and such “broken” feedback can actually be useful in figuring out the location of otherwise unseen hostile units as a form of built-in cheating. Unfortunately, this can also lead to a decrease in immersion given its unrealistic nature if the intent of the game developers is to provide a realistic sensory experience. Unsurprisingly, situations like this where players are thrust into an improperly designed auditory interface can be described as having a “virtual” hearing impairment [14].

Multi-sensory displays, such as those commonly used for most video games, can be characterized as either complementary, conflicting, or redundant [15]. The example of the audio-obstructing wall given earlier can be categorized as conflicting due to the expectation from what the player sees visually, as opposed to what they actually hear. VR-based games will suffer from the breaking of immersion in a scenario such as this, as will most training simulators and serious games that typically strive to provide highly immersive virtual environments. If the interface is complimentary, then alongside seeing the wall visually, the player will hear the indirect propagation of sound around the level geometry, confirming the artificial properties of the game world (such as concrete wall being too thick for direct sound propagation) and providing more realism to the virtual environment.

2.4 Localizing Sounds and Establishing Presence in Virtual Environments

Assuming a “standard” consumer-level desktop/laptop computer system setup with a keyboard, mouse, monitor, and some form of audio output device (headphones, earphones, speakers), feedback is limited to the visual and audio streams output through the monitor and speakers
respectively. Hearing allows us to detect changes in air pressure as audible vibrations which we perceive within a specific range of frequencies; in other words, sounds. The act of being able to determine the position of a sound source is known as auditory localization [16]. By determining the location of the sound source in regards to the azimuth (the horizontal plane), elevation (the vertical plane), and distance, that sound is localized in three dimensional space. With the addition of environmental filters, the sound is then spatialized fully into the environment, as well as directionally for the listener. Spatialized sound, including directional audio and room acoustic cues, plays an important role in establishing presence in auditory virtual environments [17] [18] [19], while a first-person perspective most closely matches real-world navigation.

Supplementary hardware for head tracking may be used for a more immersive virtual reality experience, along with some external haptic devices allow for touch to be another source of perceptual information of the game world similar to physical sensations felt by a person in real-life (analogue to their natural senses, or by way of using a tool such as the white cane for the visually impaired). Many mobile devices and game pads have built-in gyroscopes and programmable vibrators which can be used to implement basic motion tracking and haptic feedback features with low-cost. In the case of game pads, there are often two asymmetrical-sized (left/right) rotating motors which can be controlled individually. Combining the two can be used to convey virtual terrain surface information and directional collision information of in-game events to the user.
2.5 Sound Source Localization: Direction

2.5.1 Overview

There are three primary cues used by humans to localize sounds based on two very different means of analyzing the acoustic waveform [20] [21]: the interaural time difference (ITD), the interaural level difference (ILD), and spectral cues. The first two, the ITD and ILD, are known as “binaural” cues because they relate to the comparison of sounds signals reaching both ears. Counter to this, spectral cues are “monaural” because the information for localization is contained in the analysis of the distribution of frequencies reaching the ear from different locations [16], rather than in a comparison of the signal between both ears.

2.5.2 Binaural Cues: ITD and ILD

Whereas the ITD is the difference in time at which a sound reaches our two ears, the ILD is the difference in intensity (Figure 1). The closer a sound source is to being completely to the side of the head, the greater the ITD and ILD between the two ears since the time at which the signal reaches the ear closest to the source will be less, while the intensity will be greater. As a result, these binaural cues are used for localization on the horizontal plane. However, due to the diminishing effect of creating an “acoustic shadow”, the ILD becomes increasingly less reliable the lower the frequency of a sound signal. Only wavelengths roughly equal to or shorter than the diameter of the head produce a significant ILD as a result of a shadowing effect from blocked sound waves (although even low frequency sounds will yield a noticeable ILD if positioned roughly within arm’s length [21]). To produce an acoustic shadow the obstruction must be larger than the wavelength to prevent longer wavelengths (lower frequency sounds) from “bending” around the obstacle, an effect known as diffraction [16]. Wavelengths shorter than the barrier
(higher frequency sounds) are more easily absorbed and/or reflected. As a result, the obstacle produces a region of decreased intensity opposite to the direction of incoming soundwaves, similar to light casting a shadow behind an opaque object. This can be compared to ripples in a pond colliding with the broad side of a large object, such as a boat. Smaller ripples will simply bounce off (reflect) or be absorbed into the side of the boat, larger ripples will diffract and bend around the boat. As a result, the ILD is more effective at high-frequency localization (discounting near-field sounds), while the ITD is better suited for low-frequency localization (based on behavioural research) [16]. Both are ineffective at localizing elevation (including top/bottom) and front/back sound sources due to the magnitude of the cues being too small. Additionally, a so-called “cone of confusion” generates nearly identical ITD and ILD cues for sound sources located at intermediate distances along the surface of the cone, which makes localization extremely difficult using just these two cues, further leading to the importance of the spectral cue.
Figure 1: The ITD is caused by the difference in time between sound energy reaching the two ears, whereas the ILD is the difference in intensity.

### 2.5.3 Monaural Cue: Spectral Cue / HRTFs

Spectral cues are largely a result of the direction-specific attenuation of particular frequencies primarily by the pinna (outer ear), and to a lesser extent, the head, neck, shoulders and torso [21] [16]. Depending on the incoming direction, the sound that reaches the inside of the ear is modified by the physical geometries of the ear and body, allowing us to determine the elevation and front/back localization of sound sources (Figure 2). The function describing these spectral modifications is known as the head-related transfer function (HRTF) [21]. HRTFs predominantly model the directional-dependent filtering of the pinna (outer ear), and to a lesser degree, the torso and head.
Figure 2: The spectral cue mapped using HRTFs. Notice the subtle differences in the frequency of the sound between front and back positioning (using the Oculus Spatializer VST in FLStudio).

While these cues are useful for determining the direction of an incoming sound in 3D, they do not provide any detailed information on the characteristics of the environment in which the sound source is located, or its distance from the listener (although the spectral cue does provide distance information for sounds positioned less than 1m away).

2.6 Sound Source Localization: Distance

Distance localization cues include the intensity (which generally requires a frame of reference for the sound), initial time delay to the reflection of a sound compared to its direct arrival (requiring a closed and/or sufficiently reflective environment), the ratio of direct sound to reverberant sound, motion parallax, and high frequency attenuation (which generally requires that the sound travel large distances for any noticeable effect) [22] [23]. A velocity cue can also help simulate a
sense of movement for sounds through the use of the Doppler effect. Sounds moving towards a listener will sound higher pitched than the actual emitted frequency due to subsequent waves being closer, thus creating a shorter wavelength. Sounds moving away from the listener will undergo the opposite effect and the perceived frequency will be lower as subsequent waves emitted from the moving source are farther apart (Figure 3).

*Figure 3: A visualization of the Doppler effect, where sound waves ahead of the object are compressed and the reverse happens for sound waves opposite the direction of motion.*

2.7 Sound Source Localization: Acuity

Acuity in sound localization varies; it is most accurate for the azimuth, reasonably accurate for elevation, and less accurate for distance [22]. An important concept to note is that in the real-world, orientation and mobility (tilting your head, moving around, etc.), is a natural part of the localization process which helps to resolve any lingering directional ambiguities. In virtual
environments, head tracking can drastically improve localization and 3D audio [24]. Environmental modelling requires simulation of the reverberant field and any obstruction/occlusion effects. Without these, sounds will be perceived as dry, artificial, and disconnected from the visuals presented in a virtual environment.

2.8 Sound Source Localization: The Environment

2.8.1 Reflections and Reverb

As sound waves propagate through the environment, they reflect and/or are absorbed by the surfaces they encounter. The amount of reflection and absorption is determined by the surface materials. Smooth surfaces such as glass or metal walls are more reflective than rough or uneven surfaces such as soft fabric. Sounds that have an unobstructed path to the listener (direct sounds) will be perceived first, as the path they take is essentially a straight line (the shortest distance), while reflected sounds (indirect sounds) will be delayed (Figure 4). The delay time is based on the distance propagated by the reflected sound waves before they reach the listener. If the delay time between subsequent instances of the sound reaching the listener is sufficiently large, a distinct echo is created which is perceived as a separate event. The echo threshold can vary substantially based on criteria enforced and the individual perceiver (see [25]). If the delay is small enough to the point where several reflections are closely mixed together, the effect is then known as reverb.
Figure 4: Sound waves propagating as direct and reflected sound energy.

Naturally, sounds will reflect many times off of many surfaces, diminishing in intensity as they are absorbed and diffused against surfaces and through the medium they are travelling. The initial reflections, known as “early reflections”, “primary reflections” or “first-order reflections”, will reinforce the original signal while providing cues for distance, direction, and information about the environment relative to the location of the source. As an example, a strong set of first-order reflections will cue the listener on the near proximity of walls. These same reflections also typically create strong projections of the original sound along the incoming angle from the surface relative to the listener (Figure 5). Later reflections that mix together are what form the “late reverberant tail” and the actual reverb effect over time which provides a general sense of space to the environment rather than any specific information [26]. Eventually, as a result of absorption and diffusion, the reflected sound energy will decay and become inaudible.
These same principles are what allow an empty bathroom to have an entirely different acoustic environment than that of a large, mountainous landscape. This presents an interesting scenario for localization, as there may be several directions from which the same sound is heard due to reflection. Research has shown that human listeners will still effectively localize such sounds based on the first perceived incidence, even in the absence of visual cues – a phenomena known as the “precedence effect” [25] [27] [16]. This will be the direct sound if it exists, or else it will be the earliest reflection reaching the listener. It’s important to note, however, that the precedence effect is incumbent upon perceiving multiple signals as part of the same sound. If the delay is too large or if the spectra varies significantly then it will simply be perceived as a different sound entirely and the effect will not apply.
There are generally two types of reverb used in game sound: *parametric reverb* and *convolution reverb* [26] [28].

### 2.8.2 Parametric Reverb

Parametric reverb is used to create a perceptual approximation of a reverberant space (see Figure 6 for examples). By today’s standard, it’s computationally affordable and can easily provide a general sense of space to virtual environments where anechoic sounds would be perceived as dry and artificial (gunshots fired in a video game sewer should probably have some sort of reverb so they don’t seem out of place).
2.8.3 Convolution Reverb

Another type of reverb, known as a convolution reverb, can provide exceptionally accurate environmental modelling. A convolution reverb samples the *impulse response* (IR) from specific real-world locations and then applies it to a signal later [26]. Given a sample input signal played back in an environment (the *impulse*), the resulting reverb and spectral characteristics will be the
impulse *response* (see Figure 7 for the full impulse response, and Figure 8 for a depiction of only the initial impulse spectrum, where the input is an ideally short burst that covers a large frequency range).

*Figure 7: Impulse response waveform and spectrum from the "Fruity Convolver" plugin in FLStudio.*
The IR can then be convolved with any sound signal. In this way, the signal can be made to sound as if it’s actually playing back at the location where the IR was sampled, with the majority of the acoustics of that environment taken into account (see [29] for examples of convolution reverb applied to a dog’s bark based on different IRs). Due to the mathematical processes of convolution involved, it’s much more expensive to model than parametric reverb.

![Image](image.png)

Figure 8: The impulse spectrum, covering a broad range of frequencies

2.8.4 Occlusion, Obstruction, Transmission, and Diffraction

In addition to producing reverb, sound in our natural environment is often obstructed by objects of varying size and can even be occluded entirely by walls, ceilings, and other barriers. This leads back to the model of diffraction, by which sound waves will propagate around obstructions and spread out through openings. Given enough sound energy and a suitable material for an occluding surface between the source and listener, the surface can then act as a medium that
will continue to propagate the sound – a process known as transmission. Denser and thicker materials will require more intense sounds to be able to penetrate through and transfer enough sound energy to be audible as a result of their composition, and vice-versa for lighter, thinner materials. This is the effect of acoustic transmission as sound is propagated through another medium. It’s the difference between hearing a loud radio placed in a thin cardboard gift box as opposed to hearing it through a dense wooden cube. If the obstruction is partial, then an acoustic shadow of decreased sound intensity is created on the side opposite from the incoming soundwaves. Depending on the distance from the obstruction, its size, and the frequency spectrum of the incoming sound, this effect may or may not be noticeable. As mentioned earlier when discussing the ILD binaural cue for localizations, wavelengths shorter or roughly equal to the obstruction will be blocked, while wavelengths of greater length will bend around the object. The wavelength is the distance between consecutive similar parts of a wave, and is inversely related to frequency. The farther the listener is from the obstruction, the less the occlusion will be apparent since the sound waves will diffract back into the area behind the object (the shadow region) as they move on. If the spectrum of a sound source contains audible high-frequency parts, it will be perceived by a listener after being notably attenuated on the opposite side of an obstruction, where the diffraction effect is incumbent on the relative size of the obstruction and wavelengths of the high-frequency bands. This applies to any area of obstruction with edges, including openings in walls such as doors and windows (see Figure 9). Combined with non-point sound sources casting shadows, obstructed sound produces a soft occlusion effect when the line of “sight” between a listener and sound source is lost. This is very similar to how light behaves, except that the wavelength of visible light is much smaller than that of audible sound for humans
such that miniscule light diffraction for non-point sources is often just masked by the penumbra [30].

Figure 9: Partial occlusion (obstruction) produces an area of decreased intensity known as a "sound shadow". The sound energy colliding with the obstruction is partially reflected and absorbed. Note that diffraction, a wave effect, is not demonstrated in the image. In reality, when the wavelength of propagated sound is greater than the width of an obstructing surface, sound waves will diffract, or “bend” into the shadow region.

2.9 Acoustical Modelling

2.9.1 Overview

With sound having been a largely ignored aspect of games over the years, games have traditionally used left/right panning with distance attenuation for limited localization potential. While 3D game audio made many advancements in the mid-to-late 1990’s and early 2000’s with binaural audio, HRTFs, and sound propagation featuring environmental reflection and occlusion
processed by advanced soundcards (and a robust market surrounding them), this trend faded away by the mid 2000’s [24]. With an increase in immersive virtual reality software development and sufficiently available computational power, 3D audio is making a resurgence.

2.9.2 Distance, Velocity, and Directionality

Distance can be modelled for a direct sound path by simply attenuating the intensity based on the distance between the source and listener using a falloff curve. The greater the measured distance, the lower the volume of a sound source that’s perceived by the listener (see Figure 10 for an example of a curve modelling the inverse relationship with a logarithmic roll-off). Depending on the state of the propagation medium (air, dense fog, water, etc.), a noticeable high-frequency attenuation may also occur as an effect of atmospheric absorption, which be applied using a simple low-pass filter. The relative velocity of the sound source and listener determines the overall Doppler effect, which represents the dilation and compression of waves (and therefore, the frequency) as a sound source moves away from- or towards a listener respectively. By employing a set of HRTF transforms and binaural cues for both ears using the source’s position in relation to the listener, a selective set of (left/right) filters may be applied to the signal, allowing for full lateral, front/back, and elevation sound localization. HRTF-based filtering is computationally expensive and only recently has become viable for real-time application [31].
2.9.3 Impulse Response

While IRs sampled from real-world spaces can be used to convolve virtual sound sources, the response generated is specific to the environment in which it was created and the position of the source impulse in the recording space. For interactive virtual environments with non-static sound
sources and a moving listener (a common element of most video games), and scene geometry which may be completely different than what can feasibly be sampled and loaded prior to application runtime (games and virtual environments often contain entirely fictional settings with many dynamic elements), an IR must be generated specifically for these virtual spaces. The IR of a virtual space from an arbitrarily positioned sound source to a listener can be computed through methods that model sound propagation. These methods may be divided into those based on sound field decomposition via geometrical acoustics, and those based on a numerical solution to the underlying wave equation for sound propagation [32] [33] [34].

2.9.4 Wave-Based Acoustics

Both finite and boundary element methods can be used to solve partial differential equations (such as the wave equation) numerically in domains of any kind of geometry by subdividing space into discrete components [35] [36]. At each update tick, the entire domain is solved through all of its elements. As a result, they are computationally insensitive to scene complexity, and instead scale with its physical dimensions (the acoustic simulation space) due to requiring more elements to maintain the subdivision resolution [37]. Finite element methods divide the acoustic simulation space into a 3D volume of discrete components (such as axis-aligned cubes), whereas boundary element methods only require a subdivision of the surface(s) of the simulation space. The latter has the advantage of having reduced the dimension of the problem by one, and can model propagation through infinite mediums such as that of large outdoor environments (which may be treated as sufficiently “infinite” spaces), due to boundary element methods not requiring a mesh of the whole domain [36]. Wave-based solutions are effective at solving sound propagation for all frequencies, and can model all acoustic effects including diffraction and
scattering [38]. Recall that diffraction is a wave effect most prominent at low frequencies, and will characteristically be most prominent when the wavelength is in the range of several centimeters to a few meters. This also depends on the actual composition of the virtual environment such that the wavelengths are larger than physical obstructions. As a result, low frequency simulation requires a low subdivision resolution and works well for simple scenes. For modelling high frequency effects such as reflection, where the wavelengths are much smaller than the surfaces in the environment, a much higher subdivision resolution is needed. Typically, six to ten elements are required per wavelength [32] [33]. This implies that for an accurate simulation of the sound field with a uniform, non-adaptive resolution, the computational requirements are largely bound to the highest frequency that needs to be accounted for. In addition, although methods used to compute the IR of a virtual environment based on a numerical solution to the wave equation give accurate results, they are further typically limited to static scenes (geometry and/or sound sources), come at a high computational price (including particularly long precomputation times), are inflexible regarding scalability of accuracy and performance requirements, and may require excessively large amounts of memory to store [32] [37] [38] [39]. See the work of Raghuvanshi et al. [33] and Mehra et al. [38] for notable examples of wave-based sound propagation capable of simulation at interactive rates. While their solutions produced relatively accurate results for sound propagation compared to many geometric methods, the runtime performance, storage requirements for precomputed data, and slowdown of iteration times as a result of the precomputation stage would still be less suitable for the needs of most real-time virtual environments and their development.
2.9.5 Geometrical Acoustics-Based Acoustics

Sound field decomposition methods, on the other hand, calculate the IR as a sum of secondary source contributions to the direct path from the environment, and are better suited for interactive applications due to being flexible in accuracy and scalability [32] [39]. Subsequently, most current sound propagation systems for interactive virtual environments are based on geometric acoustics [37] [38] [39]. Geometric acoustic simulations model sound propagation as straight lines that interact with the actual surface geometry and materials of the virtual environment. These generally assume (and are only valid approximations for-) high frequency sound propagation where low frequency wave effects such as diffraction are ignored. Instead, geometric acoustics focuses on recursively computing several reflection paths throughout the scene geometry using ray-based techniques [33]. This is largely similar to geometric optics and ray tracing in the computer graphics domain, which treats light propagation as rays. A prominent advantage of geometric acoustics is that priorities can be assigned to the contributions, allowing for detailed early part and a simplified later part [32]. This can be taken advantage of and used for high quality early reflections with only an approximation of the late reverberant field where the complexity of the simulation would become much more complex. Notable geometric acoustics techniques include the image-source method, ray tracing, and beam tracing [33]. See the work of Taylor et al. [39], Tsingos et al. [40], and Funkhouser et al. [41] for fast, asynchronous implementations of geometric acoustical modelling suitable for real-time virtual environments.

2.9.6 Geometrical Acoustics: The Image-Source Method

The image-source method relies on simulating mirror “images” (virtual projections) of a sound source over all the polygonal surfaces of an environment to model specular reflection. The
reflected trajectories of the room between a listener and real source are represented by the mirror sources. When applied rigorously, the image-source method can calculate all possible paths between a source and listener. While mathematically easy to construct, the image-source method becomes expensive to compute with late-order reflections deep in recursion involving complex geometry that requires additional screening for valid projections (such as visibility tests) [42]. It’s most efficient with rectangular enclosures (which can be used to produce a predictable image expansion “grid” in either 2D or 3D), allowing the image-source method to approach an exact solution to the wave equation when assuming rigid walls [43]. As long as the sound source doesn’t move, the virtual images remain correct and don’t require a recalculation.

2.9.7 Geometrical Acoustics: Ray Tracing

At its core, a ray is a simple structure that can be defined by an origin and direction. Ray tracing in geometrical acoustics begins with the emission of a number of rays from either the sound source or the listener which are then propagated throughout the scene via ray-intersection tests. More rays are generated at surface contacts which represent the path a sound takes as its wave fronts reflect and scatter against the scene geometry [32]. Once a set of rays representing a path reach the target (which will either be a sound source or the listener, depending on where the rays originated from), that path is then considered complete. Unlike the image-source method, ray tracing is not limited to specular reflections, and can easily model diffuse reflections by emitting a “burst” of new rays at appropriate surfaces. Ray tracing also allows for simple accuracy/speed trade-offs through the adjustment of the number of rays emitted [32]. However, some important paths may be missed entirely if the ray count is insufficient/too low, and due to the discrete nature of generating a finite number of rays that may not reach the target, there will
be sampling artifacts [44]. A statistical analysis can be used to find a criteria for the number of rays required [44]. If either the sound source or listener moves, the paths must be recomputed.

2.9.8 Geometrical Acoustics: Beam Tracing

Beam tracing is similar to ray tracing in that a number of rays are emitted that will propagate throughout the scene geometry to find valid reflection paths towards a target [32]. The most notable difference is that sequences of rays are actually pyramidal sweeps, or “beams” that are tested against obstructing surfaces to produce spatially coherent reflections and shadow regions. Each beam-surface intersection represents an infinite number of ray-surface intersections which prevents the sampling artifacts present with ray tracing [41]. Unlike casting rays throughout a scene to find propagation paths or mirroring sound sources using the image-source method, beam tracing requires relatively complex geometric operations for recursive intersection and clipping against 3D models as each beam is reflected and obstructed by several surfaces [41].

2.9.9 Geometrical Acoustics: Diffraction

Geometrical acoustics simulate high-frequency sound propagation. The image-source method, ray tracing, and beam tracing techniques for acoustical modelling do not account for diffraction, which is a low-frequency wave effect. The Uniform Theory of Diffraction (UTD) is a principle that integrates well into a geometrical framework for simulating diffraction to provide satisfying results for most of the audio spectrum [40]. According to the UTD, an incoming ray at an edge will give rise to a cone of diffracted rays (that can also reflect and diffract) originating from a point on the edge that is the cone’s axis [40]. The half angle of this cone is dependent on the incoming ray angle and the edge being used for diffraction. A single ray describes the diffracted field over
an edge for a given source and receiver whose contribution is attenuated by a complex diffraction coefficient [40] [45].

Other approaches involve a perceptual solution to diffraction and occlusion effects, such as those of Cowan and Kapralos, who presented fast, GPU-accelerated methods for approximating sound propagation and occlusion in 2D using acoustical texture maps and pathfinding in real-time [6] [46] [47]. In one of their methods, Cowan and Kapralos rendered the scene as layers of occluding objects using a virtual camera placed at the sound source to generate a listener occlusion map with a [0.0, 1.0] grayscale weighting range (0.0 = black, 1.0 = white). Diffraction effects were approximated by assigning manual occlusion factors to objects in the scene, which determined their ability to block sound via the cumulative darkening of the occlusion map [6] [46]. This method was later extended with additional data encoded into colour channels to store separate acoustical parameters, such as that of occlusion, reflectivity, and precomputed ambient occlusion data (that could be loaded in from a separate graphical rendering software package, such as Maya) [47]. In another method, they used pathfinding on the GPU to determine the distance ratio of direct sound to the path taken around obstructions, which could then be used to drive filters to simulate diffraction [6].

2.10 Summary

This chapter discussed video games and the nature of feedback on the multi-sensory displays that accompany them as a means to present the virtual world to players. I focused on the importance of conveying game state information accurately in regards to the context of the experience that a video game is meant to provide. While a horror game may benefit from
confusing players and making them jump from a well-time sound clip at the thought of a threat that isn’t really there, unintentionally deceptive audio feedback in other genres of video games can lead to frustration and a breaking of immersion when the audio and visual channels have a discrepancy between them and what the player expects. The chapter then concluded with a review of how humans localize sounds perceptually based on directional and spatial cues from the environment, as well as how acoustic spaces are (and have been-) modelled in virtual environments.
Chapter 3: The History of Sound in Video Games

3.1 Overview

This chapter provides an overview of the evolution of game sound over the years. The chapter begins with the era of hardwired circuitry producing simple blips and buzzer sound effects, to fully programmable built-in synthesizers and high-quality sampled audio playback with support for spatialized 3D sound. The chapter ends with a brief comparison between the progress of lighting and sound in video games, and the similarities between their respective feature implementations.

3.2 The Early Years

While it’s true that, today, playing back and manipulating any number of sounds asynchronously in video games is a relatively trivial task on most hardware platforms, this wasn’t always the case. Early video games, such as “Tennis for Two” (1958) and “Spacewar!” (1962) did not have any sound to begin with [48]. Later, when video games entered mass production and evolved to include some sound, both the implementation and results were still intensely primitive by today’s standards. Early arcade video games had differences in their hardware just so that they could have the specific sounds required for that game and its content [48]. As a result, sound circuits were hard-wired and often unique to the game depending on which sounds it featured. Atari’s “Pong” (1972), the first commercially successful video game, is a well-known example of this paradigm [49]. Initially, it was supposed to feature “realistic” sound effects along with a roaring crowd as well as “boo” and “hiss” noises, but the developer who created the game, Allan Alcorn,
did not know how to simulate these on the existing circuit board. Instead, he discovered he could generate different tones, leading to the “beep”, “boop”, and buzzer sound effects that are popularly associated with the game today [49] [50].

3.3 The First Generation: Silence, Bleeps, and Bloops

The first official video game console, the Magnavox Odyssey (1972), brought video games and associated hardware to the home market as viable commercial products, thus marking the start of the first generation of video game consoles [51] [52]. Despite this, it had no sound capability, and subsequent consoles from different companies still only produced a fixed set of sounds through their built-in chips, such as another first generation console, Atari’s “Home Pong” (1975) (see [53], where you can hear the well-known pong sounds near the end) and even the second generation’s first console, the Fairchild Channel F (1976) (see: [54], which consolidates information from the original console manuals, including the one for the Channel F).

3.4 The Second Generation: Programmable Game Sound and Synths

It wouldn’t actually be until the release of second generation consoles such as the Atari 2600 (1977) and Magnavox Odyssey 2 (1978) that we would see improvements to programmable video game sound finally entering the market, allowing games to have their own unique audio for both sound effects and music [55] [56]. Therefore, instead of being limited to the discrete sound wiring configurations fixed to each console, such as a few single-frequency tones or clicks, developers could use the newer programmable sound generator (PSG) chips (integrated with video, or otherwise) on the console to make their own. However, the degree of freedom for sound in games at this time would still have several limiting factors, such as the number of oscillators on
the PSG, waveform types available per-oscillator, as well available space on early video game storage media for playable sound and music data. Another notable landmark of the second generation was the introduction of microprocessor-based technology for all the consoles, allowing several game titles to be released on a “standardized” set of hardware specifications that weren’t just variations of a single game built into the console’s dedicated circuitry [57]. This translated into the use of removable storage media such as ROM cartridges, cassettes, and floppy disks.

3.5 Home Computers and the Fourth Generation: DSP and Stereo for Everyone

Debuting in the late 1970’s [58] and spurred by the increasing popularity of first and second generation video game consoles, home computers became common starting in the early 1980s [59].

Although stereophonic (“stereo” - relating to two output channels) audio had existed in video games for some time, an example being in Midway’s arcade game, Gun Fight (1975) - where the relevant channel would play the sound synced with the visuals for the two players opposite to each other on the left and right (see: [60], which describes the dedicated channel for each player, and [61], where the audio on the video demonstrates stereo separation for players 1 and 2’s sound effects), audio on home consoles from generations one to three was still monophonic (“mono” - relating to a single output channel). It wasn’t until computers like the Amiga in 1985 and the 4th generation of home video game consoles marked by the release of the TurboGrafx-16 system in 1987 that we saw stereo becoming an increasingly common element of game audio [62] [63]. This was also different given that the stereo panning and sounds were programmable,
coming from the games themselves rather than explicit configurations tied to some hardware or arcade cabinet. The 4th generation’s Super Nintendo Entertainment System (SNES) released in 1990 contained an audio processor capable of digital signal processing (DSP) effects applied live to the audio, such as echo, sweep, and envelope [64]. Previously, such as on systems like the NES, effects such as delay, or a “pseudo-reverb” could be achieved by manually duplicating and playing back the audio softer and with a delay on a different channel.

3.6 Enter the CD-ROM: Massive Storage = Quality Sampled Audio

The limitations for sampled digital game audio imposed by game cartridge and floppy disk storage capacity began changing in the early-to-mid 1990’s with the popular adoption of CD-ROMs as a storage medium (often attributed to the game, Myst [65] [66]), and the release of 5th generation consoles which supported them such as the Sega Saturn and Sony PlayStation – both released in 1994. Another console of the 5th generation which was released earlier in 1993, the Atari Jaguar, came out with the “Atari Jaguar CD” add-on in 1995 [67]. The Nintendo 64, released in 1996, was a notable exception to this generation in that it never adopted CD-ROMs as a storage medium. Regardless, some games for the console still had fully recorded digital speech and dialogue as noted in Resident Evil 2 [68], and Spider-Man [69]. Unlike audio generated entirely by sound chips, high quality digitally sampled audio has the advantage that sound and music will sound nearly identical during playback, just as the composer and designers intended.

3.7 First-Person Shooters: Play-by-Sound Mechanics

Games such as Doom popularized the first-person shooter genre to the point that many games released after it were labeled as “Doom clones” (regardless of whether they were actual clones
or not) [70]. Unlike its predecessor, Wolfenstein 3D [71], Doom was not limited to playing a single
digital sample at any time, and sound effects were a key part of the game as a means to alert
players of enemies and their type [72]. Volume attenuation and channel panning (distributing
the sound signal between multiple outputs, such as left and right speakers) were determined by
the location of enemies and their distance, allowing players to localize enemies non-visually.

3.8 Surround Sound and 3D Audio: Maximum Localization

Surround sound and 3D audio arose in video games at about the same time as the new 3D first-
person shooter genre, and evolved throughout the 90s [72]. For gamers, other than in movie
 cinemas, surround sound was likely first experienced in arcades with racing and simulator games
[72], where multiple speaker outputs would literally surround listeners, rather than being only in
front of them as was otherwise common. While traditional stereo panning can be used to roughly
localize a sound in a player’s forward visual field (on a plane), 3D audio and surround sound can
extend that to a full 360-degrees of localization. This is because simple channel panning cannot
appropriately account for sound spatializing on all axis around the player’s view in a game.
Directional ambiguities will exist when the sound source is either in front or behind the player,
and above or below the player.

While the definition of surround sound lends itself to the image of a multi-speaker setup which
encompasses one or more intended listeners, “3D audio” can have various meanings. For clarity,
we’ll define the latter as “positional 3D audio”, which is the intention or ability to give the
perception of sounds coming from space in an arbitrary place around the listener in an arbitrary
environment through either headphones or (external) speakers [24]. This is characterized by
externalization. While a typical stereo pan is simply a change in volume crossing over from the left and right speakers, 3D audio will simulate a sound “outside” of the head and can be set to appear to be sourced somewhere in the XYZ space of a virtual environment [24].

3D audio and surround sound became increasingly prominent in the mid-to-late 90s when several soundcards from various manufacturers were developed with HRTF-based and multi-speaker solutions [24]. These allowed game developers access to more realistic programmable acoustics for their games where properties such as a room’s size and other variables could be set to simulate the virtual sound space using hardware-accelerated algorithms [72].

3.9 DirectSound: Hardware Accelerated Programming for 3D Sound with Abstraction

Microsoft’s DirectX API, which was released with Windows 95, helped to abstract programmers away from tricky hardware-specific code. The sound component of the API, originally known as DirectSound (specifically DirectSound3D for the 3D audio component), was used to create a more realistic 3D sound environment by providing an API for basic 3D aural modelling [73]. This included the ability to render sounds for any point in 3D space and basic effects such as the Doppler effect (by setting a source’s velocity), distanced-based roll-off, a directional “sound cone”, and the simulation of interaural time and level differences for better lateral localization potential [74].

During this time, processing for 3D audio was relatively more expensive, and hardware acceleration helped route some of the additional effects necessary for greater realism. The trade-off for the speed and features that specialized soundcards offered was the flexibility and compatibility of software emulation, thus requiring vendor support and developer opt-in for
implementation. This is less of an issue today with much faster multicore CPUs, but DSP introducing more advanced and realistic audio effects is still too expensive to be fully done purely on the CPU, and AMD and NVIDIA have both brought forward GPU-accelerated solutions via TrueAudio Next and VRWorks respectively.

3.10 Environmental Audio Extensions (EAX): The Environmental Spacialization Standard

A popular example of older hardware-accelerated sound processing is Environmental Audio Extensions (EAX) by Creative Technologies. First released in the late 90’s, EAX was itself an extension to Microsoft’s DirectSound API. While DirectSound did provide some 3D audio features as mentioned earlier, it lacked important environmental effects and filtering for sounds such as reverberation, reflection, occlusion, and obstruction. Although listeners had limited localization cues available directly from instanced sound sources, developers could not define any characteristics of the virtual environment in which the sound was supposed to be in, such as a small padded cell, winding caverns, or a large cathedral. Implemented as DirectSound property sets, EAX was compatible with any manufacturer’s soundcard that cared to support it [73]. Although v1.0 was limited to simulating reverb, later versions included additional effects such as the ones mentioned earlier and support for OpenAL - another DirectSound-like API that supports more platforms [75] [76]. EAX was used widely throughout PC games in the early-to-mid 2000’s (see Figure 11) for an example of a game from this time that can use EAX).
3.11 A3D: Real-time Sound Propagation

EAX’s main competitor at the time was A3D - an API developed by Aureal Semiconductor in collaboration with NASA for use in flight simulators [78] [79], as well as with Matsushita and Disney [80]. A3D was technically more advanced than EAX, being able to simulate thick fog and underwater environments in its first release. A3D v2.0 used the geometric information of an area to render realistic reflections and occlusions, calculating up to 60 first-order reflections interacting in real-time with the environment while grouping later reflections into an overall reverb effect [78]. This version used Aureal’s trademarked “Wavetracing” technology to path trace through the geometry for their advanced real-time effects, while also introducing larger HRTF filters and “A2D” for CPU-based emulation on PCs lacking hardware acceleration [81]. A3D v3.0 added volumetric sound sources to better simulate sounds from large sources such as rivers.
and crowds [82], which require a more complex emission of sound than just a single point. This also helped reduce the number of sound sources required to simulate volumes which would otherwise have to awkwardly use several point sources spread out in sequence over the area of the volume from which they are intended to propagate. Aureal was later acquired by Creative following a legal battle which rendered the former bankrupt [79], after which Creative ceased development on the A3D technology.

3.12 The Sixth Generation: Surround Sound for Consoles

PC gaming had matured a great deal throughout the 1990’s and had the advantage of supporting 3D audio and surround sound years before consoles. While the PlayStation’s “SPU” (sound processing unity) had built-in support for reverb and envelope filters [83], at least one game, Twisted Metal 4 [84] for the original Sony PlayStation, featured relative velocity cues for sound sources using a custom Doppler effect. Beyond that, it was during the sixth generation with consoles such as the Sony PlayStation 2 and Nintendo GameCube, both released in 2000, and Microsoft’s Xbox released in 2001, that consoles games would finally have access to surround sound and 3D audio technology (see Figure 12). While the PlayStation 2 and GameCube both supported surround sound, the original Xbox took console game audio to the next level with additional support for 3D audio via HRTFs built into an enhanced version of DirectSound that was pre-configured for a speaker setup (as opposed to being configured for headphone output) [24] [85]. Unfortunately, because of marketing emphasizing Dolby Digital Surround Sound 5.1 and with consoles being largely home theatre experiences, the HRTF feature didn’t catch on with developers and rarely did games support an option to “use headphones” – which is what HRTFs work best over to deliver directional sound to listeners’ ears [85].
3.13 The Stealth Genre: A Quantum Leap for Play-by-Sound Game Mechanics

Thief: The Dark Project [87] for Windows made sound a strong focus of its game design, going so far as to create an additional “room database” which was used to simulate the realistic propagation of sounds throughout the game levels [88], along with optional support for hardware-accelerated sound effects using EAX v2.0. The AI could react to disturbances created by the player and current sound-based events in the game, such as nearby combat, cries for help from other NPCs, and player-environment interactions, such as movement over noisy surfaces. At the same time, the player could keep track of the game world through sound – such as for out-of-sight patrols around corners – all while being able to read NPC states based on their speech clips. This is one of the first examples of such a strong emphasis on play-by-sound mechanics that was often critical to success and progress throughout a game. Later stealth-genre titles such as
Splinter Cell [89] and the Thief reboot (a 2014 reboot of the “Thief” series by Square Enix) for Windows [90] not only made use of 3D audio, but also had features including the ability to throw objects and distract NPCs with the resulting sound as an additional game mechanic.

3.14 The Death of EAX and DirectSound: The Rise of Cross-Platform Audio Libraries

Just as OpenGL is used for developing portable graphical applications such as games (counter to DirectX’s Direct3D API), the need for a robust cross-platform audio API brought about libraries such as FMOD and OpenAL as popular choices for game sound as alternatives to DirectX’s DirectSound API [91] [92]. Both FMOD and OpenAL support basic DSP effects such as reverb, Doppler Shift, distance attenuation, and more [91] [93]. Although EAX was, for several years, a standard in 3D game audio for the PC, it was later deprecated with the release of Windows Vista in 2006 when Microsoft dropped hardware acceleration support for DirectSound, which is what EAX was built upon [94]. Creative later released the Effects Extension (EFX) suite the same year as a complete, cross-platform replacement for EAX that works with OpenAL to bring all the features of EAX without the need for specialized hardware [93] [95].

Several games throughout this decade made use of both hardware and software for environmental filters and 3D sound, albeit hardware DSP requiring explicit soundcard support quickly faded away following the deprecation of EAX and the increased power of modern CPUs being able to handle common DSP effects. Then, in 2014, Hardware-accelerated DSP for game audio once again made a short resurgence with the release of the Thief series reboot, which was the first game to use AMD’s new “TrueAudio” technology for advanced 3D sound through
programmable DSP cores built directly into the GPU itself [96]. Very few games made use of TrueAudio before it, too, faded away, with very little mention on further developments since.

3.15 VR and the Next Generation of 3D Audio

Whereas modern consoles often emphasize the living room experience with surround sound and motion sensor technology (e.g. Wii, Kinect), VR, mobile, and desktop platforms tend to focus strongly on delivering audio immersion through headphones. The return of VR coincided with the development and successful crowdfunding campaign of the Oculus Rift VR headset in 2012, which was later acquired for further development by Facebook in 2014 [97] [98]. Subsequently, with the increased interest in establishing a strong sense of presence in virtual environments, several APIs, systems, and frameworks emerged post-2014 to allow developers to implement advanced 3D audio into their games and interactive applications using only stereo headphones. Some of the earlier work around this time includes TwoBigEars’ 3Dception audio engine and Impulsonic’s Phonon System. They both featured (and emphasized) support for cross-platform Unity deployment, binaural rendering, environmental occlusion, obstruction, reflection, and more (see [99], [100], and [101] for the 3Dception audio engine demo, Phonon tech preview/walkthrough, and Phonon “SoundFlow” (Propagation) demo respectively). Facebook then acquired TwoBigEars in 2016 to continue working on 3D audio with the Oculus team [102]. In 2016, shortly after the official commercial release of the Oculus Rift, the cross-platform Oculus Spatializer plugin for 3D audio was integrated natively into the Unity engine (v5.4.0b16), giving developers easy and direct access to more advanced audio for commercial use [103] [104] for both VR and non-VR experiences. Valve acquired Impulsonic in 2017 and shortly after released a rework of Impulsonic’s original Phonon System as the Steam Audio SDK [105] [106]. Steam Audio was then
integrated directly with Unreal Engine 4.16 and its native features in 2017 [107], whilst still being actively developed as a plugin/package for Unity. Later in 2017, Google released “Resonance Audio” as a means to introduce multiplatform, scalable spatial audio with SDKs supporting both Unity and Unreal [108].

While the original TrueAudio technology using dedicated/embedded DSP cores has faded away, AMD has since introduced LiquidVR – a GPU-accelerated SDK for their high-end GPUs aimed at increasing comfort and immersion for VR environments, announced in 2015 [109]. The audio component of the SDK is known as “TrueAudio Next”, which, unlike the original TrueAudio, uses existing AMD GPU compute resources (GPGPU) rather than dedicated DSP hardware [110]. In contrast to AMD’s LiquidVR SDK, NVIDIA released VRWorks – a suite of APIs, libraries, and engines to bring physically realistic visuals, sound, touch interactions, and simulated environments to virtual reality in 2017 for Unreal and Unity [111]. The audio component, known simply as VRWorks Audio features ray-traced sound in real-time leveraging NVIDIA’s most recent line of GPUs (the Maxwell and Pascal architectures) [112] [113]. In early 2018, Steam Audio opted to officially support TrueAudio Next [114].

3.16 The Importance of Sound in Video Games

Most games, including first-person shooters, are still playable without any sound. However, for many games, sound can enhance the experience, and even amplify positive outcomes by feeding additional information to the player [115]. In the most likely case, someone playing a first-person shooter with sound will have a clear tactical advantage over someone playing without it, as sound can provide additional cues for common first-person shooter themes such as targets and
objectives (e.g., as done in the original Doom). In these games, it’s often the case that players are made aware of a target or objective before they can confirm it visually. It’s easier to first localize a potential sound source, such as an enemy, and then turn towards it, as opposed to turning around and seeing a visual cue by chance. The first of these is an informed action based on input (hearing a sound somewhere), the second is an explorative action without any initial precursor (you’re looking for a target, but you don’t know if there is one, or where it is specifically). This, of course, mirrors the real-world, where people (not accounting for impairments) will use a combination of audio-visual cues to navigate their environment to perform common everyday tasks such as driving, reading, and communicating with one another. In the real-world, it is also common for hearing to guide the more the more finely tuned visual system.

3.17 Similarities between Light and Sound in Video Games

It’s often suggested that game sound tends to lag behind the implementation and refinement of game visuals and graphical improvements [116] [117] [118]. I propose a closer comparison between lighting in games and game audio. Just as light provides depth and clarity to a virtual environment by illuminating objects and dynamically casting shadows, sound in games has a similar function. Although lighting in games has made vast improvements over the years (fundamentally as a result of more powerful GPUs and newer algorithms) [119], sound has been almost entirely limited to being rendered in “artificial” 3D. Light casts soft shadows that provide depth to objects. It can be occluded, and it diffracts, reflects, and diffusely propagates throughout a game world. Previously, this level of detail in lighting was achieved through baked textures where the calculations were pre-computed and applied over objects (see Figure 13 for a visual comparison of baked indirect lighting).
Figure 13: A simple scene rendered real-time in Unity with and without high-quality baked lighting. The bottom image features only direct illumination. The top image features pre-computed global illumination, where propagated light diffusion is apparent throughout the scene (including ambient lighting provided by a procedural day time skybox and colour bleeding by the emissive blue cube).
More powerful hardware (specifically GPUs) have made it possible to do much of this in real-time [119]. Some examples include pre-computed real-time global illumination through the commercial Enlighten system in Unity on the CPU (precomputed surface bounces on static objects which smoothly react to dynamic lighting over multiple frames) [120] [121], and the fully dynamic voxel-based global illumination of SEGI using the GPU [122]. We can begin to make some connections between methods of lighting in video games and how many games continue to render their soundscapes. The earliest video games offered only crude visual representations of gameplay elements, either through simplistic single-colour vector graphics or low-resolution sprites. This is akin to being able to play the simplest sounds, just as the earliest implementations of game audio did through hardwired circuitry that could only make specific noises through a limited set of channels. Over time, games began supporting more, higher-resolution colour sprites that could be loaded in from external storage media, which is similar to being able to playing back programmable sound sequences using hardware synthesized tones on multiple channels. Eventually, games entered the era of real-time 3D graphics. The earliest of these did not feature any dynamic lighting or shadows, as all the lighting information was baked into the textures prior to being mapped to in-game meshes. Although a later entry when real-time lighting was commonly used and supported in games, Kingdom Hearts [123] for the Sony PlayStation 2 is a prominent example of a game that relied solely on unlit textured geometry as part of its visual aesthetic. The number of real-time lights required would have been too high and impractical for the technology of the time (see Figure 14). Some of the lighting effects were faked using “billboarded” radial texture gradients positioned at the light source (flat quads with a simple radial gradient that always faced the forward camera plane) to simulate volume, and through the
use of smooth texture colour interpolation on nearby dynamic objects (such as the player, crates, and NPCs) which would take on some of the nearest light source colour when entering an invisible volume or distance from the source.

![Figure 14: A scene from the 1st District of "Traverse Town" in Kingdom Hearts, featuring many false light sources (this scene is unlit, and lighting is baked into the textures).](image)

Often, games will have a layer of audio with high detail that provides an acoustic image of the surrounding visuals. An example of this would be a forest setting in a game, where a background “sound texture” will have the sounds of birds chirping, leaves rustling, and a generic wind effect. These static soundscapes can be found in Halo: Combat Evolved, Final Fantasy X [124], Kingdom Hearts, Fable [125], Deus Ex: Human Revolution [126], and many more. The problem is, just like baked lighting, this is not dynamic. Moving the camera around will not pan and simulate the propagation of these effects through the environment because they are “baked” into a single
audio clip or collection of audio clips playing asynchronously. While effective at establishing a
general sense of presence in a virtual environment and saving processing from having to render
too many active sound sources, it can be jarring to notice the lack of dynamism, especially in VR
when using head tracking [24]. With the advent of real-time lighting and shadows, games were
able to convey a very active sense of movement in their games, where moving objects (such as
crates and wall fans) would project shadows onto the occluded surfaces when a strong light was
cast from the other side. Further improved lighting saw the inclusion of real-time global
illumination (bounced indirect lighting) and sub-surface scattering through translucent materials.
This is similar to the model of propagated sound being obstructed and occluded by objects
between the listener and the source, as well as transmission through arbitrary materials and
sound reflected off surfaces (bounced indirect sound). For both lighting and sound, these
respective features aid in presenting a far more cohesive representation of a virtual environment
than just their “dry” counterparts lacking indirect propagation.

3.18 Summary
So far, we’ve looked at how game sound has evolved throughout the history of video games,
leading to the development and availability of several systems and technologies for rendering
more realistic audio. While the very first games either had no sound or built-in configurations for
specific sound effects, later hardware for games, such as dedicated consoles and PCs, featured
increasingly complex sound chips which could render programmable sequences stored on
removable storage media. Eventually, with the popularity of CD-ROMs and the relatively massive
storage they provided, game sound moved towards sampled audio which could faithfully play
back recorded audio. Play-by-sound mechanics began to take hold as audio became another
factor for spatialized 3D game state feedback. Initially just simple panning between the ears (for a headphone or two-speaker setup), early hardware-based technologies such as EAX and A3D provided better spatialized sound through HRTFs and environmental filters without burdening the relatively weak consumer CPUs of their times. This helped provide players a greater sense of immersion in virtual environments in which the directionality of sources and acoustic features of the levels were no longer sonically ambiguous. Later improvements and implementations saw the transition from hardware-specific solutions to more generic software algorithms which could run on the evolving and more powerful CPUs of the following years. With the popular resurgence of VR there has been an increased interest in more immersive audio for games and virtual environments, resulting in several new technologies being made freely available to developers that provide many of the most important features for better spatialized 3D sound. The next chapter will focus on both the new and older technologies by comparing them against each other and highlighting their strengths and weaknesses.
Chapter 4: Comparison of Modern 3D Audio and Spatialization Technology

4.1 Overview

This chapter highlights the most recent technological implementations that brought spatialized 3D sound to games and how they compare against each other in terms of features and application. For API comparisons, some details were omitted to keep the overview brief and focused on the major features.

4.2 DirectSound

DirectSound was capable of handling several of the earliest common requirements for audio in a 3D environment. The most important of these was matching the position of the audio source with the visuals through distance attenuation and adjusting the left/right channel volume (panning) to simulate “3D” audio as the source moved to the left or right of the camera perspective. Developers were also able to specify the directionality of sounds using a simple sound cone model with two cones, where one had a smaller radius than the other and was inside the larger cone. The smaller inner cone was used to create an area of absolute sound intensity where the area between the surface of the smaller cone and the larger cone produced an intensity roll-off (see the [127] for an MSDN example from the XAudio2 API). Using this model, it would be possible to crudely simulate sound diffraction as a directed sound source could be set up near openings, such as a door. DirectSound could also simulate the Doppler effect for moving sounds to cue listeners in on source’s velocity. By rendering the ITD and ILD for sound sources,
DirectSound allowed for effective localization on the lateral plane, although it lacked any cues to help with disambiguating elevation and front-back positional audio. Later revisions included environmental reverb for more realistic simulation of 3D environments, HRTFs, as well some creative effects such as chorus, echo, and more (see [128] and [129] for the effect and HRTF references respectively).

4.3 A3D

4.3.1 A3D v1.0

A3D v1.0 supported HRTFs for better localization potential than DirectSound. With the addition of atmospheric absorption using high frequency attenuation, A3D had better and more realistic depth cues than simply using distance attenuation [74].

4.3.2 A3D v2.0

A3D v2.0, unlike EAX v1.0 and v2.0 reverb presets, actually calculated several first-order reflections (up to 60) and used Aureal’s own trademarked “Wavetracing” technology to propagate through the environment geometry for automated real-time reflection and occlusion effects with support for specifying wall surface materials for more realistic acoustic transmission and dynamic geometry [81] [82] [82]. To do this, developers must define the position, shape, and acoustic material type of the polygons defined for modelling sound propagation, which did not necessarily have to be the same as the visible scene geometry [130]. Later reflections would then be grouped into an overall reverb as a cue to the general sense of space for the environment. Aureal described Wavetracing as “…the sonic equivalent to ray tracing”, where A3D v2.0 was computationally expensive to the point that Aureal developed a dedicated processor to run their
algorithms [78]. A3D also provided a fall-back solution referred to as “A2D” which ran stand-alone on any host CPU to emulate A3D in a software-only environment with a reduced feature set (see Table 1) [81].

<table>
<thead>
<tr>
<th></th>
<th>DirectSound</th>
<th>DirectSound3D</th>
<th>A2D</th>
<th>A3D</th>
<th>A3D 2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default rendering rate</td>
<td>22kHz/8bit</td>
<td>22kHz/8bit</td>
<td>22kHz/16bit</td>
<td>22kHz/16bit</td>
<td>48kHz/16bit</td>
</tr>
<tr>
<td>CPU usage*</td>
<td>16%</td>
<td>68%</td>
<td>10%</td>
<td>&lt;5% (runs h/w accelerated)</td>
<td>&lt;5% (runs h/w accelerated)</td>
</tr>
<tr>
<td>32 Bit Mixing w/ Saturation</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Sample rate conversion/ Doppler</td>
<td>Add/drop sample</td>
<td>Add/drop sample</td>
<td>2nd order interpolation</td>
<td>4th order interpolation</td>
<td>27 point interpolation</td>
</tr>
<tr>
<td>Distance Model</td>
<td>Gain</td>
<td>Gain</td>
<td>Gain</td>
<td>Atmospheric filter, gain</td>
<td>Atmospheric filter, gain</td>
</tr>
<tr>
<td>Positioning: Left/Right</td>
<td>Pan</td>
<td>Pan &amp; delay (ITD)</td>
<td>Pan &amp; delay (ITD)</td>
<td>HRTF</td>
<td>HRTF</td>
</tr>
<tr>
<td>Front/Back</td>
<td>-</td>
<td>Gain</td>
<td>Gain</td>
<td>HRTF</td>
<td>HRTF</td>
</tr>
<tr>
<td>Up/Down</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>HRTF</td>
<td>HRTF</td>
</tr>
<tr>
<td>Occlusions</td>
<td>-</td>
<td>-</td>
<td>Gain**</td>
<td>Gain, material filter**</td>
<td>Gain, material filter</td>
</tr>
<tr>
<td>Reflections</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>HRTF, reverb, material filters</td>
</tr>
</tbody>
</table>

* Intel’s VTune utility measuring Pentium 166 with 8 sounds playing on DSshow software from Microsoft
**only when coupled with A3D 2.0 drivers

Table 1: DirectSound and A3D feature comparison. Source: A3D 2.0 Technology Brief [81].

For compatibility and performance, the A3D Wavetracing engine would apply the available (platform-dependent) and enabled rendering modes [130]. In increasing order of complexity, these were Direct Path (always enabled and available on all platforms), Occlusion (optionally enabled and available on all platforms), and First-Order Reflections (optionally enabled, and not available on all platforms).

**Direct Path** rendering determined the path of sound along a straight line from each source to the listener (see Figure 15). It was modified by the sound source’s cone parameters, orientation, gain, equalization, and relative velocity to the listener (according to the Doppler shift factor). The
sound emitted was attenuated based on the distance between the source and listener, the
distance model scale factor and cut-off distances, as well as the attenuation rate and high-
frequency absorption rate of the environment [130].

Figure 15: Direct path rendering with no geometry (red line). Source: A3D 2.0 Technology Brief [81].

Occlusion functions in much the same way as Direct Path, except that it accounted for intervening
polygons (see Figure 16). When the source and listener were on opposite sides of a wall, sound
would be attenuated based on the low and high frequency occlusion factors of the obstructing
surface materials [130].
Figure 16: Simple occlusion with a single occluding wall between the source and listener (faint red line). Source: A3D 2.0 Technology Brief [81].

*First-Order Reflections* included the same calculations as occlusion, in addition to reflections off intervening polygons and surrounding walls based on the low and high frequency reflection factors defined by the surface materials (see Figure 17, 18, and 19). Reflection was added to the sound from the direct path calculation when both the source and listener were on the same side of a wall (the source and listener may be on the same side of some walls while on the opposite sides of others) [130].
Figure 17: Simple reflection case against three walls (green lines) without any occlusion. Source: A3D 2.0 Technology Brief [81].
Figure 18: Simple occlusion/reflection with two walls reflecting the sound (green lines) and another occluding the direct path (faint red line). Source: A3D 2.0 Technology Brief [81].
Figure 19: More complex reflection/occlusion with six walls reflecting the sound (green lines), where one wall’s reflection is occluded (faint green lines). Source: A3D 2.0 Technology Brief [81].

4.3.3 A3D v3.0

A3D v3.0’s major new additions included volumetric sound sources, geometric reverb, multi-channel surround sound audio (via Dolby Digital / the AC-3 format), and downloadable HRTF support for choosing from a selection of heads for better, more customized 3D localization (as different ear shapes and head sizes produce different HRTFs) [131] [132]. A3D v3.0 was even more computationally expensive than the previous version, requiring an additional digital signal processor to handle the newer commands [78]. Volumetric sound sources allow developers to
define sound sources that are not limited to simple point emitters. In reality, not all sounds emit from a single, infinitesimally small point in space, and it’s unrealistic to have this behaviour in virtual environments. Spherical and conical sound sources with an inner and outer shape are simple forms of volumetric sound. These feature an area in which sound intensity is stable all around the listener (the core, or inner shape), with additional area in which sound can be attenuated over distance (the outer, expanded form of the same shape). This is similar to the directional sound cones in the original DirectSound. This version of A3D introduced the ability to set arbitrary \([x, y, z]\) bounding box dimensions for a sound source along with any orientation [130]. This feature was also accounted for when calculating occlusion effects by measuring the amount of the bounding volume (the box) that was occluded using the size-relative fraction (percentage of the source occluded by a polygon) and the visibility-relative fraction (number of corner points occluded) [130]. Another notable feature of A3D 3.0 was the addition of geometric reverb as part of their Wavetracing technology [132]. With geometric reverb, the process of adding reverb to a scene was automated and A3D would pick the most appropriate reverb based on relevant audio geometry [132]. This allowed developers to avoid having to individually tag and define areas in their game for reverb, although presets were still made available and could be used and/or customized as needed. A3D 3.0 also introduced support for streaming audio, which was beneficial for the playback of larger sound files without having to load them fully into memory, thus saving time and resources [132]. Developers were also able to manually create and place reflections in their virtual environments with controls for gain and delay (as opposed to relying on the automation of the geometry-based Wavetracing engine) to create additional effects, such as echoes [132].
4.4 Environmental Effects Extensions (EAX) and A3D

4.4.1 EAX v1.0

The original v1.0 EAX only had reverb presets for different environments, modelling the reflections and reverb for that specific space. Developers would then choose which preset to apply to their game sound with a limited set of parameters that could be adjusted (gain, decay time, and damping). Several presets were defined in a header file (EAX.H), and developers had the option to create their own (see Figure 20) [133].
<table>
<thead>
<tr>
<th>Preset</th>
<th>Base Environment</th>
<th>Volume</th>
<th>Decay Time</th>
<th>Damping</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX_PRESET_GENERIC</td>
<td>EAX_ENVIRONMENT_GENERIC</td>
<td>0.5F</td>
<td>1.493F</td>
<td>0.5F</td>
</tr>
<tr>
<td>EAX_PRESET_PADDEDCELL</td>
<td>EAX_ENVIRONMENT_PADDEDCELL</td>
<td>0.25F</td>
<td>0.1F</td>
<td>0.0F</td>
</tr>
<tr>
<td>EAX_PRESET_ROOM</td>
<td>EAX_ENVIRONMENT_ROOM</td>
<td>0.417F</td>
<td>0.4F</td>
<td>0.666F</td>
</tr>
<tr>
<td>EAX_PRESET_BATHROOM</td>
<td>EAX_ENVIRONMENT_BATHROOM</td>
<td>0.653F</td>
<td>1.499F</td>
<td>0.166F</td>
</tr>
<tr>
<td>EAX_PRESET_LIVINGROOM</td>
<td>EAX_ENVIRONMENT_LIVINGROOM</td>
<td>0.208F</td>
<td>0.478F</td>
<td>0.0F</td>
</tr>
<tr>
<td>EAX_PRESET_STONEROOM</td>
<td>EAX_ENVIRONMENT_STONEROOM</td>
<td>0.5F</td>
<td>2.309F</td>
<td>0.888F</td>
</tr>
<tr>
<td>EAX_PRESET_AUDITORIUM</td>
<td>EAX_ENVIRONMENT_AUDITORIUM</td>
<td>0.403F</td>
<td>4.279F</td>
<td>0.5F</td>
</tr>
<tr>
<td>EAX_PRESET_CONCERTHALL</td>
<td>EAX_ENVIRONMENT_CONCERTHALL</td>
<td>0.5F</td>
<td>3.961F</td>
<td>0.5F</td>
</tr>
<tr>
<td>EAX_PRESET_CAVE</td>
<td>EAX_ENVIRONMENT_CAVE</td>
<td>0.5F</td>
<td>2.886F</td>
<td>1.304F</td>
</tr>
<tr>
<td>EAX_PRESET_arena</td>
<td>EAX_ENVIRONMENT_arena</td>
<td>0.361F</td>
<td>7.284F</td>
<td>0.332F</td>
</tr>
<tr>
<td>EAX_PRESET_HANGAR</td>
<td>EAX_ENVIRONMENT_HANGAR</td>
<td>0.5F</td>
<td>10.0F</td>
<td>0.3F</td>
</tr>
<tr>
<td>EAX_PRESET_CARPETEDHALLWAY</td>
<td>EAX_ENVIRONMENT_CARPETEDHALLWAY</td>
<td>0.153F</td>
<td>0.259F</td>
<td>2.0F</td>
</tr>
<tr>
<td>EAX_PRESET_HALLWAY</td>
<td>EAX_ENVIRONMENT_HALLWAY</td>
<td>0.361F</td>
<td>1.493F</td>
<td>0.0F</td>
</tr>
<tr>
<td>EAX_PRESET_STONECORRIDOR</td>
<td>EAX_ENVIRONMENT_STONECORRIDOR</td>
<td>0.444F</td>
<td>2.697F</td>
<td>0.638F</td>
</tr>
<tr>
<td>EAX_PRESET_ALLEY</td>
<td>EAX_ENVIRONMENT_ALLEY</td>
<td>0.25F</td>
<td>1.752F</td>
<td>0.776F</td>
</tr>
<tr>
<td>EAX_PRESET_FOREST</td>
<td>EAX_ENVIRONMENT_FOREST</td>
<td>0.111F</td>
<td>3.145F</td>
<td>0.472F</td>
</tr>
<tr>
<td>EAX_PRESET_CITY</td>
<td>EAX_ENVIRONMENT_CITY</td>
<td>0.111F</td>
<td>2.767F</td>
<td>0.224F</td>
</tr>
<tr>
<td>EAX_PRESET_MOUNTAINS</td>
<td>EAX_ENVIRONMENT_MOUNTAINS</td>
<td>0.194F</td>
<td>7.841F</td>
<td>0.472F</td>
</tr>
<tr>
<td>EAX_PRESET_QUARRY</td>
<td>EAX_ENVIRONMENT_QUARRY</td>
<td>1.0F</td>
<td>1.499F</td>
<td>0.5F</td>
</tr>
<tr>
<td>EAX_PRESET_PLAIN</td>
<td>EAX_ENVIRONMENT_PLAIN</td>
<td>0.097F</td>
<td>2.767F</td>
<td>0.224F</td>
</tr>
<tr>
<td>EAX_PRESET_PARKINGLOT</td>
<td>EAX_ENVIRONMENT_PARKINGLOT</td>
<td>0.208F</td>
<td>1.652F</td>
<td>1.5F</td>
</tr>
<tr>
<td>EAX_PRESET_SEWERPIPE</td>
<td>EAX_ENVIRONMENT_SEWERPIPE</td>
<td>0.652F</td>
<td>2.886F</td>
<td>0.25F</td>
</tr>
<tr>
<td>EAX_PRESET_UNDERWATER</td>
<td>EAX_ENVIRONMENT_UNDERWATER</td>
<td>1.0F</td>
<td>1.499F</td>
<td>0.0F</td>
</tr>
<tr>
<td>EAX_PRESET_DRUGGED</td>
<td>EAX_ENVIRONMENT_DRUGGED</td>
<td>0.875F</td>
<td>8.392F</td>
<td>1.388F</td>
</tr>
<tr>
<td>EAX_PRESET_DIZZY</td>
<td>EAX_ENVIRONMENT_DIZZY</td>
<td>0.139F</td>
<td>17.234F</td>
<td>0.666F</td>
</tr>
<tr>
<td>EAX_PRESET_PSYCHOTIC</td>
<td>EAX_ENVIRONMENT_PSYCHOTIC</td>
<td>0.486F</td>
<td>7.563F</td>
<td>0.806F</td>
</tr>
</tbody>
</table>

Figure 20: EAX reverb presets defined in the EAX.H header file. Source: EAX v1.0 User Manual [133].
4.4.2 EAX v2.0

EAX v2.0 introduced occlusion and obstruction, as well as early reflections to the API. Once the environments were configured with their appropriate materials and similar settings, the occlusion and obstruction properties could be manipulated in real-time to simulate relevant scenarios (although the final output was based on the configuration of the environment, detecting changes and obstructions would be up to the developers to implement and then react to using the obstruction and occlusion properties) [134]. When a sound source was fully occluded (with no open-air path to the listener), the sound would be completely muffled and EAX’s occlusion properties would be used to simulate wall materials and thickness. If a sound source was obstructed (partially occluded, with an object between the source and listener), only the direct path sound would be modified, whereas the reverb would be left unaltered [134]. Similar to the first version of A3D, EAX v2.0 added a programmable atmospheric absorption model to simulate sound transmission through various mediums, including foggy air, dry air, and smoke [134]. EAX v2.0 also improved DirectSound’s source directivity. Whereas DirectSound set the directivity across all frequencies equally, EAX v2.0 allowed sound sources to be more directive at higher frequencies than at lower frequencies, which is closer to natural acoustics.

4.4.3 EAX v3.0

EAX v3.0 (known as “EAX Advanced HD 3.0” or simply “EAX Advanced HD” since it was the first) allowed developers to “morph” between multiple environments, such as from a large cave to a small corridor, and provided improved 3D positional audio through the use of HRTFs [135] [78].
4.4.4 EAX v4.0

EAX v4.0 (known as “EAX Advanced HD 4.0”) added support for multiple active environments. While v3.0 supported morphing between multiple environments, there was really only one active environment at any time, and sounds playing in other domains would not carry over the characteristics of their own environment as they passed through to another. This iteration of EAX changed this, allowing scenarios where a sound could carry over reflections from a cave environment into a corridor as it transitioned towards the listener [136]. Previous versions put a strong emphasis on reverb, but with EAX v4.0 developers also had access to several creative DSP filters such as echo, chorus, distortion, and more [137].

4.4.5 EAX v5.0

EAX v5.0 (known as “EAX Advanced HD 5.0”) marked a noticeable increase in the total number of channels available for playback from 64 to 128 (simultaneous voices that could be played). In addition, v5.0 added support for occluding the reverb as well as the direct sound, the latter of which was already possible in previous versions [138]. Another feature of EAX v5.0 was the ability to route the input from a microphone through the EAX DSP chain, allowing external, real-time audio to be simulated in the EAX-defined environment just like any other native sound source [139].

Both EAX and A3D required sound cards that supported their hardware accelerated features for full compatibility with all effects.
4.5 OpenAL, FMOD, and Effects Extensions (EFX) for OpenAL

OpenAL is another audio API similar in functionality to DirectSound, albeit with the advantage of greater cross platform support. With DirectSound having been deprecated following the release of Windows Vista, Creative opted to release their “Effects Extensions” suite for OpenAL known as EFX. EFX ported over features from EAX, this time without being tied to specific hardware. The caveat was that only one EAX-like environment could be simulated on basic hardware and the “Generic Software” device, while the multi-environment model required hardware that explicitly supported it [93]. Another popular cross-platform API worth mentioning is FMOD. FMOD has features similar to OpenAL and is able to automatically select an underlying output system – including DirectSound for Windows XP and below, or Windows Audio Session API (WASAPI) for Windows Vista and above [140]. It’s also kept more up to date than OpenAL and readily supports newer platforms such as the PlayStation 4 and Xbox One consoles.

4.6 Oculus Spatializer

The Oculus spatializer supports HRTFs, volumetric sources, near-field rendering, and environmental modelling [141]. The spatializer explicitly omits features such as occlusion and obstruction, the Doppler effect, creative filters such as chorus and distortion, and directional sound sources. These features are left to higher-level implementation by developers or through the use of middleware [141]. Environmental modelling is implemented through the use of early reflections and late reverberation characteristics simulated with a simple “shoebox” model. This model assumes a room with four walls, a floor, and a ceiling all set at variable distances and with each having its own reflection coefficient. Although this model is basic and not likely to scale well with complex environments, it’s closer to real-life acoustical phenomena than simple artificial
reverb alone or no reverb at all [26]. While the Oculus spatializer doesn’t support consoles, it does support newer versions of Windows, macOS, and Android [142].

### 4.7 Steam Audio

Steam Audio is the updated and rebranded release of Impulsonic’s Phonon system. The following information was aggregated through the Steam Audio documentation and my own experimentation in Unity [143]. Steam Audio has native integration with Unreal, and a separate downloadable package for Unity and a lower level implementation available to use as an API in C/C++. Steam Audio supports newer versions of Windows, macOS, Linux, and Android. It’s extremely customizable, where many of the options can be enabled/disabled and adjusted to strike a balance between quality and performance. Most settings can be configured on a per-source basis. While much of the functionality can be manipulated programmatically, the visual interface provided is enough to easily and deeply configure Steam Audio. Steam audio supports HRTFs (including custom HRTF data loaded through a byte array) and optional bilinear filtering to help smooth out artifacts when the listener’s orientation changes (at the cost of potentially using x2 CPU overhead). In addition, Steam Audio supports indirect sounds via ray traced reflections and convolution reverb, as well as occlusion. HRTF support is available for both direct and indirect sounds. There is also built-in distance attenuation and air absorption. One of the key features that sets apart Steam Audio from previous systems like it such as EAX and A3D is the ability to easily use the actual scene geometry for environmental effects. The geometry does not have to be passed in through an API by programmers, and can instead be marked visually in Unity’s editor. The geometry itself can be either the rendered geometry as will be rendered visually, or something else entirely. Once the geometry to be used by Steam Audio is specified, the scene
must be baked and exported into a format readable by Steam Audio (this involves as single button click if using the Unity integration). Every surface can have its own material that can be customized using either several presets, or custom material properties. Materials specify the low, mid, and high frequency absorption and transmission, as well as a scattering coefficient. The system runs across three different threads: the main thread which is the same as the main game thread in Unity, plus two others for simulation and rendering (the audio) respectively. Steam Audio uses convolution reverb by generating/calculating IRs throughout the level, which is the most expensive part of the system [110], but it can also use listener-centric reverb for reduced CPU usage which takes the listener’s position into account and then ignores source positions. Environmental effects can be baked prior to running the application, and each source can be set to use either real-time or baked indirect sound. Using real-time indirect sound equates to more CPU usage, but less memory consumption, with the opposite being true for using baked simulations. This is highly configurable and has the advantage that static sources to be simulated with extremely high quality, while still allowing the listener to move around the level dynamically. This “semi-dynamic” feature for baked calculations is achieved through the use of probes which will store the pre-calculated data sampled throughout the level for interpolation. Probe density and alignment can be configured. Using Steam Audio’s simulation of indirect sound, it’s possible to determine if you’re near a wall by hearing the reflection get louder on the ear closest to the reflecting surface. The maximum number of sound sources using Steam Audio can be specified to prevent burdening the CPU when the number of audio sources may be dynamic for the duration of the application runtime. The duration/length of calculated IRs can be set globally. A longer calculated IR will generally mean a higher quality simulation, but can be extremely
computationally expensive. For environmental effects, Steam Audio allows users to set the initial ray count, secondary ray count (for diffuse reflections), and a maximum number of bounces separately for both real-time and baked environmental effects. Initial rays are traced from the listener’s position into the environment, while subsequent rays are traced from the surfaces they reflect off. The number of bounces determines how many times rays will reflect off solid objects. The minimum number of bounces is one, since the first bounce is the first-order reflection, without which there is no reflection at all. The real-time reflection component of Steam Audio is similar to A3D’s “Wavetracing” system. Support for AMD’s TrueAudio Next was recently added, which enabled GPU-accelerated convolution operations where a select number of compute units can be reserved for use by Steam Audio (the TrueAudio Next library provides algorithms that can run on both CPUs and GPUs). Since convolution is the most resource intensive part of Steam Audio, using GPU-accelerated convolution can be the difference between 11 ms of CPU time used vs. 2 ms (for 32 sound sources) [110]. TrueAudio Next requires a supported AMD GPU and only works on 64-bit versions of Windows 7 and higher. Steam Audio will seamlessly fallback to using the CPU if either the user’s GPU or configured settings are not supported.

Steam Audio supports direct sound occlusion using a variety of transmission models. If occlusion calculation for a sound source is enabled (it can be blocked by geometry), it can either have no transmission, frequency independent transmission, or frequency dependent transmission. If transmission is disabled, the occluded sound is inaudible. If transmission is frequency independent, the sound is attenuated as it passes through the geometry based on the material properties on the occluding object. The low, mid, and high frequency coefficient are averaged so that the result can be used as a frequency-independent transmission coefficient. If transmission
is frequency dependent, then the result is largely the same as frequency independent transmission, except that the frequency of the transmitted sound is attenuated based on the independent coefficients for the low, mid, and high ranges.

Occlusion calculations can be either based on a binary point source model using a single raycast to determine the on/off state, or a multi-raycast source radius model for partial occlusion (likely achieved using an array of evenly distributed raycasts at the edge of a circle aligned to the plane facing the listener). For the latter, the proportion of rays occluded determine how much of the direct sound is occluded. If transmission is enabled, it is applied only to the occluded portion of the direct sound. Steam Audio has no directional sound, and assumes all sound sources are point emitters when calculating environmental effects. Since occlusion is based on the baked geometry of a scene, features such as non-static doors and windows for occlusion are not natively supported.

4.8 Resonance Audio

Resonance Audio is similar to Steam Audio, with both advantages and disadvantages over the latter. The following information was aggregated through the Resonance Audio documentation and my own experimentation in Unity [144]. Unlike Steam Audio, Resonance also supports iOS and Web with native support for more developer environments such as Android Studio. Resonance also provides more native APIs beyond just C/C++ (Steam Audio), including Java, Objective-C, and the web. With Resonance, it’s possible to simulate directivity for both sound sources and the listener, with all sources being simulated with HRTFs. Resonance uses a simple room/shoebox model, where each of the six surfaces can have their own material presets to
define reflectivity and absorption. Reflectivity, reverb gain, reverb brightness, and reverb time can be controlled through scalar values on the room component. It’s possible to use geometric reverb similar to Steam Audio as opposed to the shoebox model by baking the scene geometry in advance. In this case, it’s also possible to use materials assigned to arbitrary surfaces. The difference is that Resonance Audio will then require manual probe placement where the reverb is likely to fluctuate (similar to placing Light Probes in Unity for static, baked lighting in areas of high contrast), whereas Steam Audio can calculate the reverb in real-time and is able to distribute probes automatically using built-in tools. Occlusion modelling is relatively naïve, using only a simple raycast model that amplifies/aggregates the occlusion effect based on the number of obstructions. However, unlike Steam Audio, this is done in real-time and is not baked with the geometry so dynamic obstructions are supported. In addition to basic real-time occlusion, Resonance Audio features source directivity with adjustable width and control over patterns (cardioid, circular, and figure-eight) that can be interpolated smoothly between one another [145].

4.9 VRWorks Audio

VRWorks Audio is a GPU accelerated path-traced audio solution that creates a complete acoustic image of the environment in real-time without requiring any precomputed filters [113]. The following information is from the VRWorks Audio SDK v1.0.0 documentation and official write-up [146] [28]. VRWorks Audio requires a high-performance computer with a Core i7 CPU or equivalent processor, and Nvidia Maxwell GPU or later. Similar to AMD’s TrueAudio Next, VRWorks Audio is only supported on 64-bit Windows 7 and later. VRWorks Audio uses the Nvidia Acoustic Raytracer (NAVR) API, which is a GPU-accelerated acoustic path-tracing solution that
makes use of the ray tracing hardware capabilities of Nvidia GPUs. The VR Works audio path tracer itself leverages Nvidia’s OptiX, a general purpose, programmable ray tracing engine that can be used to hardware accelerate ray tracing algorithms using the GPU in both the graphics and non-graphics domain [147] [148]. Scene geometry is passed through the API, which is then used to simulate 3D sound source propagation between the source and listener by following the direct path, as well as many indirect (reflected/bounced) paths which contain additional information about the environment and surfaces encountered before being resolved. Although it’s recommended to pass the geometry before tracing is started for efficiency, it’s possible to add meshes during runtime in the game loop. This gives VRWorks Audio the exceptional advantage of being fully dynamic, in contrast to opposing libraries such as Steam Audio and Resonance Audio. The acoustic model and subsequent audio effect filters are all generated in real-time. Acoustic materials containing information for reflectivity and transmission can be applied to meshes and adjusted during the game loop. NVAR traces are scheduled asynchronously from the calling thread and will not stall other operations on that thread, such as graphics draw calls, physics updates, or other work. The trace launch function accepts a Windows event handle that will be signaled once the trace completes, which can then be used to process the results (a set of filters for each output channel). Executing traces faster than they complete will create a backlog, and for this reason it’s recommended to query the event handle for completion before launching another trace. For each pair of source and listener, VRWorks Audio builds a pair of long (up to two seconds), high-resolution (up to 48,000 Hz) convolution filters to model how a sound from a particular source will be heard by the listener’s left and right ears at their position and orientation. The filters include directionality, reverb, occlusion, attenuation,
diffraction, transmission, and more. The trace results can then be applied to a “dry” (unfiltered) audio signal where the input is assumed to be a single, monaural stream. It’s also possible to get the filter data directly for use with custom convolution implementations. Audio trace time scales with the number of sources, which is why it’s recommended to combine certain signals (such as those that may be spatially close to one another and belong to the same entity) into a submix which is then used as the input signal for applying filters to. While the hardware requirements for VRWorks Audio are particularly high, and the platforms supported are limited (currently just 64-bit versions of Windows), it’s also the most advanced system freely available to developers looking for the most realistic 3D audio.

4.10 Summary

Throughout this chapter we’ve reviewed the technology that has made spatialized 3D sound in games possible. From the most “simple” hardware-abstracted APIs such as DirectSound and EAX’s DirectSound property sets, to the full-featured spatialized audio implementations of Steam Audio and VRWorks Audio. We’ve seen advanced game sound progress from being calculated on discrete hardware, then to software, and then partially to hardware again with the rise of GPGPU computing for accelerated DSP. Although the most recent systems have largely managed to implement advanced sound propagation for use in games at highly interactive rates, we’ve seen that there is still an inconsistency between providing sound propagation models that are realistic, dynamic, cross-platform, and scalable. The next chapter will discuss the implementation of a new system that provides an approximation of dynamic sound propagation that can “fill in the gaps” of existing methods.
Chapter 5: The Development of a New System

5.1 Overview

In this chapter, I discuss the development of a new system using the Unity game engine that aims to “fill in the gaps” of previous and current 3D sound systems while still maintaining a high level of scalability between simple mobile platforms and high-end PC gaming hardware. This involves overcoming limitations of platform restriction, performance, and the dynamism of occlusion and diffraction effects with moving listeners, sound sources, and static geometry. The details highlight the use of powerful modern GPUs for hardware-accelerated data pre-processing with a CPU fallback that has no platform or hardware limitations.

5.2 Game Engines

Game engines help to accelerate video game development by providing the underlying features common to many video game development environments. These include graphics rendering, multi-platform deployment, audio playback, networking, 2D/3D physics, input, logic control etc. Developers can then provide a greater focus on the design and gameplay without having to worry about the complex low-level functions that are often specific to certain hardware, although several game engines make it possible to access and customize low-level functions, too. With respect to software, the use of a game engine can exponentially accelerate development for any scale of project, and several fully-featured game engines exist. The most popular of these are Unity, Unreal, and CryEngine. All three are free for both commercial and non-commercial use (although scaling revenue models differ between them), and include a suite of features and
starter assets that make them suited for different focuses. Unity is the most popular game engine at the time of this writing and has already been used for the development of serious games and simulations across multiple platforms [149]. Perhaps the most important feature that makes Unity (v5.4.0b16 and onwards) an ideal choice for virtual environments is the built-in 3D audio spatialization integration that includes HRTF transforms and rapidly advancing support for virtual reality applications [150]. Conversely, CryEngine includes native support for simple approximation of sound obstruction and occlusion using a many raycast model [151] which is not present by default in Unity but is present in the Unreal engine as an on/off state for occlusions [152]. With multiple raycasts, it’s possible to determine an occlusion factor (i.e. obstruction) since some raycasts may be visible, while others may be obstructed. With only a single raycast, this is not possible as it will either be obstructed or it won’t be. Regardless, the techniques that will be discussed are engine-agnostic and can be tailored to work with either one.

5.3 Approximation of Acoustics

Most 3D video games that are played from the perspective of the player-character (first-person view), include sound that simulates left/right panning and distance attenuation. As computational power increased, it became possible to simulate additional effects such as reverb and frequency filters (low-pass, high-pass, band-pass, etc.) with live DSP effects applied to the audio. In its current state, important features that could be used for auditory environment modelling are still omitted and there is no standard for how they are approximated. Although there is plenty of research available (see [46], [47], and [153]), actual game-ready/tested implementations that support scalable multi-platform deployment are sparse with the best ones currently available through audio middleware known as Steam Audio (formerly, the Phonon
system), and Google Resonance, both of which have some limitations. Both systems allow for modelling real-time acoustics using various features, including HRTFs, occlusion, and sound propagation. However, Steam Audio does not account for moving objects and runtime dynamic geometry of any kind (confirmed through my own tests and developer acknowledgement on the official “Steam Community” forum [154]). Google Resonance employs simple additive occlusion values based on the number of colliders between the source and listener (confirmed by my own testing and from reading the component source code, ResonanceAudio.cs -> ComputeOcclusion method). As a result, neither smoothly interpolate sound obstruction when moving throughout the game level (in the case of Steam Audio, this limitation is only for dynamic geometry).

Using the Unity game engine, my current research has resulted in the development of component-based approximations that “fill in the gaps” using distinct methods that are each suited for scaling on their target platforms and the nature of the sound source environment (dynamic/static). This allows for scalable deployment on ranges from low-end mobile devices, to modern high-performance gaming PCs. The first of these is a simple on/off state raycast system that checks the line of sight between the sound source and the listener. If the line of sight is obstructed, the sound signal is occluded either completely or partially using a low-pass filter. This is one of the most commonly used techniques and has been used in games for many years with only minor variation (CryEngine, Unreal, Google Resonance).

The second method makes use of precomputed lighting information positioned at the sound source in-engine that, instead of being applied over the surface geometry in the game as a lightmap, is sampled directly as a texture to retrieve information about the sound throughout
the scene (see Figure 21 for a demonstration of how this 2.5D occlusion is visualized in 3D when all lights are baked into a single lightmap, and Code 1 for the implementation).

Figure 21: A "complex" 3D level rendered real-time in Unity where each sound source (represented as a loudspeaker/megaphone icon) is also an offline point light rendered to a texture.

```csharp
float ColourToLuminance(Color colour)
{
    return (colour.r * 0.3f + colour.g * 0.59f + colour.b * 0.11f) * colour.a;
}

protected override void UpdateOcclusion(bool pathToListenerBlocked)
{
    base.UpdateOcclusion(pathToListenerBlocked);

    Ray ray = new Ray(audioListener.transform.position, Vector3.down);
    Debug.DrawLine(ray.origin, ray.direction, Color.white);

    RaycastHit hitInfo;

    if (Physics.Raycast(ray, out hitInfo))
```
Light and sound sometimes behave in a similar manner when encountering physical surfaces. With this in mind, the high-quality information within the lightmap can be used as a “soundmap” to model acoustic propagation (see Figure 22 for the lightmap texture used in Figure 21). However, this has the limitation that it only works in 2.5D space (flat plane + height) and the sound source must be in a fixed location, as the texture is projected to the surface (it is not volumetric), and cannot be calculated in real-time without a significant performance impact. Depending on the size and configuration of compression for each texture, it may also introduce long load times and significantly increased storage requirements. Pre-computed lighting may require significant time to calculate and render into a texture. Although the texture in Figure 22 has multiple light/sound sources for demonstration, a unique lightmap with only the lighting for a single sound source should be generated so there’s no ambiguity when retrieving the data. This also opens up the possibility for creative use, since different light types can be used for a sound
source (point lights, spotlights, area lights, etc.), and multiple lights representing the propagation of that sound source can be baked into a soundmap.

Figure 22: A high-quality baked lighting texture, known as a "lightmap". It's possible to sample the grayscale values to use as an attenuation factor for sound occlusion if each light source is positioned with a fixed sound source.

The third method utilizes 2.5D pathfinding and has the added benefit that some of the geometry in the scene can be dynamic. The pathfinding data is used to calculate the shortest distance that the sound would have to travel around the level geometry to reach the listener. This is then compared to the direct point-to-point distance and the difference is used to filter the frequency and intensity of the source. Since calculating the path using a navigation mesh in 2.5D is much quicker, this is all processed using the CPU. Using this information, simple sound propagation can be modelled by taking the last visible corner of the path from the listener to the source and projecting a virtual sound source some distance equal to the remaining distance of the path from
this corner to the source in the direction of the corner from the listener (Figure 23). Pathfinding-based sound propagation and/or occlusion has been consistently used in AAA video game titles such as Tom Clancy’s Rainbow Six Siege [155] and Overwatch [156] for immersive audio and “play by sound” mechanics. Some of the limitations of the pathfinding-based propagation systems in these games involve limited spatial dimensions [157] and limited or simulated environment dynamics [158] [159] [160]. My implementation of pathfinding-based sound propagation and occlusion is also scalable to mobile devices and can handle fully destructible environments.
Figure 23: An orthographic top-down view from an area of my game in Unity’s editor. Using pathfinding, the sound will appear to come through the doorway, rather than from behind the player where it actually is – a rough approximation of diffraction (red line) and reflected (blue line) sound.

The final method—and the innovation described here—builds upon the pathfinding-based solution for approximating audio propagation while circumventing the 2.5D limitation entirely by sampling the current scene as a voxel graph and making use of the highly parallel nature of modern GPUs to process this data. Combined with additional optimizations on the CPU, the extreme throughput of the GPU here allows for multiple (potentially several thousands of-) real-time pathfinding calculations to be completed within a single frame to use as raw data for DSP
once retrieved back to the CPU. As a result, the entire implementation down to the end of the DSP chain, even with multiple active, movable sound sources and dynamic level geometry can be run in a complex-geometry, game environment at over 60 frames per second. To my knowledge, this technique is unique and serves as both a proof of concept and an actual implementation of this kind of GPGPU computing for approximating the acoustics of sound.

5.4 Pathfinding

Pathfinding in games can be treated as a graph problem where the virtual environment is represented as a set of traversable and non-traversable vertices grouped together with the edges that connect them. The vertices are points on the map that are either open or closed depending on the existence of obstacles, and the edges are weights based on the cost of the traversal to any given vertex. A common task required in video games is finding the shortest (lowest-cost) path between two points on a map (the virtual environment) in real-time, given only an abstract representation of its features and details. Several algorithms exist for this purpose, along with numerous variations suited for special cases and tasks. Some of the most common pathfinding algorithms are breadth-first search (BFS), Dijkstra’s algorithm, and its heuristic-guided variant, A-Star (A*) [161] [162]. A BFS expands one step towards connected vertices every iteration from a given source vertex and assumes a virtually non-existent cost of traversal each time. It can be used to find the least number of steps required to get between two points, which may also be the shortest path if costs are irrelevant to the results of the search. Dijkstra’s algorithm is similar to BFS, except that it allows for weighted costs for traversal between vertices, allowing for a better definition of the shortest or fastest path. This is often used in video games that include difficult terrain so that an agent passing over it may be taking a more direct path to the
destination, albeit a slower one regardless. A* is a variant of Djikstra’s algorithm that applies a heuristic to guide the search so that instead of “blindly” expanding the frontier until a target is reached, the search tends to gravitate towards the most likely shortest path. As a result, it is more performant than Djikstra’s algorithm when requiring a single path to a target for any given source. For games, this heuristic is frequently a simple distance function of some kind, such as the Euler, or discrete Manhattan distance to the target vertex (depending on use case and requirements).

The use of pathfinding to determine the path followed by sound in video games provides an opportunity for optimization by way of reducing the requirement to a single source, shortest path problem. All sound sources in the virtual environment are perceived by, at most, a single listener. For this reason, A* is unlikely to be useful when there are more than a few sound sources in the scene as each source requires a unique, or partially unique traversal for the heuristics-guided algorithm. By using an expanding BFS-type search, such as a pure BFS, or Djikstra’s algorithm, the shortest path to every vertex in the graph can be resolved from a single source (listener). With respect to performance, the addition of sound sources is then insignificant since a path to the vertex containing the sound source already exists. However, real-time pathfinding on a large enough dataset can still be a complex and intensive task that can severely impact the performance of a video game. For this reason, the operations involved are often offset to worker threads or solved in sequential steps over several frames with a limit to the maximum number of vertices or paths processed in a single update cycle to prevent impacting the game’s performance too severely (Unity does this automatically as part of their built-in pathfinding solution, see [163] and [164]). Several other techniques are also often integrated into pathfinding used in video
games, such as a simplified set of waypoints or steering behaviours to help alleviate the load when a level may be too complex or dynamic. Although this may be suitable and entirely unnoticeable for pathfinding game entities, such as non-player character agents – for which one might expect a realistic reactionary delay – it can still hinder the usefulness of play-by-sound mechanics by not providing immediate useful feedback on the game state. Additionally, 3D pathfinding as required for simulating acoustics propagation with sound in VR environments poses an additional challenge, as the size of the dataset representing the world becomes significantly larger. The parallelization of the pathfinding algorithms that work with this data makes real-time execution possible.

Research regarding the parallel nature of GPUs for solving graph and pathfinding problems has been ongoing, and the results regarding increases in performance over using traditional, optimized CPU implementations (sequential, or otherwise), have been optimistic [165] [166] [167] [168]. Various libraries for GPU-graph processing do exist, including NVIDIA’s own nvGraph [165], as well as Gunrock [169] – both of which are built upon NVIDIA’s parallel computing platform for their line of GPUs, known as CUDA [170]. Despite the availability of these libraries, I have not found any complete implementations of a fully 3D GPU-accelerated pathfinding algorithm for real-time use in video games for sound and existing research that has been done in this area has yet to be tested on modern, consumer-level GPUs like those that would be likely found in today’s generation of gaming PCs (see [166], [167], [168], and [171] for examples of prior research into GPU pathfinding, and [6] for an example of GPU pathfinding for spatial sound). There are also specific optimizations that can be taken advantage of throughout the implementation pipeline when considering 3D GPU pathfinding for sound in video games which
will be discussed in more detail later. Furthermore, prior work regarding parallel processing for pathfinding algorithms has employed parameters that would not traditionally apply to most game-like scenarios. For example, Bleiweiss uses the A* algorithm on the GPU with a maximum testing block of $20 \times 20$ nodes and over 100,000 agents [167]. A $20 \times 20$ grid is unlikely to provide a sufficient resolution for realistic pathfinding applications in most game levels, as the detail of the level geometry and potential obstacles would have to be overly simplified.

5.5 Implementation and System Details

For my implementation of 3D GPU pathfinding I created my own complete framework from the CPU to the GPU in Unity using C# and Unity’s device-independent compute shader language respectively. This provided us with maximum control over the pipeline so that I could test multiple algorithms and still retain the potential for multi-platform deployment later on. Unity compute shaders are written in DirectX 11 HLSL (high level shading language) style and are almost identical to Microsoft’s DirectCompute 5 technology in functionality [172] [173].

I will briefly overview some basic terms specific to Unity that are critical to fully understanding how my implementation works in a game-like environment. Unity has the concept of a “scene”, which can be thought of as a blank canvas for anything that will be part of the final build from the engine. It can be treated as a single level, a menu, multiple levels, multiple menus, an empty “hidden” level used to preload parts of the game, etc. How it is used is determined by the developers. Scenes contain “GameObjects”, which are the base entities for anything that will be in a scene, including props, scenery, and characters, etc. It’s through the addition of a “component” or several components that a GameObject is further defined into a more complex
and purposeful entity in the scene. By default, however, all GameObjects will always have a single “Transform” component which gives it a position, rotation, and scale. There are several built-in components in Unity, but additional behaviours can be created as components using C#. Finally, Unity has a built-in profiler that can be used to monitor execution times (including deeply nested methods), memory usage, garbage generation, and more. I optimized my implementation by heavily using this feature for feedback regarding how changes in my pipeline affected performance.

My pathfinding primarily functions through three custom components and a single compute shader (see Figure 24 for a high-level overview of the system). I have two component scripts that handle all the pathfinding-related operations on either only the CPU using Unity’s NavMesh system, or both the CPU and GPU using my own pathfinding implementation (including the data as it’s transferred to and from the compute shader). Both return similar results and have almost identical function names — the difference is in how they retrieve those results for their roles as the “pathfinder”. I’ll be focusing on the latter which involves the GPU. The third component selects which pathfinder to use and processes the results to control low-pass, reverb, and intensity DSP filters applied to a specific audio stream. I’ll refer to this last component as the “DSP processor”. There is only a single active pathfinder required for each type in the scene (so at most there will be the CPU pathfinder and GPU pathfinder), but there may be multiple DSP processors attached to GameObjects which have Unity’s AudioSource component for manipulating the audio through DSP filters.
Figure 24: A high-level flow diagram showing the overall structure of my spatial audio system. NavMesh obstacles are handled directly by Unity’s system, whereas GPU pathfinder obstacles update node states on the CPU which are then re-sent to the GPU.

I begin on the CPU through a pre-processing stage which first takes into account all the static geometry in the scene once for the duration of the session when the pathfinder component first
“awakes” immediately after the scene loads in. This geometry must be specifically designated in Unity and assigned to a layer which is configured on the pathfinder component to use as a search mask. The pathfinder is given a three-dimensional vector representing the discrete \([x, y, z]\) scale of the area in which the algorithm will operate as an axis-aligned “room” or grid. The origin is in the center and offset by the position of the GameObject through the Transform component. The resolution of this grid is then defined by another value which represents the uniform cell size. I treated 1.0 units in Unity as equivalent to 1 meter so that I could setup realistic game-like scales later. My grid for benchmarking was \([100, 100, 100]\) units, with a cell size of 1.0, resulting in a resolution of 1 cell per meter and a total graph length of 1,000,000 cells (see Figure 25).
Figure 25: The GPU pathfinder component as seen configured in Unity’s editor in the performance benchmarking scene.

The representation of a single cell in the pathfinder is through a custom C# struct called “Node”. A node is defined by an accumulative array index, it’s [x, y, z] index, and world position (see Code 2). It is essentially a vertex in the pathfinding graph.
A global array of nodes containing all the cell data is then created, along with two additional global integer and boolean arrays that will record the traversable state of that node and whether the state may have been changed by dynamic geometry (marked “dirty” for re-evaluation) respectively. Both are actually treated and used as Boolean variables, but since the former will be passed to the compute shader, it must be stored as an array of integers due to the minimum stride of compute buffers being four bytes. The pathfinder then scans each cell area and performs an axis-aligned box test to check for colliders, marking the integer value in the traversable array as either 0 for not traversable, and 1 for traversable. See Code 3 for the implementation.
void CreateGrid()
{
    float nodeSizeHalf = nodeSize / 2.0f;

    gridSizeRaw = gridWorldSize / nodeSize;

    gridSizeX = Mathf.FloorToInt(gridSizeRaw.x);
    gridSizeY = Mathf.FloorToInt(gridSizeRaw.y);
    gridSizeZ = Mathf.FloorToInt(gridSizeRaw.z);

    //...

    nodes = new Node[gridSizeX * gridSizeY * gridSizeZ];

    nodeWalkable = new int[nodes.Length];
    nodeWalkableStatic = new int[nodes.Length];

    isNodeDirty = new bool[nodes.Length];

    Vector3 halfRight = Vector3.right * (gridWorldSize.x / 2.0f);
    Vector3 halfTop = Vector3.up * (gridWorldSize.y / 2.0f);
    Vector3 halfForward = Vector3.forward * (gridWorldSize.z / 2.0f);

    Vector3 positionBottomLeftBack = transform.position;

    // Move to left edge.
    positionBottomLeftBack -= halfRight;

    // Move to bottom edge.
    positionBottomLeftBack -= halfTop;

    // Move to back edge.
    positionBottomLeftBack -= halfForward;

    // Cycle through grid.

    Vector3 vec_right = Vector3.right;
    Vector3 vec_up = Vector3.up;
    Vector3 vec_forward = Vector3.forward;

    Vector3 position;
Vector3 pos_right;
Vector3 pos_up;
Vector3 pos_forward;

Vector3 vec_zero = Vector3.zero;
Vector3 vec_nodeSizeHalf = Vector3.one * nodeSizeHalf;

Quaternion quat_identity = Quaternion.identity;

for (int x = 0; x < gridSizeX; x++)
{
    float precalcX = (x * nodeSize) + nodeSizeHalf;

    pos_right.x = vec_right.x * precalcX;
    pos_right.y = vec_right.y * precalcX;
    pos_right.z = vec_right.z * precalcX;

    for (int y = 0; y < gridSizeY; y++)
    {
        float preCalcY = (y * nodeSize) + nodeSizeHalf;

        pos_up.x = vec_up.x * preCalcY;
        pos_up.y = vec_up.y * preCalcY;
        pos_up.z = vec_up.z * preCalcY;

        for (int z = 0; z < gridSizeZ; z++)
        {
            float precalcZ = (z * nodeSize) + nodeSizeHalf;

            pos_forward.x = vec_forward.x * precalcZ;
            pos_forward.y = vec_forward.y * precalcZ;
            pos_forward.z = vec_forward.z * precalcZ;

            position = positionBottomLeftBack;

            position.x += pos_right.x + pos_up.x + pos_forward.x;
            position.y += pos_right.y + pos_up.y + pos_forward.y;
            position.z += pos_right.z + pos_up.z + pos_forward.z;

            bool walkable = !Physics.CheckBox(
                position, vec_nodeSizeHalf,
                quat_identity, staticCollidersLayerMask);

            int index = x + (gridSizeX * (y + gridSizeY * z));
        }
    }
}
nodes[index] = new Node();

nodes[index].index = index;
nodes[index].position = position;

nodes[index].x = x;
nodes[index].y = y;
nodes[index].z = z;

nodeWalkable[index] = walkable ? 1 : 0;
nodeWalkableStatic[index] = nodeWalkable[index];

Code 3: The CreateGrid function called from the GPU pathfinder component on awake. This initializes the voxel-node graph for pathfinding as a bounding volume in the scene. The basis of this function is from Sebastian Lague’s 2D A* pathfinding tutorial [174].

The next stage of the pathfinder takes place in the update loop every frame and contains the code for the compute shader kernel dispatches. Although I tested two promising GPU-parallelized versions of pathfinding algorithms for viability (i.e., collaborative diffusion [168] and Dijkstra’s algorithm [171]), I found the implementation of parallelized Dijkstra to be more performant early on due to its more reserved use of compute shader kernel dispatches. For this reason, although collaborative diffusion lends itself much better to parallelization by the nature of its operations [168], it was dropped from my testing early on as I used Unity’s profiler to progressively compare the two before moving on to further optimizations. Specifically, it was the kernel requiring a full swap of all values on the grid for every iteration of the diffusion that served as a major bottleneck to the performance of the algorithm. The BFS carried out by the GPU
implementation is similar to a single wave propagation which can be used to trace back towards the source.

A few special optimizations are set in place to determine if the pathfinder needs to engage the GPU at all. In the main update loop, dynamic obstacles are handled at a variable frequency that can be set by the user. In my case, I used a default minimum update delay value of 0.125 s. If an obstacle is detected, the axis-aligned bounding box defining its collider is used to mark all nodes within that area as “dirty” and checked again using an isolated box test for traversable state updates. The array containing these states is then sent to the GPU in one call.

It is important to minimize large data transfers between the CPU and GPU as this can create a performance bottleneck. Therefore, the node data is passed in only once at the start of the scene while the traversable state is stored in a separate array consisting only of integers rather than being a part of the full struct (see Code 4 for the start function and shader bindings). This allows dynamic scene geometry to update the traversable state of nodes without having to communicate unchanged data between the host and device. The amount of data transferred depends on the size of the buffer (which, in my case, is the number of nodes) multiplied by the length of the individual elements in bytes.

```csharp
void Start()
{
    csKernel_setNeighbours = computeShader.FindKernel("SetNeighbours");
    csKernel_reset = computeShader.FindKernel("Reset");
    csKernel_resetSingle = computeShader.FindKernel("ResetSingle");
    csKernel_expandFrontier = computeShader.FindKernel("ExpandFrontier");
    csKernel_checkCompletion = computeShader.FindKernel("CheckCompletion");
}
```
csKernel_resetPathCorners = computeShader.FindKernel("ResetPathCorners");
csKernel_calculatePaths = computeShader.FindKernel("CalculatePaths");

//dispatchSizeX = nodes.Length;
dispatchSizeX = nodes.Length / 16;

int nodeStructSize = Marshal.SizeOf(typeof(Node));
int gpuNodeDataStructSize = Marshal.SizeOf(typeof(GPUNodeData));
int nodeNeighbourStructSize = Marshal.SizeOf(typeof(NodeNeighbours));

pathRequestsArray = new Vector3[maxPathRequests];
pathCorners = new int[maxPathRequests * maxPathCorners];
pathLengths = new float[maxPathRequests];
pathStatus = new int[maxPathRequests];

computeShader.SetInt("maxPathCorners", maxPathCorners);
computeShader.SetInt("maxPathSearchIterations", maxPathSearchIterations);

tempData = new TempData[1];

int tempDataStructSize = Marshal.SizeOf(typeof(TempData));
int vectorStructSize = Marshal.SizeOf(typeof(Vector3));

computeShader.SetInts("gridSize", gridSizeX, gridSizeY, gridSizeZ);
computeShader.SetFloats("gridSizeRaw", gridSizeRaw.x, gridSizeRaw.y, gridSizeRaw.z);
computeShader.SetFloats("gridWorldSize", gridWorldSize.x, gridWorldSize.y, gridWorldSize.z);

computeBuffer_nodes = new ComputeBuffer(nodes.Length, nodeStructSize, ComputeBufferType.Default);
computeBuffer_nodeWalkable = new ComputeBuffer(nodeWalkable.Length, sizeof(int), ComputeBufferType.Default);

computeBuffer_gpuNodeData = new ComputeBuffer(nodes.Length, gpuNodeDataStructSize, ComputeBufferType.Default);

computeBuffer_nodeNeighbours = new ComputeBuffer(nodes.Length, nodeNeighbourStructSize, ComputeBufferType.Default);

computeBuffer_pathRequests = new ComputeBuffer(
pathRequestsArray.Length, vectorStructSize, ComputeBufferType.Default);

computeBuffer_pathCorners = new ComputeBuffer(
    pathCorners.Length, sizeof(int), ComputeBufferType.Default);

computeBuffer_pathLengths = new ComputeBuffer(
    pathLengths.Length, sizeof(float), ComputeBufferType.Default);

computeBuffer_pathStatus = new ComputeBuffer(
    pathStatus.Length, sizeof(int), ComputeBufferType.Default);

computeBuffer_tempData = new ComputeBuffer(
    tempData.Length, tempDataStructSize, ComputeBufferType.Default);

computeBuffer_args = new ComputeBuffer(
    3, sizeof(int), ComputeBufferType.IndirectArguments);

computeBuffer_nodes.SetData(nodes);
computeBuffer_nodeWalkable.SetData(nodeWalkable);

computeShader.SetBuffer(
    csKernel_setNeighbours, "nodes", computeBuffer_nodes);
computeShader.SetBuffer(
    csKernel_setNeighbours, "nodeNeighbours", computeBuffer_nodeNeighbours);

computeShader.Dispatch(
    csKernel_setNeighbours, dispatchSizeX, 1, 1);

computeShader.SetBuffer(
    csKernel_reset, "nodes", computeBuffer_nodes);
computeShader.SetBuffer(
    csKernel_reset, "gpuNodeData", computeBuffer_guNodeData);
computeShader.SetBuffer(
    csKernel_reset, "nodeNeighbours", computeBuffer_nodeNeighbours);
computeShader.SetBuffer(
    csKernel_reset, "tempData", computeBuffer_tempData);

computeShader.SetBuffer(
    csKernel_resetSingle, "nodes", computeBuffer_nodes);
computeShader.SetBuffer(
    csKernel_resetSingle, "nodeWalkable", computeBuffer_nodeWalkable);
computeShader.SetBuffer(
    csKernel_resetSingle, "gpuNodeData", computeBuffer_guNodeData);
computeShader.SetBuffer(
    csKernel_resetSingle, "nodeNeighbours", computeBuffer_nodeNeighbours);
computeShader.SetBuffer(
    csKernel_resetSingle, "tempData", computeBuffer_tempData);
<table>
<thead>
<tr>
<th>Call</th>
<th>Arguments</th>
<th>Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>computeShader.SetBuffer</code></td>
<td><code>csKernel_resetSingle</code>, &quot;args&quot;</td>
<td><code>computeBuffer_args</code></td>
</tr>
<tr>
<td></td>
<td><code>csKernel_expandFrontier</code>, &quot;nodes&quot;</td>
<td><code>computeBuffer_nodes</code></td>
</tr>
<tr>
<td></td>
<td><code>csKernel_nodeWalkable</code>, <code>computeBuffer_nodeWalkable</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>csKernel_gpuNodeData</code>, <code>computeBuffer_gpuNodeData</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>csKernel_nodeNeighbours</code>, <code>computeBuffer_nodeNeighbours</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>csKernel_tempData</code>, <code>computeBuffer_tempData</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>csKernel_args</code>, <code>computeBuffer_args</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>csKernel_checkCompletion</code>, &quot;tempData&quot;</td>
<td><code>computeBuffer_tempData</code></td>
</tr>
<tr>
<td></td>
<td><code>csKernel_checkCompletion</code>, &quot;args&quot;</td>
<td><code>computeBuffer_args</code></td>
</tr>
<tr>
<td></td>
<td><code>csKernel_resetPathCorners</code>, &quot;pathCorners&quot;</td>
<td><code>computeBuffer_pathCorners</code></td>
</tr>
<tr>
<td></td>
<td><code>csKernel_resetPathCorners</code>, &quot;pathLengths&quot;</td>
<td><code>computeBuffer_pathLengths</code></td>
</tr>
<tr>
<td></td>
<td><code>csKernel_calculatePaths</code>, &quot;pathStatus&quot;</td>
<td><code>computeBuffer_pathStatus</code></td>
</tr>
<tr>
<td></td>
<td><code>csKernel_calculatePaths</code>, &quot;nodes&quot;</td>
<td><code>computeBuffer_nodes</code></td>
</tr>
<tr>
<td></td>
<td><code>csKernel_calculatePaths</code>, &quot;gpuNodeData&quot;</td>
<td><code>computeBuffer_gpuNodeData</code></td>
</tr>
<tr>
<td></td>
<td><code>csKernel_calculatePaths</code>, &quot;nodeNeighbours&quot;</td>
<td><code>computeBuffer_nodeNeighbours</code></td>
</tr>
<tr>
<td></td>
<td><code>csKernel_calculatePaths</code>, &quot;pathRequests&quot;</td>
<td><code>computeBuffer_pathRequests</code></td>
</tr>
<tr>
<td></td>
<td><code>csKernel_calculatePaths</code>, &quot;pathCorners&quot;</td>
<td><code>computeBuffer_pathCorners</code></td>
</tr>
<tr>
<td></td>
<td><code>csKernel_calculatePaths</code>, &quot;pathLengths&quot;</td>
<td><code>computeBuffer_pathLengths</code></td>
</tr>
<tr>
<td></td>
<td><code>csKernel_calculatePaths</code>, &quot;tempData&quot;</td>
<td><code>computeBuffer_tempData</code></td>
</tr>
</tbody>
</table>
computeShader.SetFloats("gridOffset", transform.position.x, transform.position.y, transform.position.z);
}

Code 4: Start is called after Awake in Unity. The scene graph has already been created at this point and the compute shader is initialized with buffer/kernel bindings. Many of the elements that will be used on the CPU are also initialized here.

Although not previously viable, advances in compute shader and GPU technology have allowed for much faster transfer speeds, making the possibility of large-scale data handovers and solutions between the host and device possible. This also makes modern GPUs critical to the success of my implementation which possess a larger number of cores (to process more data) and may use proprietary technology for high-speed bandwidth [175]. Next, I check whether the listener GameObject has moved to a position that is different from a previous update cycle, otherwise there’s no need to update the internal grid paths at all. If the dynamic obstacle check is executed and obstacles are found, or if the listener GameObject has moved to a new cell, the pathfinder marks the grid for immediate updating and dispatches the required shader kernels for execution. This is an inherently useful advantage of using a discrete grid for pathfinding, as the listener is effectively treated as the player’s ear. If the player is not moving and the scene remains static, then there is no need to recalculate new paths since the results are guaranteed to be the same when using a BFS search, which calculates a path to every vertex in the graph. The faster the player moves and the higher the resolution of the grid, the more likely it is that an update will be required and the pathfinder will engage the GPU. For GPU benchmarking, I moved the player at a constant maximum speed of 10 m/s (this is close to the maximum recorded sprint speed for humans), or 10 cells/s based on my resolution. This results in an actual maximum
potential update frequency of 10/s and conforms to a game-like scenario with a grounded player character in VR. Unlike multi-frame CPU solutions, updated feedback is guaranteed on movement within the same frame. See Code 5 for the entire update function.

```csharp
void Update()
{
    if (holdToSpawn ? Input.GetKey(KeyCode.F) : Input.GetKeyDown(KeyCode.F))
    {
        for (int i = 0; i < pathRequestSpawnsPerFrame; i++)
        {
            Vector3 position = new Vector3(
                Random.Range(-gridWorldSize.x / 2.0f, gridWorldSize.x / 2.0f),
                Random.Range(-gridWorldSize.y / 2.0f, gridWorldSize.y / 2.0f),
                Random.Range(-gridWorldSize.z / 2.0f, gridWorldSize.z / 2.0f));
            position += transform.position;
            Instantiate(pathRequestPrefab, position,
                        Quaternion.identity, transform);
        }
    }
}

dynamicObstacleGraphUpdateTimer += Time.deltaTime;

for (int i = 0; i < seekers.Length; i++)
{
    pathRequests.Add(seekers[i].position);
}

int pathRequestCount = pathRequests.Count;
int previousPathRequestCount = previousPathRequestNodes.Count;

List<Node> pathRequestNodes = new List<Node>();

if (Application.isEditor)
{
    for (int i = 0; i < pathCorners.Length - 1; i++)
```
if (pathRequestCount != 0)
{
    // Clear and reset before next check.
    // This is ONLY so I can draw gizmos for debugging.
    // It can actually be cleared immediately after
    // sending the update to the GPU.

    int changedNodeCount = changedNodes.Count;

    for (int i = 0; i < changedNodeCount; i++)
    {
        Node node = changedNodes[i];

        if (isNodeDirty[node.index])
        {
            nodeWalkable[node.index] = 1;
            isNodeDirty[node.index] = false;

            nodes[node.index] = node;
        }
    }

    changedNodes.Clear();

    // ...

    int obstacleCount = 0;

    if (dynamicObstacleGraphUpdateTimer >
        dynamicObstacleGraphUpdateInterval)
    {
        // NOTE: I NEED to wait until the timer to sync with
        // regular updates for the obstacle checks, else an
        // obstacle may cause a bounds-check twice due to being
        // removed the frame immediately after an obstacle update
        // which will cause issues with not updating the nodes
        // because of isNodeDirty assignments.
// I could probably find a way around it so that updates
// are performed immediately if an obstacle is added/removed,
// but waiting for the timer also makes sense because I
// ensure dynamic obstacle updates are kept
// at the interval rate.

for (int i = 0; i < obstaclesToRemove.Count; i++)
{
    RemoveAndUpdateNodesInObstacleBounds(obstaclesToRemove[i]);
}

// If there are no obstacles after this one is removed,
// the graph won't update until the player moves.

// Use this to force a refresh after an obstacle is removed.
// I could also store the previous and current obstacle
// count and use that (compare if current < previous).

// But this is easier to understand.

obstacleRemoved = obstaclesToRemove.Count != 0;
obstaclesToRemove.Clear();

obstacleCount = obstacles.Count;

for (int i = 0; i < obstacleCount; i++)
{
    GPUPathfinderObstacle obstacle = obstacles[i];
    UpdateNodesInObstacleBounds(obstacle);
}

// ...

for (int i = 0; i < pathRequestCount; i++)
{
    pathRequestsArray[i] = pathRequests[i];
}

Node targetNode =
    NodeFromWorldPosition(target.transform.position);

bool targetNodeChanged =
    targetNode.index != previousTargetNode.index;
previousTargetNode = targetNode;

if (targetNodeChanged || obstacleCount != 0 || obstacleRemoved)
{
    updateGraph = true;
}

obstacleRemoved = false;

// If target node changed, paths have to be recalculated anyway.

bool recalculatePaths = updateGraph;

if (updateGraph)
{
    computeShader.SetFloats("startPosition",
        target.position.x, target.position.y, target.position.z);

    computeShader.Dispatch(csKernel_reset, dispatchSizeX, 1, 1);
    computeShader.Dispatch(csKernel_resetSingle, 1, 1, 1);

    for (int i = 0; i < maxDispatchIterations; i++)
    {
        computeShader.DispatchIndirect(  
            csKernel_expandFrontier, computeBuffer_args);
        computeShader.Dispatch(csKernel_checkCompletion, 1, 1, 1);
    }

    updateGraph = false;
}

// If graph is not going to be updated, then paths may not
// have to be re-calculated if the path requests since last
// check haven’t been changed. So... check for changes!

if (!recalculatePaths)
{
    if (pathRequestCount != previousPathRequestCount)
    {
        recalculatePaths = true;
    }
    else
    {
        recalculatePaths = true;
    }
}
for (int i = 0; i < pathRequestCount; i++)
{
    Node pathRequestNode =
    NodeFromWorldPosition(pathRequests[i]);

    if (pathRequestNode.index !=
        previousPathRequestNodes[i].index)
    {
        recalculatePaths = true;
    }

    pathRequestNodes.Add(pathRequestNode);
}

if (recalculatePaths)
{
    computeBuffer_pathRequests.SetData(pathRequestsArray);

    computeShader.Dispatch(
        csKernel_resetPathCorners, pathCorners.Length, 1, 1);
    computeShader.Dispatch(
        csKernel_calculatePaths, pathRequestCount, 1, 1);

    computeBuffer_pathCorners.GetData(pathCorners);
    computeBuffer_pathLengths.GetData(pathLengths);
    computeBuffer_pathStatus.GetData(pathStatus);
}

previousPathRequestNodes.Clear();
int pathRequestNodeCount = pathRequestNodes.Count;

for (int i = 0; i < pathRequestNodeCount; i++)
{
    previousPathRequestNodes.Add(pathRequestNodes[i]);
}

pathRequestCountGlobal = pathRequestCount;

pathRequests.Clear();

Code 5: The GPU pathfinder update function (called every application frame update).
If an update to the grid is required, the pathfinder sends the latest position of the listener to the compute shader and begins a loop that dispatches an expand frontier kernel (see Code 6) for Dijkstra’s algorithm indirectly, as well as another kernel directly with a single active thread to monitor completion (see Code 7). In DirectCompute, when dispatching a kernel directly, the number of threads to execute on \([x, y, z]\) are pre-determined in the call, whereas an indirect dispatch takes in an argument buffer that can be updated on the GPU-side as a means for dynamic parallelism, where the number of threads launched per dispatch can change through code executed on the GPU. By using this feature, I saved approximately 2 ms of execution time for the pathfinder in the worst-case scenario (per-frame execution).

```cpp
[numthreads(16, 1, 1)]
void ExpandFrontier(
    uint3 id : SV_DispatchThreadID, uint groupIndex : SV_GroupIndex)
{
    int offset = id.x + (gridSize.x * (id.y + (gridSize.y * id.z)));
    if (gpuNodeData[offset].frontier)
    {
        gpuNodeData[offset].frontier = 0;
        int neighbours[26];
        GetNeighbours(nodes[offset], neighbours);
        [unroll]
        for (uint i = 0; i < neighbours.Length; i++)
        {
            int neighbourIndex = neighbours[i];

            if (neighbourIndex == -1 || !nodeWalkable[neighbourIndex])
            {
                continue;
            }

            int newCost = gpuNodeData[offset].cost + 1;
        }  
    }
```
int previousCost = gpuNodeData[neighbourIndex].cost;
InterlockedMin(gpuNodeData[neighbourIndex].cost, newCost);

if (previousCost > newCost)
{
    tempData[0].count++;
    gpuNodeData[neighbourIndex].frontier = 1;
}

}
}

Code 6: The frontier propagation kernel in the compute shader.

Once the second kernel determines that expansion has completed and there are no more nodes
left to explore on the frontier, the first kernel becomes inactive until all dispatch calls from the
pathfinder’s loop have been exhausted.

[numthreads(1, 1, 1)]
void CheckCompletion(uint3 id : SV_DispatchThreadID)
{
    if (tempData[0].count == 0)
    {
        args[0] = 0;
        args[1] = 0;
        args[2] = 0;
    }
    tempData[0].count = 0;
}

Code 7: The kernel that checks to see if frontier propagation has stopped flowing into new cells of the graph.

At this point, the grid is up to date and all nodes stored on the GPU have the correct neighbour
indices that can be traced to the starting vertex node. The pathfinder can then dispatch a final
calculation kernel once that can be used to update and retrieve an array of indices containing all
the path data in sequence (calculated in parallel, per path), and another array containing the
length of all the paths in meters (see Code 8). The length of the path is a sum of the distances between the corners.

```csharp
[numthreads(1, 1, 1)]
void CalculatePaths(uint3 id : SV_DispatchThreadID)
{
    Node targetNode = GetNodeFromWorldPosition(startPosition);
    Node currentNode = GetNodeFromWorldPosition(pathRequests[id.x]);

    float distance = 0.0f;
    int pathIndexOffset = id.x * maxPathCorners;

    uint pathCornerCount = 0;
    int3 previousDirection;

    for (uint i = 0; i < maxPathSearchIterations; i++)
    {
        // Expansion never reached this node from the source.
        if (gpuNodeData[currentNode.index].cost == INT_MAX)
        {
            pathLengths[id.x] = INT_MAX;
            pathCorners[pathIndexOffset] = -1;

            pathStatus[id.x] = PathStatus_Invalid;

            return;
        }

        int neighbours[26];

        GetNeighbours(currentNode, neighbours);

        Node nextNode;
        int minCost = INT_MAX;

        [unroll]
        for (uint n = 0; n < neighbours.Length; n++)
        {
            int neighbourIndex = neighbours[n];

            if (neighbourIndex == -1)
            {
```
continue;

if (gpuNodeData[neighbourIndex].cost < minCost)
{
    nextNode = nodes[neighbourIndex];
    minCost = gpuNodeData[neighbourIndex].cost;
}
}

distance += length(currentNode.position - nextNode.position);

int3 currentDirection = int3(
    currentNode.x - nextNode.x,
    currentNode.y - nextNode.y,
    currentNode.z - nextNode.z);

if (currentDirection.x != previousDirection.x ||
    currentDirection.y != previousDirection.y ||
    currentDirection.z != previousDirection.z)
{
    pathCorners[pathIndexOffset + pathCornerCount] = currentNode.index;
    pathCornerCount++;

    if (pathCornerCount == maxPathCorners)
    {
        pathStatus[id.x] = PathStatus_Partial;
        return;
    }
}

currentNode = nextNode;
previousDirection = currentDirection;

if (currentNode.index == targetNode.index)
{
    distance += length(currentNode.position - nextNode.position);
    pathCorners[pathIndexOffset + pathCornerCount] = currentNode.index;
    pathStatus[id.x] = PathStatus_Complete;
break);

// If path wasn't marked as completed when the loop finished,
// then I never reached the source due to running out of search
// iterations, even though a full path is possible.
if (pathStatus[id.x] != PathStatus_Complete)
{
    pathStatus[id.x] = PathStatus_Partial;
}

pathLengths[id.x] = distance;

Code 8: The final kernel in the GPU pathfinder system that sums up path data and records its state.

The DSP processor component works individually on every AudioSource GameObject in the scene and must be attached as an additional component to each one to have it work in conjunction with the pathfinder (see Figure 27). This part of my implementation replicates existing solutions that use pathfinding as the base for approximating propagation acoustics, except that it can query paths using my custom GPU pathfinder component for results (it can switch between choosing the appropriate pathfinder at runtime).
A raycast from every DSP processor’s position (based on its GameObject Transform component) is executed towards the listener (see Code 9).

```csharp
protected virtual void Update()
{
    PathfinderManager.instance.audioOcclusionPerformanceStopwatch.Start();

    pathToListenerBlocked =
        Physics.Linecast(
            audioSource.transform.position,
            audioListener.transform.position,
            raycastLayerMask, QueryTriggerInteraction.Ignore) ||
        Physics.Linecast(
            audioListener.transform.position,
            audioSource.transform.position,
            raycastLayerMask, QueryTriggerInteraction.Ignore);

    PreUpdateOcclusion(pathToListenerBlocked);
}
```

Code 9: A double linecast (a form of raycasting defined as a line) to check for obstructions between source and listener.

A hit triggers an indication of some obstruction, which in turn creates a request to the target pathfinder for a path to the current DSP processor (see Code 10).

```csharp
protected override void PreUpdateOcclusion(bool pathToListenerBlocked)
{
    if (pathToListenerBlocked)
    {
        // Note: While the GPU pathfinder is intentionally locked
        // to the voxel simulation bounds, the only way requestPathSuccess
```
The length of this path is compared to the direct, straight-line distance between the sound source and the listener, and the difference ratio to the maximum range of the audio source is then used as an input to various functions with exposed properties in Unity’s editor so that sound designers can tweak how the DSP filters should. Additional propagation is modelled in the DSP processor by sampling the last visible corner from the perspective of the listener and using that to instantiate a duplicate sound source positioned the remaining distance away from the angle of
the listener to the corner. This is an approximation of the last reflected sound wave reaching the
listener and also simulates sound that “curves” or diffracts around corners so that its position
isn’t heard as strongly directly from its initial position, but can be used to trace back to the original
source if the player carefully aligns themselves and follows the duplicate sound (see Figure 23).
See Code 11 for the main DSP processor update function.

```csharp
protected override void UpdateOcclusion(bool pathToListenerBlocked)
{
    base.UpdateOcclusion(pathToListenerBlocked);

    float pathLength = 0.0f;
    float directPathLength = 0.0f;

    float pathLengthDifference = 0.0f;

    Vector3 sourcePosition = transform.position;
    Vector3 listenerPosition = audioListener.transform.position;

    if (!pathToListenerBlocked)
    {
        directPathLength = Vector3.Distance(sourcePosition, listenerPosition);

        Debug.DrawLine(sourcePosition, listenerPosition, Color.green);

        float normalizedDistanceToSoundSource = Mathf.Clamp01(directPathLength / audioSource.maxDistance);

        if (volumeFromPathLengthDivergence)
        {
            targetVolume *= DistanceFade_ExponentialLog10(normalizedDistanceToSoundSource);
        }
    }

    virtualAudioSourceVolumeDSP.volume = Mathf.Lerp(virtualAudioSourceVolumeDSP.volume, 0.0f, Time.deltaTime * virtualAudioSourceVolumeDSPlerpSpeed);
```
// When line of sight to target is clear, reflected propagated
// sound source volume should be inversely proportional to
// max distance from the original sound source.

// This smoothly fades the original signal to the
// reflected signal as I back away from the source,
// and allows a coherent change to the volume if an
// obstruction appears.

// When the virtual volume target for the reflected
// propagation was zero, I had the problem that I could
// have complete silence if I was >= original sound source
// max distance away from the source, and then I'd hear
// the reflected sound source once I moved behind a corner
// to enable obstruction... which didn't make sense.

// Now I smoothly go between the two!

float propagatedVolumeTarget = normalizedDistanceToSoundSource;

// Although I use a constant zero for when the path is blocked,
// I lerp towards zero as I get closer to the source, which sounds
// better as it fades out.

virtualAudioSourceVolumeDSP2.volume = Mathf.Lerp(
    virtualAudioSourceVolumeDSP2.volume,
    propagatedVolumeTarget,
    Time.deltaTime * virtualAudioSourceVolumeDSPLerpSpeed);

virtualAudioSource.transform.position = Vector3.Lerp(
    virtualAudioSource.transform.position,
    sourcePosition,
    Time.deltaTime * virtualAudioSourcePositionLerpSpeed);

} else {
    if (requestPathSuccess)
    {
        bool calculatePathSuccess;
switch (pathfinder)
{
    case Pathfinder.NavMeshCPU:
    {
        calculatePathSuccess =
            PathfinderManager.instance.cpuPathfinder.
            IsPathComplete(pathId);
        break;
    }
    case Pathfinder.VoxelGraphGPU:
    {
        calculatePathSuccess =
            PathfinderManager.instance.gpuPathfinder.
            IsPathComplete(pathId);
        break;
    }
    default:
    {
        throw new System.Exception("Unknown enum.");
    }
}
if (calculatePathSuccess)
{
    switch (pathfinder)
    {
        case Pathfinder.NavMeshCPU:
        {
            pathLength = PathfinderManager.instance.
                cpuPathfinder.GetPathLength(pathId);
            // ONLY FOR THE CPU PATHFINDER.
            // From this point on,
            // use the projected positions.
            // Anything above should be using
            // the original values, but now I
            // should be using these so proper
            // aligning is done.
// Useful for reflected virtual source positioning.

sourcePosition = PathfinderManager.instance.cpuPathfinder.GetProcessedPathRequest(pathId).audioSourcePositionOnNavMesh;

listenerPosition = PathfinderManager.instance.cpuPathfinder.GetProcessedPathRequest(pathId).audioListenerPositionOnNavMesh;

break;

} case Pathfinder.VoxelGraphGPU:
{
    pathLength = PathfinderManager.instance.gpuPathfinder.GetPathLength(pathId);

    break;

} default:
{
    throw new System.Exception("Unknown enum.");
}

} directPathLength = Vector3.Distance(sourcePosition, listenerPosition);

pathLengthDifference = pathLength - directPathLength;

pathLengthDifference = Mathf.Max(0.01f, pathLengthDifference + (maxPathLengthDifference * maxPathLengthDifferenceInflation));

float normalizedPathLengthDifference = Mathf.Clamp01(pathLengthDifference / maxPathLengthDifference);
targetVolume *= DistanceFade_ExponentialLog10(
    normalizedPathLengthDifference);

if (volumeFromPathLengthDivergence)
{
    targetVolume *= DistanceFade_ExponentialLog10(
        pathLength / audioSource.maxDistance);
}

targetLowPassFilterCutoffFrequency *=
    1.0f - normalizedPathLengthDifference;

// Find last visible path corner.

List<Vector3> pathCorners = pathfinder ==
    Pathfinder.NavMeshCPU ?
    PathfinderManager.instance.cpuPathfinder.
        GetPathCorners(pathId) :
    PathfinderManager.instance.gpuPathfinder.
        GetPathCorners(pathId);

    int pathCornerCount = pathCorners.Count;

    for (int i = 0; i < pathCornerCount - 1; i++)
    {
        Vector3 position = pathCorners[i];
        Vector3 nextPosition = pathCorners[i + 1];

        Debug.DrawLine(
            position, nextPosition, Color.red);

        Debug.DrawRay(
            position, Vector3.up,
            Color.Lerp(
                Color.yellow, Color.red,
                i / (float)pathCornerCount));
    }
// If using nav mesh, just take the second last corner.

// Doing a cast can produce incorrect results since
// unlike the voxel graph the shape of the mesh doesn't
// necessarily cover the colliders, leading to an
// early hit from the cast against colliders that
// protude through the nav mesh geometry.

// This happened in the case of the stairs, where the
// nav mesh was a smoother slope, but the ray hit the
// edge of the steps before reaching the corner at the
// end of the stairs.

// This doesn't happen with the voxel graph because
// there are several corners calculated above the
// steps themselves.

// EDIT: Only take the second last result for nav mesh
// if the last visible corner result is the next immediate
// corner from the listener (or if I couldn't work out the
// obstruction), since that would produce the incorrect
// result I was talking about.

// Else, just use the actual last visible corner because
// otherwise I'd get the wrong result if I just used the
// second -last corner as-is, due to heightened areas that
// are immediately next to flat area with no obstruction
// (I'm not talking about the stair ways like in
// Spooky Game which are within their own closed halls,
// I'm talking about open ramps/stairs like in my test
// environment).

// This helps somewhat with both scenarios,
// but because a corner may be around the edge of
// a ramp, that can cause an obstruction and cause a
// source to go to that area even if it shouldn't be
// projected in that direction.

// Not much I can do here when limited to a 2.5D plane.

// Obviously, this problem doesn't happen with the
// GPU voxel graph method.

    int lastVisibleCornerIndex = -1;
// This next part assumes the perspective of the listener,
// hence the reversals.

// Reversed so I calculate from listener
// position towards source.

// Trim start by one (-2 vs. -1) because that's (approx.)
// the listener position.

// It would be count - 1 otherwise (zero-indexed array,
// and I'm starting with the total count).

    for (int i = pathCornerCount - 2; i >= 0; i--)
    {
        Vector3 position = pathCorners[i];

        // If this corner is obstructed, take the previous
        // corner (which would have been visible) and use that.

        if (Physics.Linecast(
                listenerPosition, position,
                raycastLayerMask, QueryTriggerInteraction.Ignore))
        {
            lastVisibleCornerIndex = i + 1; break;
        }
    }

    if (pathfinder == Pathfinder.NavMeshCPU)
    {
        if (lastVisibleCornerIndex == pathCornerCount - 1)
        {
            // Same as x = pathCornerCount - 2.

            lastVisibleCornerIndex--;
        }
        else if (lastVisibleCornerIndex == -1)
        {
            lastVisibleCornerIndex = pathCornerCount - 2;
        }
    }

    if (lastVisibleCornerIndex != -1)
    {
        Vector3 lastVisibleCorner =
            pathCorners[lastVisibleCornerIndex];
```csharp
Debug.DrawLine(
    listenerPosition, lastVisibleCorner, Color.white);

float remainingDistanceToSource = 0.0f;

for (int j = lastVisibleCornerIndex; j >= 1; j--)
{
    remainingDistanceToSource +=
        Vector3.Distance(pathCorners[j], pathCorners[j - 1]);
}

Vector3 directionToLastVisibleCorner =
    Vector3.Normalize(
        lastVisibleCorner - listenerPosition);

Vector3 reflectedSoundSourcePosition =
    lastVisibleCorner +
    (directionToLastVisibleCorner *
    remainingDistanceToSource);

reflectedSoundSourcePosition = Vector3.ProjectOnPlane(
    reflectedSoundSourcePosition, Vector3.up);

reflectedSoundSourcePosition.y = lastVisibleCorner.y;

Debug.DrawRay(
    reflectedSoundSourcePosition,
    -directionToLastVisibleCorner * 2.0f, Color.cyan);

float targetVirtualVolume =
    1.0f - Mathf.Clamp01((
        pathLength - directPathLength) /
        maxPathLengthDifference);

virtualAudioSourceVolumeDSP.volume = Mathf.Lerp(
    virtualAudioSourceVolumeDSP.volume, 
    targetVirtualVolume, 0.5f);
```
targetVirtualVolume,

float virtualAudioSourceVolumeDSPLerpSpeed = 0.1f;

virtualAudioSourceVolumeDSP2.volume = 1.0f;

virtualAudioSource.transform.position = Vector3.Lerp(
    virtualAudioSource.transform.position,
    reflectedSoundSourcePosition,
    Time.deltaTime * 12.0f);

else
{
    LerpVirtualSourcesToZero();
}

else
{
    LerpVirtualSourcesToZero();
}

else
{
    LerpVirtualSourcesToZero();
}
platforms that Unity supports. The pathfinder component is not tied to the sound part of the system and can be used for other tasks in a game by simply requesting a path from any other part of the game loop. It is an approximation that can be used as-is, or in conjunction with other systems such as Steam Audio or Google Resonance to fill in the gap for real-time, dynamic occlusion modelling.

5.6 Benchmarking Results

The extended spatial audio system described was run on several devices with varying hardware specifications and operating systems. For optimizations during development, performance was monitored using Unity’s built-in profiler in the editor (see Figure 27).
For standalone runtime performance tests I created a custom profiler integrated into the global pathfinder manager that recorded execution times for the entire, framerate-unlocked application (disabled vertical synchronization, commonly known as “Vsync”), as well as isolated execution times for the spatial audio system over a specified duration (see Code 12).

```csharp
class PathfinderManager : MonoBehaviour
{
    public static PathfinderManager instance;

    public CPUPathfinder cpuPathfinder;
    public GPUPathfinder gpuPathfinder;

    public bool enablePathfinding = false;


    public Stopwatch audioOcclusionPerformanceStopwatch = new Stopwatch();
    public Stopwatch applicationPerformanceStopwatch = new Stopwatch();

    public float logPerformanceDuration = 60.0f;
    float logPerformanceTimer;

    bool logPerformance;

    double logPerformanceApplicationMin;
    double logPerformanceApplicationMax;
    double logPerformanceApplicationAverage;

    double logPerformanceAudioOcclusionMin;
    double logPerformanceAudioOcclusionMax;
    double logPerformanceAudioOcclusionAverage;

    int logPerformanceSampleCount;
}
void Awake()
{
    instance = this;

    // Update to current pathfinder is called first
    // so it can set the pathfinder mode on all AudioOcclusion
    // components in the game.
    // UpdateToCurrentPathfinderEnabledState will potentially
    // just disable both pathfinders after, so it's safe.
    UpdateToCurrentPathfinder();
    UpdateToCurrentPathfinderEnabledState();
}

void Start()
{
    QualitySettings.vSyncCount = 0;
    Application.targetFrameRate = int.MaxValue;
}

void Update()
{
    if (Input.GetKeyDown(KeyCode.N))
    {
        ToggleEnablePathfinding();
    }

    if (enablePathfinding)
    {
        if (Input.GetKeyDown(KeyCode.L))
        {
            SetNextPathfinder();
        }
    }

    if (Input.GetKeyDown(KeyCode.P))
    {
        LogPerformance();
    }
}

void LateUpdate()
{
    float deltaTime = Time.unscaledDeltaTime;
```csharp
double applicationPerformanceElapsedTime = applicationPerformanceStopwatch.Elapsed.TotalMilliseconds;
applicationPerformanceStopwatch.Reset();
applicationPerformanceStopwatch.Start();

if (logPerformance)
{
    logPerformanceTimer += deltaTime;
}

double audioOcclusionPerformanceElapsedTime = audioOcclusionPerformanceStopwatch.Elapsed.TotalMilliseconds;

if (logPerformanceApplicationMin > applicationPerformanceElapsedTime)
{
    logPerformanceApplicationMin = applicationPerformanceElapsedTime;
}
if (logPerformanceApplicationMax < applicationPerformanceElapsedTime)
{
    logPerformanceApplicationMax = applicationPerformanceElapsedTime;
}

if (logPerformanceAudioOcclusionMin > audioOcclusionPerformanceElapsedTime)
{
    logPerformanceAudioOcclusionMin = audioOcclusionPerformanceElapsedTime;
}
if (logPerformanceAudioOcclusionMax < audioOcclusionPerformanceElapsedTime)
{
    logPerformanceAudioOcclusionMax = audioOcclusionPerformanceElapsedTime;
}

logPerformanceApplicationAverage += applicationPerformanceElapsedTime;
logPerformanceAudioOcclusionAverage += audioOcclusionPerformanceElapsedTime;
```
logPerformanceSampleCount++;

PathfinderManagerUI.instance.logPerformanceButtonText.text = PathfinderManagerUI.instance.

logPerformanceButtonTextStartString + "; " + 
logPerformanceTimer.ToString("F2") + "/ " + 
logPerformanceDuration.ToString("F2");

if (logPerformanceTimer >= logPerformanceDuration)
{
    logPerformance = false;

    logPerformanceApplicationAverage /= logPerformanceSampleCount;

    logPerformanceAudioOcclusionAverage /= logPerformanceSampleCount;

    string performanceLog = string.Empty;

    string performanceLogSavePath = 

        Path.Combine(Application.persistentDataPath,

        "Performance Log (" + DateTime.Now.
        ToString("MM-dd-yyyy hh-mm-ss") + ").txt");

    string mode = string.Empty;

    if (!enablePathfinding)
    {
        mode = "NONE";
    }
    else
    {
        mode = currentPathfinder.ToString();
    }

    performanceLog += "Duration: " + Math.Round(
        logPerformanceDuration, 2).ToString("F2") + " s\n\n";

    performanceLog += "Mode: " + mode + "\n\n";
performanceLog += "Total Runtime Average: " + Math.Round(
    logPerformanceApplicationAverage, 2).ToString("F2") + 
    " ms (" + Math.Round(1000.0d / logPerformanceApplicationAverage, 0) + 
    " FPS)\n";

performanceLog += "Total Spatial Audio Average: " + Math.Round(
    logPerformanceAudioOcclusionAverage, 2).ToString("F2") + 
    " ms (" + Math.Round(1000.0d / logPerformanceAudioOcclusionAverage, 0) + 
    " FPS)\n\n";

performanceLog += "Runtime Min: " + Math.Round(
    logPerformanceApplicationMin, 2).ToString("F2") + 
    " ms (" + Math.Round(1000.0d / logPerformanceApplicationMin, 0) + " FPS)\n";

performanceLog += "Runtime Max: " + Math.Round(
    logPerformanceApplicationMax, 2).ToString("F2") + 
    " ms (" + Math.Round(1000.0d / logPerformanceApplicationMax, 0) + " FPS)\n"
);

File.WriteAllText(performanceLogSavePath, performanceLog);

UnityEngine.Debug.Log("Performance Log saved to: " + 
    performanceLogSavePath);

PathfinderManagerUI.instance.logPerformanceButtonText.text = 
    PathfinderManagerUI.instance.logPerformanceButtonTextStartString;
audioOcclusionPerformanceStopwatch.Reset();

public void SetNextPathfinder()
{
    instance.currentPathfinder =
        (AudioOcclusion_PathDifference.Pathfinder)
        ((int)(instance.currentPathfinder + 1) %
        System.Enum.GetNames(
            typeof(AudioOcclusion_PathDifference.Pathfinder)).Length);

    instance.UpdateToCurrentPathfinder();
}

public void ToggleEnablePathfinding()
{
    instance.enablePathfinding = !instance.enablePathfinding;
    instance.UpdateToCurrentPathfinderEnabledState();
}

void UpdateToCurrentPathfinderEnabledState()
{
    AudioOcclusion_PathDifference[] objects =
        Resources.FindObjectsOfTypeAll<AudioOcclusion_PathDifference>();

    for (int i = 0; i < objects.Length; i++)
    {
        objects[i].enabled = enablePathfinding;
    }

    if (!enablePathfinding)
    {
        cpuPathfinder.gameObject.SetActive(false);
        gpuPathfinder.gameObject.SetActive(false);
    }
    else
    {
        switch (currentPathfinder)
        {
            case AudioOcclusion.Pathfinder.NavMeshCPU:
cpuPathfinder.gameObject.SetActive(true); break;

case AudioOcclusion.Pathfinder.VoxelGraphGPU:
    gpuPathfinder.gameObject.SetActive(true);
    break;

default:
    throw new System.Exception("Unknown enum.");
}

void UpdateToCurrentPathfinder()
{
    switch (currentPathfinder)
    {
        case AudioOcclusion.Pathfinder.NavMeshCPU:
            
            gpuPathfinder.gameObject.SetActive(false);
            cpuPathfinder.gameObject.SetActive(true);

            currentPathfinder = AudioOcclusion.Pathfinder.NavMeshCPU;
            break;

        case AudioOcclusion.Pathfinder.VoxelGraphGPU:
            
            cpuPathfinder.gameObject.SetActive(false);
            gpuPathfinder.gameObject.SetActive(true);

            currentPathfinder = AudioOcclusion.Pathfinder.VoxelGraphGPU;
            break;

        default:
            throw new System.Exception("Unknown enum.");
    }
}
AudioOcclusion_PathDifference[] objects = Resources.FindObjectsOfTypeAll<AudioOcclusion_PathDifference>();

for (int i = 0; i < objects.Length; i++){
    objects[i].pathfinder = currentPathfinder;
}

public void LogPerformance()
{
    logPerformance = true;
    logPerformanceSampleCount = 0;
    logPerformanceTimer = 0.0f;
    logPerformanceApplicationMin = double.MaxValue;
    logPerformanceApplicationMax = 0.0d;
    logPerformanceAudioOcclusionMin = double.MaxValue;
    logPerformanceAudioOcclusionMax = 0.0d;
    logPerformanceApplicationAverage = 0.0d;
    logPerformanceAudioOcclusionAverage = 0.0d;
}

*Code 12: The pathfinder manager used to assert global control over the spatial audio pathfinding configuration.*

*Features an integrated profiler for the system.*

The parameters recorded were the average execution time (calculated as the sum of values over the recording duration divided by the number of frames executed), as well as the minimum and maximum execution times. From these I derived the rounded integer frame rates. All base measurements were calculated in milliseconds using the high precision *Stopwatch* class (see [176]). Once started (a manual process), the profiler will collect runtime data until the set duration runs out and the profiler terminates recording automatically. At the end of the
recording, the measurements are written out to a timestamped file on the device with the duration and spatial audio pathfinder mode (see Figure 28). The base measurement values written to file are rounded to two decimal places (internally, they are doubles with decimals to 14 places, which is excessive to record to a file). The derived frames per second (FPS) values are calculated using the original doubles so they remain an accurate representation of the performance, although this may sometimes give the impression of miscalculation in the output file. As an example from Figure 28, the “Total Spatial Audio Average” execution time is recorded as 0.87 ms, which should resolve to a rounded 1149 FPS (1000.0 / 0.87), however, the measurement is also rounded to two decimal places when printed, and was likely closer to 0.8675…, which would produce the 1153 FPS that’s recorded (1000.0 / 0.8675 = 1152.7…).

![Profiler Output](image)

*Figure 28: A sample profiler output with all the recorded and calculated measurements.*

All performance tests were recorded with a duration of 60.0 s. Please note that even with the frame rate of the application unlocked, the maximum frames per second that can be executed by mobile devices is fixed to 60, 30, and lower due to mandatory vertical synchronization (see
for a detailed article and explanation). A user interface for controlling the pathfinder manager parameters and functions was also created for mobile devices (see Figure 29).

Figure 29: The pathfinder manager in-game UI used for touch/mouse-based control over the global spatial audio pathfinding settings.

The test scene consisted of 10 sound sources spread out in an area of 100 m$^3$ with a player size to scale of about 2.0 m. The player moved at the maximum speed of 10 m/s at all times for the GPU tests with dynamic obstacle delay values (when enabled) set to either 0.125 s or 0.0 s (indicated in the benchmark tests). There was one physics-enabled dynamic object in the scene as a large 4 m$^3$ cube to ensure the dynamic obstacle part of the pathfinder would execute. The actual playable area was 100 m × 10 m × 100 m, but the full 100 m$^3$ node block was processed (see Figure 30).
Figure 30: My benchmarking level in Unity’s editor with 10 sound sources. Sound sources are visible as small solid spheres inside larger, yellow-tinted wireframe spheres.
The uniform node size was fixed to 1.0 m. Table 2 provides a description of the systems used for all runtime performance tests, while Table 3 lists the results. Not all of the systems supported the GPU pathfinder due to hardware restrictions.
<table>
<thead>
<tr>
<th>ID</th>
<th>Pathfinder</th>
<th>RT Min</th>
<th>RT Max</th>
<th>SA Min</th>
<th>SA Max</th>
<th>RT Avg.</th>
<th>SA Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>CPU</td>
<td>0.56 ms (1774 FPS)</td>
<td>3.88 ms (258 FPS)</td>
<td>0.23 ms (4264 FPS)</td>
<td>2.51 ms (399 FPS)</td>
<td>0.90 ms (1107 FPS)</td>
<td>0.26 ms (3884 FPS)</td>
</tr>
<tr>
<td>A</td>
<td>GPU (Static)</td>
<td>0.78 ms (1275 FPS)</td>
<td>16.67 ms (60 FPS)</td>
<td>0.37 ms (2711 FPS)</td>
<td>16.02 ms (62 FPS)</td>
<td>1.32 ms (758 FPS)</td>
<td>0.85 ms (1177 FPS)</td>
</tr>
<tr>
<td>A</td>
<td>GPU (Dynamic 0.125)</td>
<td>0.80 ms (1257 FPS)</td>
<td>21.93 ms (46 FPS)</td>
<td>0.39 ms (2552 FPS)</td>
<td>19.94 ms (50 FPS)</td>
<td>1.57 ms (638 FPS)</td>
<td>1.09 ms (916 FPS)</td>
</tr>
<tr>
<td>A</td>
<td>GPU (Dynamic 0.0)</td>
<td>12.06 ms (83 FPS)</td>
<td>20.20 ms (49 FPS)</td>
<td>11.25 ms (89 FPS)</td>
<td>18.55 ms (54 FPS)</td>
<td>16.07 ms (62 FPS)</td>
<td>14.99 ms (67 FPS)</td>
</tr>
<tr>
<td>B</td>
<td>CPU</td>
<td>57.86 ms (17 FPS)</td>
<td>75.66 ms (13 FPS)</td>
<td>0.60 ms (1674 FPS)</td>
<td>1.65 ms (604 FPS)</td>
<td>66.34 ms (15 FPS)</td>
<td>0.65 ms (1549 FPS)</td>
</tr>
<tr>
<td>C</td>
<td>CPU</td>
<td>0.83 ms (1204 FPS)</td>
<td>18.62 ms (54 FPS)</td>
<td>0.33 ms (3018 FPS)</td>
<td>9.42 ms (108 FPS)</td>
<td>2.21 ms (453 FPS)</td>
<td>0.41 ms (2457 FPS)</td>
</tr>
<tr>
<td>C</td>
<td>GPU (Static)</td>
<td>170.17 ms (6 FPS)</td>
<td>458.15 ms (2 FPS)</td>
<td>167.65 ms (6 FPS)</td>
<td>454.33 ms (2 FPS)</td>
<td>279.33 ms (4 FPS)</td>
<td>275.76 ms (4 FPS)</td>
</tr>
<tr>
<td>C</td>
<td>GPU (Dynamic 0.125)</td>
<td>172.55 ms (6 FPS)</td>
<td>474.79 ms (2 FPS)</td>
<td>169.35 ms (6 FPS)</td>
<td>464.07 ms (2 FPS)</td>
<td>291.59 ms (3 FPS)</td>
<td>287.31 ms (3 FPS)</td>
</tr>
<tr>
<td>C</td>
<td>GPU (Dynamic 0.0)</td>
<td>171.83 ms (6 FPS)</td>
<td>473.42 ms (2 FPS)</td>
<td>169.09 ms (6 FPS)</td>
<td>469.05 ms (2 FPS)</td>
<td>292.91 ms (3 FPS)</td>
<td>288.61 ms (3 FPS)</td>
</tr>
<tr>
<td>D</td>
<td>CPU</td>
<td>6.83 ms (146 FPS)</td>
<td>49.33 ms (20 FPS)</td>
<td>0.65 ms (1532 FPS)</td>
<td>23.05 ms (43 FPS)</td>
<td>16.78 ms (60 FPS)</td>
<td>1.27 ms (790 FPS)</td>
</tr>
<tr>
<td>D</td>
<td>GPU (Static)</td>
<td>13.15 ms (76 FPS)</td>
<td>980.62 ms (1 FPS)</td>
<td>5.42 ms (184 FPS)</td>
<td>961.99 ms (1 FPS)</td>
<td>534.07 ms (2 FPS)</td>
<td>519.63 ms (2 FPS)</td>
</tr>
<tr>
<td>D</td>
<td>GPU (Dynamic 0.125)</td>
<td>529.80 ms (2 FPS)</td>
<td>1195.18 ms (1 FPS)</td>
<td>515.32 ms (2 FPS)</td>
<td>1181.29 ms (1 FPS)</td>
<td>817.41 ms (1 FPS)</td>
<td>799.50 ms (1 FPS)</td>
</tr>
<tr>
<td>D</td>
<td>GPU (Dynamic 0.0)</td>
<td>493.09 ms (2 FPS)</td>
<td>1272.30 ms (1 FPS)</td>
<td>478.23 ms (2 FPS)</td>
<td>1253.31 ms (1 FPS)</td>
<td>863.54 ms (1 FPS)</td>
<td>845.64 ms (1 FPS)</td>
</tr>
<tr>
<td>E</td>
<td>CPU</td>
<td>8.21 ms (122 FPS)</td>
<td>30.56 ms (33 FPS)</td>
<td>0.68 ms (1480 FPS)</td>
<td>3.33 ms (300 FPS)</td>
<td>16.76 ms (60 FPS)</td>
<td>0.98 ms (1022 FPS)</td>
</tr>
<tr>
<td>F</td>
<td>CPU</td>
<td>7.70 ms (130 FPS)</td>
<td>39.91 ms (25 FPS)</td>
<td>0.92 ms (1090 FPS)</td>
<td>8.69 ms (115 FPS)</td>
<td>16.69 ms (60 FPS)</td>
<td>1.06 ms (943 FPS)</td>
</tr>
</tbody>
</table>

*Table 3: A summary of all performance tests in the benchmark environment, where RT = Runtime Total, SA = Spatial Audio. Lower is better (less time to execute equates to higher FPS).*
5.7 Results Analysis

As indicated by the SA average, none of the systems tested had any problems rendering out the approximated environmental spatial audio using the CPU pathfinder component, which specifically targeted underpowered hardware.

With the most modern hardware specifications of those tested, System A significantly outperformed all other systems and had no trouble executing and updating the GPU pathfinder every frame while maintaining an average of just over 60 fps. At its worst, the GPU-based spatial audio part took 18.55 ms, and at its best, it took 11.25 ms while engaging and exchanging buffer data with the GPU. If dynamic obstacles updates are throttled to execute at a fixed frequency of eight times per second (every 0.125 s), the average performance jumped significantly to over 700 fps for the RT, and over 900 FPS for just the SA. With limited dynamic updates, the SA Max was 19.94 ms, which is comparable to the SA Max with full dynamic updates every frame timing in at 18.55 ms. This is expected, as the amount of data to process will be nearly identical, except that updates will execute at a lower frequency. This is also why the SA Min was much lower for GPU (Dynamic 0.125) at 0.39 ms than for GPU (Dynamic 0.0), since during the frames when dynamic updates are not being processed, the GPU pathfinder behaves the same as if all geometry is static. The SA min for GPU (Static) confirms this with a similar SA Min to GPU (Dynamic 0.125). The GPU on System A, the GTX 1070, is a fairly high end gaming GPU of the current generation.

System B had worst overall performance (RT Average), being outperformed by even the mobile and laptop platforms. This is likely the result of having an underpowered integrated motherboard GPU intended for low-cost, mainstream PC design [178] [179]. Regardless, all SA runtimes were
still very low, where even the SA Max was only 1.65 ms since the bottleneck for this system was the GPU.

System C’s relative GPU runtimes (runtimes relative to each other within the same system) are similar to System A, although overall this computer also had abysmal performance running the GPU-based spatial audio. Unity’s profiler revealed that most of this was a result of the host-device transfer speed, and less so the actual calculations being performed for pathfinding on the GPU. The GPU itself (NVS 4200M) is an older business-class GPU for laptops.

System D is the only mobile/smartphone tested that was capable of running the GPU pathfinder, although similar to System C, it was not performant to the point of being useful in an actual game that would require sustaining 30-60 FPS. Unlike System A and C, both of which had relatively comparable SA Min values between GPU (Static) and GPU Dynamic (0.125), System D’s SA Min for these two measurements were significantly different (5.42 ms and 515.32 ms respectively). As mentioned earlier, when the dynamic geometry part of the GPU pathfinder is inactive, it behaves identical to having only static geometry, so the SA Min between GPU (Static) and GPU Dynamic (0.125) should be similar for System D. The large gap between the runtime values is most likely due to System D’s overall low performance with the GPU pathfinder. Of the three systems that were capable of GPU-accelerated spatial audio processing (A, C, and D), System D had the worst performance by a large margin. Instead of the GPU (Dynamic 0.125) measurement being similar to GPU (Static), it’s much closer to GPU (Dynamic 0.0), indicating that it was never able to “rest” between frames, since by the time the next frame could be processed, over 1/8th of a second had passed. Even though System C did not have this issue, it’s possible that with
multiple runs, eventually there would be an output reporting a lack of a rest period such that both dynamic GPU measurements would be similar to each other and farther away from GPU (Static). Note that System C’s fastest SA FPS measurement was at 6 FPS, compared to System D’s 2 FPS, resulting in System D to have a higher likelihood of missing a rest frame. The significantly lower GPU (Static) SA Min can be explained by considering the value being in range of the CPU SA Max. While there’s no movement for GPU (Static), no recalculation is performed on the GPU, and any overhead is from having the GPU pathfinder pipeline running (see Code 5). This is reported in Unity’s profiler, and can be inferred visually on both System C and D, which don’t produce frame-lag if simply idling without character movement across graph cells (during which the benchmark total is at least 60 FPS), but then suddenly produce lag when moving. During the benchmark, the character is supposed to be moving at an average of 10 cells/s, but in one frame, due to the time delta between frames, the character may not have moved enough to engage the GPU via triggering a recalculation and graph update.

Systems E and F both had similar values, although System E was somewhat faster in most of the measurements, with noticeable improvements for SA Min and SA Max.

For reference regarding scalability, Table 4 is provided below which uses the same level and parameters, except that the boundary walls of the benchmark environment have been removed. All DSP processor components and sound sources have been removed so that only GPU pathfinding requests remain. There is no dynamic geometry, and the player is being moved automatically at a constant speed of 10 m/s using a fixed waypoint system. Path requests are generated randomly before performance logging begins at the press of a key (see the top of Code
5 for the generator). Note that runtimes do not scale linearly with the number of path requests, since this is based on the size and density of the domain. It isn’t until a significant factor of additional paths that the system shows significant measurement differences as a result of many kernel dispatches to retrieve and calculate paths (more data transfer required between host to device for path data).

<table>
<thead>
<tr>
<th>ID</th>
<th>Pathfinder</th>
<th>Paths</th>
<th>RT Min</th>
<th>RT Max</th>
<th>RT Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>GPU (Static)</td>
<td>10</td>
<td>0.75 ms (1332 FPS)</td>
<td>20.42 ms (49 FPS)</td>
<td>1.19 ms (838 FPS)</td>
</tr>
<tr>
<td>A</td>
<td>GPU (Static)</td>
<td>20</td>
<td>0.77 ms (1297 FPS)</td>
<td>20.82 ms (48 FPS)</td>
<td>1.19 ms (839 FPS)</td>
</tr>
<tr>
<td>A</td>
<td>GPU (Static)</td>
<td>100</td>
<td>1.13 ms (885 FPS)</td>
<td>20.77 ms (48 FPS)</td>
<td>1.73 ms (577 FPS)</td>
</tr>
<tr>
<td>A</td>
<td>GPU (Static)</td>
<td>200</td>
<td>1.08 ms (928 FPS)</td>
<td>20.52 ms (49 FPS)</td>
<td>1.65 ms (605 FPS)</td>
</tr>
<tr>
<td>A</td>
<td>GPU (Static)</td>
<td>1000</td>
<td>3.31 ms (302 FPS)</td>
<td>24.20 ms (41 FPS)</td>
<td>4.99 ms (200 FPS)</td>
</tr>
<tr>
<td>A</td>
<td>GPU (Static)</td>
<td>2000</td>
<td>6.46 ms (302 FPS)</td>
<td>28.06 ms (41 FPS)</td>
<td>9.06 ms (110 FPS)</td>
</tr>
<tr>
<td>A</td>
<td>GPU (Static)</td>
<td>10000</td>
<td>29.79 ms (34 FPS)</td>
<td>56.42 ms (18 FPS)</td>
<td>39.42 ms (25 FPS)</td>
</tr>
</tbody>
</table>

Table 4: A reference performance table measuring total runtimes with only the GPU pathfinder’s path requests.

5.8 Summary

This chapter provided an initial evaluation of a new system that I developed for approximating sound propagation through arbitrarily complex, dynamic environments using pathfinding at its core with GPGPU-accelerated data processing. The benchmarking results showed that, even when processing level data fully in 3D, it is capable of running at interactive rates with headroom
left on the CPU for processing other important game tasks such as physics, AI, and more. With the inclusion of a lighter fallback method that processes the environment in 2.5D using only the CPU, the system is highly scalable across a wide variety of hardware and can be deployed to many different platforms.
Chapter 6: The Game

6.1 Overview

To test and confirm the applicable performance of my system in a VR environment, I developed a fully functional 3D first-person stealth/puzzle/adventure video game optimized for deployment using my system on both mobile, and desktop/VR platforms (using either the CPU or GPU implementation of my acoustics propagation approximation respectively) capable of using virtual reality headsets such as the Oculus Rift or HTC Vive. The game places an emphasis on both interactive and passive immersion through the purposeful manipulation of sound and light in its gameplay mechanics across multiple levels. This chapter discusses the details of the game and its mechanics.

6.2 Gameplay

The bulk of the gameplay takes place in a darkened underground labyrinth faintly lit by incandescent light bulbs and torches which subtly illuminate several “compartments” of the dungeon-like environment (see Figure 31). All areas of the game are smaller than the benchmarking environment and runtime performance is smooth even with physics, artificial intelligence (AI), lighting, and other many other active game-relevant calculations.
Figure 31: Part of the level in which most of the game takes place.

The player must safely navigate the labyrinth using the faint cues of light and propagated sound to solve puzzles, find/collect keys, and stealthily avoid a dangerous lingering enemy AI which can detect the player’s movements (including in-game interactions) using simulated hearing and sight. To escape to the next area, they must use the keys they find on chests scattered throughout the level without alerting the enemy and then activate a “teleportation monolith” using the items from the chests (see Figure 32). I designed the game in a way such that it’s often vital to stop and observe cues in the environment – such as sound and light sources – to better understand the game state.
Players can walk, run, crouch (“sneak”), jump, peek around corners, climb ladders, trigger events, and manipulate dynamic physics-based objects in the game levels such as boxes, barrels, doors, and chests. Players may also pick up items, rotate and adjust the distance, and then carefully place them back into the world or throw them – where the interaction system accounts for the mass of the object being manipulated (see Figure 33).
This can be useful for creating makeshift climbable surfaces by stacking large and stable objects such as crates or barrels to reach elevated areas (one such “puzzle” involves finding an important letter out of reach above a closet). The game takes the mouse and keyboard as primary inputs, with an optional on-screen display serving as a virtual gamepad for touchscreen systems, such as mobile platforms (see Figure 34 for a screen capture of the touchscreen control interface).
Figure 34: The game’s touchscreen control system, featuring a move pad and interaction buttons.

Depending on the context of the input and the player’s current state, the same limited set of controls will function differently, allowing a broader realization of interactions. An example of this is looking around the game level using the default controls, for which an alternative state will run where the same control scheme can be used to finely rotate objects instead of the player’s virtual head orientation. Another example is object interaction, where a simple one-off event may be used to push buttons on 3D world interfaces (a virtual pad), or move away from reading a letter being presented to the player full screen.

6.3 3D Sound and Spatialization

All sounds emitted in 3D are directionally spatialized with HRTF using Unity’s native Oculus integration. This does not include environmental propagation. The keys, the enemy AI, and other select sources in the game use my propagation system to convey additional important
information about the game state using sound, such as obstruction and occlusion, which allows players to “hear” what they’re looking to find (or avoid…) by tracking sounds around corridors. However, the enemy AI uses a similar system to “listen” to what the player is up to, and taking into account how the sound should propagate path-wise (see Code 13), it may attempt to navigate to any disturbances it detects.

```csharp
void ScanForNoise()
{
    List<WorldSoundSourceMonitor.SoundSource> sounds =
        WorldSoundSourceMonitor.instance.sounds;

    perceivedSounds.Clear();

    for (int i = 0; i < sounds.Count; i++)
    {
        WorldSoundSourceMonitor.SoundSource sound = sounds[i];

        float distanceToSound = GetPathLengthToPosition(sound.position);

        if (distanceToSound < hearingRadius)
        {
            // Inverse relationship between volume and distance.
            // Greater distance -> lower perceived volume.

            float perceivedVolume =
                sound.volume * (1.0f - (distanceToSound / hearingRadius));

            if (perceivedVolume > investigateSoundSourcePerceivedVolumeThreshold)
            {
                perceivedSounds.Add(sound);

                print("Enemy detected an unseen sound source " +
                    distanceToSound + "m away @ volume " + perceivedVolume + ".");
            }
        }
    }

    WorldSoundSourceMonitor.SoundSource loudestPerceivedSound =
        perceivedSounds.OrderByDescending(x => x.volume).First();
}```
agent.destination = loudestPerceivedSound.position;

Code 13: The enemy AI's listening behaviour function.

The listening behaviour reads in all disturbances logged to a global sound source “monitor”,
which holds the position and initial volume of the source (see Code 14). Player footsteps and
world interactions are added to the monitor. The player can take advantage of this behaviour to
deceive the enemy AI by purposefully creating noise in one area and then sneaking away as it
approaches to investigate. This allows the player to continue exploring the area which the AI was
patrolling earlier.

```csharp
public class WorldSoundSourceMonitor : MonoBehaviour
{
    public struct SoundSource
    {
        public float volume;
        public Vector3 position;
    }

    public static WorldSoundSourceMonitor instance;

    public List<SoundSource> sounds = new List<SoundSource>();

    void Awake()
    {
        instance = this;
        StartCoroutine(ClearSounds());
    }

    void Update()
    {
    }

    public void AddSoundSource(WorldSoundSource source)
    {
        AddSoundSource(source.transform.position, source.volume);
    }
```
Almost all interactions in the game produce 3D spatialized sounds based on physics such as the player running into and against walls or other objects, and object-to-object collisions such as impacts and scrapes (accounting for relative velocity and pressure). In addition to providing a constant stream of information through sound to the player, this has additional gameplay consequences in the underground labyrinth where this information can also be decoded by the enemy AI. Violent and sudden interactions with objects in the environment, such as harshly swinging open doors and chests (both featuring squeaky hinges), throwing objects, or running into objects at high speeds will produce loud noises – possibly attracting unwanted attention (see Figure 35). The enemy has spatialized and propagated footsteps, growls, “barks” (one-off speech clips commonly used by AI in games to alert players to the AI’s current state), and leaves a trail of emissive particles. Additionally, its eyes glow and a spotlight attached to the head brightens anything the enemy is looking at, such as walls as it approaches closer. The player should use
these cues to remain aware of the AI’s position at all times. If the enemy AI catches the player, they are killed and must restart the level.

Figure 35: Being pursued by the enemy AI.

Although not inherently a part of the main propagation system, there is a unique feature in the game which allows players to “communicate” with the enemy AI using their own voice and a microphone. Players may project their voice into the game, and it will be propagated and potentially heard by the AI using the same system for any other sound made in the game that could catch the AI’s attention. The sound will be played back in the game world with all the spatial cues present, including HRTFs and environmental filters. This is similar to “EAX Voice”, which was capable of playing back external sounds in the game as if it was positioned in the actual game environment.
During runtime, my propagation system can be switched between being entirely inactive (using Unity’s default implementation), using only the CPU for 2.5D processing, or additionally taking advantage of the GPU to simulate fully 3D propagation acoustics so that the differences can be observed.

6.4 Performance

Runtime measurements were performed using the same custom profiler as described in the last chapter. The second level of the game was tested because it contained the largest and most complex design with the most interactions and longest likely gameplay length. All measurements were recorded over a duration of 60.0 s. The same systems were used (A to F) and the GPU pathfinder was benchmarked when possible, with the dynamic frequency set to run every 0.125 s (the gameplay did not require dynamic geometry updates every frame). The playable area’s bounding volume for the level was approximately 50 m × 15 m × 60 m. The node size was set to 0.48 m to accommodate finer details in the level geometry than that of the benchmark level. Finally, the player moved at a variable rate between no movement, crouching slow/fast, walking, and running at full speed. This results in a move speed range between 0.0 m/s to 4 m/s while performing any of the interactions that were part of the level.
<table>
<thead>
<tr>
<th>ID</th>
<th>Pathfinder</th>
<th>RT Min</th>
<th>RT Max</th>
<th>SA Min</th>
<th>SA Max</th>
<th>RT Avg.</th>
<th>SA Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>CPU</td>
<td>1.66 ms (603 FPS)</td>
<td>8.87 ms (113 FPS)</td>
<td>0.60 ms (1671 FPS)</td>
<td>4.92 ms (203 FPS)</td>
<td>2.09 ms (479 FPS)</td>
<td>0.71 ms (1416 FPS)</td>
</tr>
<tr>
<td>A</td>
<td>GPU (Dynamic 0.125)</td>
<td>2.47 ms (404 FPS)</td>
<td>15.06 ms (66 FPS)</td>
<td>0.77 ms (1294 FPS)</td>
<td>13.41 ms (75 FPS)</td>
<td>3.50 ms (286 FPS)</td>
<td>2.01 ms (497 FPS)</td>
</tr>
<tr>
<td>B</td>
<td>CPU</td>
<td>262.69 ms (4 FPS)</td>
<td>586.44 ms (2 FPS)</td>
<td>2.15 ms (466 FPS)</td>
<td>14.73 ms (68 FPS)</td>
<td>399.12 ms (3 FPS)</td>
<td>3.16 ms (316 FPS)</td>
</tr>
<tr>
<td>C</td>
<td>CPU</td>
<td>3.67 ms (273 FPS)</td>
<td>40.83 ms (24 FPS)</td>
<td>0.88 ms (1130 FPS)</td>
<td>42.80 ms (40 FPS)</td>
<td>15.04 ms (66 FPS)</td>
<td>1.36 ms (735 FPS)</td>
</tr>
<tr>
<td>C</td>
<td>GPU (Dynamic 0.125)</td>
<td>144.83 ms (7 FPS)</td>
<td>201.64 ms (5 FPS)</td>
<td>132.46 ms (8 FPS)</td>
<td>193.09 ms (5 FPS)</td>
<td>166.81 ms (6 FPS)</td>
<td>161.09 ms (6 FPS)</td>
</tr>
<tr>
<td>D</td>
<td>CPU</td>
<td>7.48 ms (134 FPS)</td>
<td>74.66 ms (13 FPS)</td>
<td>2.12 ms (472 FPS)</td>
<td>28.21 ms (35 FPS)</td>
<td>17.05 ms (59 FPS)</td>
<td>3.46 ms (289 FPS)</td>
</tr>
<tr>
<td>D</td>
<td>GPU (Dynamic 0.125)</td>
<td>426.60 ms (2 FPS)</td>
<td>623.78 ms (2 FPS)</td>
<td>396.82 ms (3 FPS)</td>
<td>570.02 ms (2 FPS)</td>
<td>531.89 ms (2 FPS)</td>
<td>476.69 ms (2 FPS)</td>
</tr>
<tr>
<td>E</td>
<td>CPU</td>
<td>8.55 ms (117 FPS)</td>
<td>38.14 ms (26 FPS)</td>
<td>2.24 ms (446 FPS)</td>
<td>23.06 ms (43 FPS)</td>
<td>16.76 ms (60 FPS)</td>
<td>3.18 ms (315 FPS)</td>
</tr>
<tr>
<td>F</td>
<td>CPU</td>
<td>9.32 ms (107 FPS)</td>
<td>85.71 ms (12 FPS)</td>
<td>2.47 ms (405 FPS)</td>
<td>17.72 ms (56 FPS)</td>
<td>23.00 ms (43 FPS)</td>
<td>5.01 ms (200 FPS)</td>
</tr>
</tbody>
</table>

Table 5: A summary of all performance tests in the game where RT = Runtime Total, SA = Spatial Audio. Lower is better (less time to execute equates to a higher FPS).

The results are largely similar to that of the previous chapter’s benchmark level, with a noticeable increase in the RT Average as a result of increased overall environment complexity (graphics quality, particle systems, physics, player calculations, interactions, etc.). This is not necessarily evidenced in Table 5, but Unity’s profiler reported increased times used by several components unrelated to the spatial audio system. Regardless, the SA times are also worse for both the CPU and GPU spatial audio systems than the benchmark level. The explanation for the CPU system can be attributed to the greatly increased NavMesh complexity (see Figure 36).
The GPU system is also less performant than the benchmark level due to a significantly higher node density, even though it has a smaller bounding volume area. The benchmark level node size was 1.0 m, but for the second game level it’s less than half of that at 0.48 m. If the node size wasn’t changed, the total number of nodes would be the same as the bounding volume in m$^3$, which is 45,000. However, because of the reduced size and the square-cube law, the actual total node count is 403,000 and the density is about eight times as high. This means that as the player moves at full speed (4 m/s), they may trigger up to ~16 (4 × 4) GPU graph recalculations / updates per second with forward movement even with static geometry, as opposed to 10 for the benchmark level. Updates may be faster for ungrounded movement where the player may accelerate downwards from the effects of gravity (for example, after jumping off the ground, stairs, stacked boxes, etc.).
6.5 Summary

The game discussed in this chapter is a complete demonstration of how my new system for approximated sound propagation can be used in an actual development scenario. The game itself strongly emphasizes play-by-sound mechanics to the point of even allowing players to catch the attention of a lingering enemy AI using their actual voice through a microphone, which is then projected into the game world and propagated to the AI as if it was originally sourced from inside the virtual environment. Many of the concepts from previous chapters are integrated into this game, such as the dynamic, context-sensitive control scheme which changes how feedback is rendered to the game based on the player’s current state. The same controls that are used to pick up and throw objects may be used to trigger one-off events such as pushing buttons and exiting from reading letters in a full screen interface. My game is deployable to many of the platforms supported by Unity, and is specifically designed to run smoothly on mobile, PC, and VR.
Chapter 7: Discussion and Conclusions

7.1 Overview

So far, I have discussed the importance of game audio and its relevance to establishing presence in virtual environments. I then discussed the physics and physiology of sound perception in humans, and how this became an increasingly meaningful aspect of games as they evolved from artistic 2D to more realistic 3D, first-person experiences. During this time, several new libraries and systems emerged that helped developers implement more immersive interactive sound, which I compared. Finding a deficiency for a highly cross-platform, generic solution for dynamic occlusion and obstruction that was more advanced than the simple raycast and binary techniques most often used in video games, I developed my own solution that circumvented many of the current limitations to provide a perceptual approximation of spatial cues from the environment, including occlusion, reflection, and diffraction. Leveraging the massively parallel nature of modern GPUs for processing large amount of data with a built-in CPU fallback, I was able to integrate a hardware-independent sound propagation system into a cross-platform game as a demonstration of the viability of this implementation. This last chapter will be a review of the major contributions that are a product of this thesis and their significance regarding future research and developer implementation.

7.2 Discussion

Driven by the popularity of VR and the desire to increase immersion in virtual environments, game sound is developing rapidly. There is a growing trend towards alleviating the high
calculation loads required for expensive calculations from the CPU, to the massively parallel GPUs of today. This is seen in the work carried out by AMD for their original TrueAudio and TrueAudio Next DSP implementations, as well as with Nvidia’s VRWorks audio.

While these libraries offer both realistic and high performance sound propagation, they come at the cost of isolating developers to proprietary hardware, which can be a limiting factor for adoption when not creating hardware-specific VR experiences. Similar to the GPU-accelerated sound propagation solutions from AMD and Nvidia, Steam Audio and Resonance Audio both offer many of the features from some of the most powerful and well-known acoustics simulation systems such as EAX and A3D. And although they don’t provide the same level of accuracy or potential for dynamic environments as the latter systems, they are more cross-platform and easily integrated into existing game engines. This can be a vital factor for developers when considering the time required for integration into their games and experiences, as well as usability and application stability for end-users. Steam Audio has even opted to integrate TrueAudio Next for the purpose of accelerated convolution filtering for those with the more recent AMD GPUs, staying true to the trend of GPGPU sound processing.

Following in the footsteps of these popular libraries, I created a new system that addresses the weak dynamic occlusion and environment obstruction filtering of the Steam Audio and Resonance Audio libraries. By considering existing sound propagation pathfinding methods that have been used in popular games for play-by-sound mechanics, such as Overwatch and Rainbow Six Siege, I accelerated the process for dynamic and fully 3D environments through the use of modern GPGPU compute technology that is not limited to a specific brand of hardware. As a fall
back, the system can still use the CPU only for easy cross-platform deployment to mobile and other lower-end hardware systems (more so than either Steam Audio or Resonance Audio).

My system requires minimal setup, effectively to the point of simply “tagging” sound sources with a custom Unity game engine component that makes the pathfinding requests, while a global manager defines the overall pathfinding variables to use at runtime. To ensure that this system is viable in the type of games and experiences that would most likely use it (first-person games with play-by-sound mechanics), I created a cross-platform game ready for deployment on mobile, PC, and VR. The mechanics in my game mirror those found in commercially available stealth-genre titles. I used similar methods as those used in my system to not only propagate sound to the listener (the player) with perceptual filters for environmental spatial audio, but to also engage the AI by propagating sounds made by player for greater interactivity with the game.

With regards to other complete systems aside from commercially available methods that have been tested in real-time virtual environments (in academia), my system is not as accurate in modelling sound propagation as that of more fine-tuned geometrical acoustics and expensive wave-based acoustical modelling. Instead, I provide a cross-platform, scalable system for modelling occlusion, obstruction, and diffraction effects in real-time that conform to the general behaviour of real-world sound propagation better than common implementations of the same effects found in games. My system is also noticeably faster at runtime, requires no time-consuming offline precomputation stage, and is able to perform the calculations within a single frame without the need for asynchronous delays. From desktops to mobile smartphones, runtime measurements in an isolated benchmark level and complete game environment
demonstrate my system’s low impact on high-end GPUs and scalability to suboptimal hardware when using only the CPU. One smartphone from the newer generation was able to run my GPU simulation at interactive rates. Although the GPU part of my system is still not suitable for mobile when considering large simulation bounds (such as those tested), it’s likely that in the future more powerful mobile GPUs will give way to better GPGPU capability, thus improving performance and the viability of GPU-accelerated spatialized sound for large scenes on smartphones, too.

Some notable examples of other research-oriented environmental spatial audio solutions that incorporate geometrical acoustics, wave-based propagation, and/or perceptual approximations include RESound [39], WAVE [173], and the work done by Cowan and Kapralos [6] [47]. The following is a summary of some of this work as a comparison to my own system.

RESound is a system for fast calculation of sound propagation paths between a source and listener using geometric techniques which take into account specular and diffuse reflections, as well as edge diffraction (based on the UTD). It prioritizes early reflections with a statistical estimation for late reverb. Similar to my system, RESound can handle a moving listener, dynamic sound sources, and dynamic geometry, albeit with some physical restrictions to provide artifact-free audio rendering. RESound’s benchmarking was done on a multi-core PC at 2.66 GHz to run their computation on multiple threads. The fastest propagation runtimes (performed on a room with 6k triangles) measured at 77 ms for first-order specular + diffraction, 359 ms for three orders of specular + diffraction, and 274 ms for three orders of diffuse. The list of calculated paths are accessed asynchronously by the audio rendering pipeline. Similar to my system’s use of per-filter
interpolation for anti-aliasing, each audio frame in RESound is convolved with current and previous IRs and cross-faded to produce the final audio signal.

WAVE (short for “Wave-based Acoustics for Virtual Environments”) is an interactive sound propagation system designed to generate accurate and realistic sound with both dynamic listeners and sources in static 3D environments. It was integrated with the Half-Life 2 game engine and Oculus Rift head-mounted display, and was used for preliminary user evaluation of navigation/task performance when compared to geometric techniques for modelling environmental sound propagation. WAVE supports both omni-directional and directional sound sources. WAVE has a lengthy precomputation stage with a combined CPU and GPU-based runtime evaluation. Larger matrix-vector multiplications required by WAVE are assigned to the GPU, and smaller ones are processed by the CPU. The runtime measurements for WAVE varied between 56 ms to 126 ms, with filters updated asynchronously. Runtime memory requirements varied between 0.2 GB and 15 GB. The computation volume in WAVE’s benchmark tests were 85 m³, and required precomputation times between 210 – 1400 minutes. The WAVE system was implemented on a desktop machine with an Intel Xeon E5-2687W CPU (3.1 GHz), 64 GB of RAM, and an NVIDIA GeForce GTX Titan GPU.

7.3 Future Work

Future work involving user testing would be highly beneficial in order to better understand and analyze the impact of my system on player immersion when compared to what other systems offer as a baseline. This involves performing control and isolation tests with the existing systems mentioned in this thesis that are easy to obtain, such as Steam Audio, Resonance Audio,
VRWorks Audio. For the former two, it’s also possible to use these systems in conjunction with my pathfinding solution for occlusion and obstruction, and opting to disable this feature in the native libraries. Further improvements and research can be done into optimizing the GPU algorithm. This can be through exploring the viability of storing level data in different, more easily traversable and efficient graph-like structures, such as octrees, or simply storing the data offline from a precomputation stage, separate from the runtime part. It may also be worthwhile to explore the potential for my system in creating dynamic experiences for serious games, simulations, and O&M training, without the need for expensive hardware setups like that of Seki and Sato [2].

7.4 Conclusions

Sound in modern video games for all platforms, such as mobile, desktop, and VR, can be made more immersive and interactive through the use of a scalable system for approximated sound propagation and occlusion modelling based on pathfinding. Spatial sound is computationally expensive, especially for real-time use. Existing cross-platform solutions such as Steam Audio and Google Resonance simulate a variety of spatial cues, but both fall short of real-time, dynamic occlusion and propagation effects. A select number of games have used pathfinding to simulate audio propagation and occlusion for better play-by-sound dynamics for desktop and consoles, but with limitations involving static environmental geometry or spatial dimensions. By exploiting the massively parallel nature of today’s high-end GPUs, I developed and implemented a complete game-tested system that can use both the CPU and the GPU as a means to scale between 2.5D and 3D data processing for the required calculations. This method can be used alongside existing audio middleware solutions as a means to fill in the gaps for real-time spatial audio propagation
and occlusion with dynamic/destructible environments. Benchmarking results indicate calculation headroom for other common game routines such as AI, physics, and graphics, even with multiple (possibly thousands of-) active sound source propagation paths being rendered, which I further confirmed through the actual playable game implementation and relevant performance measurements.
References


[Accessed 18 February 2018].


[102] Two Big Ears Ltd, "We’re excited to announce that we’re joining Facebook!," 23 May 2016. [Online]. Available:
[Accessed 19 February 2018].


[Accessed 20 December 2017].


[Accessed 20 December 2017].


