University of Portsmouth

School of Creative Technologies

Final year Project undertaken in partial fulfilment of the requirements for the BSc (Honours) in **Music and Sound Technology**

Open Propagation Library: An Indie Friendly Numerical Audio Simulation Tool

by

James Kelly

845626

Supervisor: Stephen Pearse

Final Year Project

May 2021

Project Type: **Combined**

Abstract

Numerical audio simulations have a long history and a proven track record. But the usage of comprehensive audio simulations in games has received less attention in comparison to the optimised geometric approaches.

Within the last ten years, researchers have developed ways to use numerical simulations in real-time environments like video games. Most notably, Project Acoustics [Raghuvanshi, 2017], a Microsoft research effort that runs a numerical simulation offline, similar to light baking, and uses the salient information extracted from the simulation during runtime. More recently, Planeverb [Rosen et al., 2020] is a tool similar to Project Acoustics but allows for dynamic geometry at the cost of a simplified simulation.

However, Project Acoustics requires the use of Microsoft Azure and only works for Wwise, and Planeverb only works for Unity and the Windows operating system. As there is no easy to access, cross-platform tool, this dissertation creates its own numerical simulation tool called ‘OpenPL’ - Open Propagation Library - that integrates with Wwise, FMOD, Unreal Engine and Unity. Furthermore, OpenPL runs on the user’s local machine, thereby negating the need for a cloud service like Azure.

Contents

Terminology 5

Acknowledgements 6

Introduction 7
- History Of Video Games . . . 9
- Ray Casting Clarified . . . 11
- Ray Casting For Audio . . . 12
- Wave Simulation Explained . . . 14

Literature Review 16

Methodology 18
- Code Structure . . . 18
- C++ And C#Wrapper . . . 19
- External Libraries And Tools . . . 19
- Project Management . . . 20
- Version Control . . . 20

Implementation 21
- Developing On Different Platforms . . . 21
- Setup Script . . . 21
- GitHub Actions Automation . . . 21
- Building A Single DLL . . . 22
- Physics And Voxelising The Scene . . . 22
- Passing Geometry To OpenPL . . . 24
- Graph Rendering . . . 27
- Exporting The Impulse Response . . . 28
- Parameter Extraction . . . 28
- FMOD Unity Integration (Impulse Responses) . . . 29
- FMOD Unity Integration (Parameter Extraction) . . . 31
- Wwise Unity Integration (Impulse Responses) . . . 31
- Wwise Unity Integration (Parameter Extraction) . . . 31
- FMOD Unreal Integration . . . 31
- Wwise Unreal Implementation . . . 33
- Simulation Implementation . . . 33
  - Failed Simulations . . . 40
  - Performance And Speed . . . 41
- Planeverb Codebase Review . . . 42

Conclusion 44

Appendix 47
- Code Base . . . 47

SimulationVersusNoSimulation.mp4 47
RaycastVersusNoSimulation.mp4 47
WwiseUnity.mp4 47
FMODUnreal.mp4 47
WwiseUnreal.mp4 47

List of Figures

1 First Generation Ray Casting Example Similar To Wolfenstein 3D 11
2 Ray Tracing Example In Unreal Engine 11
3 Raycast In Unity With FMOD 12
4 Scene 1. Raycasting Works Effectively 13
5 Scene 2. Raycasting Is Too Stong And Does Not Account For Diffraction 13
6 Project Triton Rectangular Decomposition 16
7 OpenPL Structure 19
8 Accurate Voxelising 23
9 Code Extracting Geometry From Unity 24
10 Scene 1 Geometry In OpenPL 25
11 Scene 1 Geometry In Unity 25
12 Scene 2 Geometry In OpenPL 26
13 Scene 2 Geometry In Unity 27
14 C++ Code Constructing The Final Impulse Response File 28
15 Sending The OpenPL Impulse Response To An FMOD Convolution Reverb 30
16 Code Extracting Geometry From Unreal 32
17 First FDTD Implementation 34
18 First FDTD Implementation With Reflecting Boundaries 35
19 First Planeverb Simulation 36
20 First Planeverb Simulation Applied To A Unity Scene 37
21 First Planeverb Simulation Unity Scene 38
22 First Planeverb Simulation Applied To The Second Unity Scene . 39
23 First Planeverb Simulation Second Unity Scene 39
24 Simulation Output Of Close Listener And Emitter With No Geometry 40
25 Failed Simulation Output 41
26 Simulation Result On A 100x100x100 Meter Scene 42

Terminology

OpenPL	"Open Propagation Library". This library is the dissertation's artefact
Wwise	Audio middleware software used in games
FMOD	Audio middleware software used in games
Unity	Game engine
Unreal Engine	Game engine
Wave Simulation	Refers to the simulation of sound/acoustic waves
Project Triton	A Microsoft research project that focuses on Wave Simulation
Projet Acoustics	The commercial release of Project Triton
FDTD	Finite-Difference Time-Domain. A simulation technique for electronics and sound
libigl	C++ geometry processing library
Eigen	C++ maths library for algebra
OpenGL	Cross-platform rendering programming interface
Glad	OpenGL loader
GLFW	OpenGL window creator and context manager
GMP	An arithmetic library that allows for arbitrary precision
MPFR	An arithmetic library that allows for arbitrary precision with floats
Boost	Peer-reviewed C++ libraries
CGAL	Geometry algorithms
MatPlot++	C++ wrapper of the Matlab library "Matplot". Allows for scientific graph rendering

Acknowledgements

Firstly, I thank Stephen Pearse, my dissertation supervisor, for his expert guidance and without whom, this dissertation would have surely failed.

Secondly, I thank my parents for their loving support, for this dissertation would have never started without them.

Thirdly, I thank Nikunj Raghuvanshi, one of the researchers and creators of Project Triton, and Matthew Rosen, creator of Planeverb, both of whom helped answer difficult questions during development.

Introduction

The video game industry is fiscally larger than both the film and music industry combined [Milijic, 2021]; with over two billion gamers globally, it is clear that video games are an integral part of today's world. Starting in the 1950s [Brookhaven National Laboratory, 2013], video games have evolved from simple interactive entertainment into culturally defining works of art. Through this evolution, game technology has advanced in leaps and bounds.

One of the prevailing advancements in recent history is Ray Tracing [Jones, 2020, Ashworth, 2019, Martindale and Roach, 2021, Judd, 2020]. Ray Tracing is a rendering technique that traces rays from each pixel to test its interactions with the virtual world. These rays emulate the effects of real-world light phenomena like refraction and reflections. Ray Tracing is popular in non-real-time environments where Ray Tracing's expense is not a concern like film and television [Ashworth, 2019]. However, since around 2018, some games have adopted the technique to heighten visual fidelity [Burnes, 2018].

For audio, advancements are not so loudly announced. However, many game audio professionals are familiar with the middleware tools Wwise and FMOD [1]. Both advertise themselves as solutions for interactive audio. Wwise's tagline is "The Engine Powering Interactive Audio" [Audiokinetic, 2021a] and FMOD's is "Made for games - FMOD is the solution for adaptive audio" [Firelight Technologies, 2021].

While both advertise as interactive audio solutions, tools are emerging that focus on spatial audio - a term this dissertation will refer to as 'recreating real-world audio effects and phenomena in a game'. For example, Wwise is promoting its new suite of features called "Wwise Spatial Audio", creating audio that is affected by the game world [Audiokinetic, 2021b].

On top of middleware tools like Wwise pushing spatial audio, other tools are emerging that further promote the development of spatial audio technology. Tools like Steam Audio, Google Resonance and Oculus Audio.

Like Ray Tracing using rays, many tools and technologies like the ones listed above use rays to query information about the game world [2]. Rays are relatively cheap and quick to use and have been part of games dating all the way back to DOOM and Wolfenstein 3D [Sanglard, 2018]. However, rays hold significant pitfalls when used to simulate audio [Siltanen et al., 2010].

Rays are effective when used for rendering due to their similarities to real-world light rays. However, audio propagates more similarly to a 'wave' [Kuttruff, 2000]. Because of this difference, audio rays struggle to capture diffraction - the effect all of us hear in daily life like sounds travelling around objects.

[1] Middleware refers to cross-engine tools that allow for powerful audio behaviour that is not normally available in a game engine.

[2] It is hard to determine the exact technology behind closed source projects but many tools advertise their use of rays, or similar geometric approaches, to calculate audio effects.

However, the technology exists which captures all audio effects like diffraction. This technology is wave simulations, also known as numerical audio simulations. For decades, the finite-difference time-domain (FDTD)[3] method has been employed by people to simulate electromagnetics and sound. But the FDTD method requires long simulation times and large amounts of computer resources.

Likely for the exact reasons above, the FDTD method has not been used in a video game. However, in the last decade, researchers have managed to run numerical simulations for video game environments [Raghuvanshi et al., 2017].

First to emerge was Microsoft Research's Project Triton. Around ten years ago, Project Triton managed to run a numerical simulation over interactive environments with limited runtime cost [Raghuvanshi et al., 2009]. This was achieved by running the simulation 'offline', meaning the simulation is done by the developer before shipping to the player/user. The saved output of the simulation is read at runtime, making the cost of Project Triton roughly the cost of reading information from the computer's storage.

Project Triton has since evolved into Project Acoustics, a commercially available tool implemented in a small number of games as of this date [Microsoft, 2021].

However, because of the long simulation times that require completion before deployment of the game, dynamic geometry is currently not possible. But 2020's Planeverb proved numerical simulations are possible when limiting the simulation to two dimensions, and running the simulation at 10 frames per second [Rosen et al., 2020].

[3]FDTD can be roughly thought of as a numerical simulation running over a discretised grid or lattice at multiple time steps.

History Of Video Games

Date	Event	Notes
1952	First video game	A.S. Douglas creates “OXO”, an electronic version of Tic-Tac-Toe
1958	First entertainment video game	Physicist William Higinbotham creates “Tennis For Two”. This creation marks the first electronic game made for entertainment purposes
1961	Games expand to multiple computers	“Spacewar!” was the first game shared to multiple computers, allowing more people to play the game
1971/1972	The first use of audio	“Computer Space” and “Pong” become the first games to use audio
1972	First-generation consoles	The first consoles were very primitive by today’s standards, and many consoles did not have sound
1976	Second-generation consoles	Consoles could start playing synthesised sounds
1981	3D graphics	“Monster Maze” was the first game to use 3D graphics
1982	US gaming market crashes	In 1982, the US market faced a massive crash due to over saturation and low-quality games
1982	Compact Disc (CD) released	With CDs, games could fit more data and deliver more content to the player
1983	Third-generation consoles	The release of the NES by Nintendo marks the onset of third-generation consoles. Consoles still use synthesised sounds, but the NES could play 8-bit audio
1987	Fourth-generation consoles	Fourth-generation consoles mark the “16-bit era”. Consoles can play 16-bit audio
1993	Fifth-generation consoles	Fifth-generation consoles mark the “32-bit era”. The Playstation is released and can play 24 channels of 16-bit 44.1kHz ADPCM samples. The Playstation also has audio effects built-in. Consoles begin using CD storage over cartridges
1995	FMOD released	FMOD is an audio middleware tool aimed to help integrate audio into games. With hardware able to play more audio than ever, there is more of a market for tools like FMOD

1998	Sixth-generation consoles	Games start using the internet. Consoles like the Xbox support multiplayer experiences. Some consoles have support for 5.1 surround sound
2000	Wwise released	Wwise is a similar product to FMOD and saw wide market adoption in years to follow. It is currently the industry standard for game audio
2005	Seventh-generation consoles	Consoles can now play 48kHz 16-bit audio. For reference, CD-quality audio is 44.1kHz 16-bit
2012	Eighth-generation consoles	Further improvements to quality. Some consoles support 7.1 surround sound
2016	Console 4K support	Microsoft and Sony release 4K support for their consoles. However, some of this support is limited to videos, not games
2018	First Ray Tracing	Battlefield V becomes the first game to release with ray-tracing
2020	Microsoft and Sony release their next-generation consoles	Microsoft and Sony promise support for super-fast loading with SSDs, 4K output and spatial audio support

Video games started as interesting experiments for academics receiving leftover technology from World War II. In 1952, A.S. Douglas made “OXO” for his doctoral dissertation, in 1958, Higinbotham created “Tennis For Two” for a welcome day, and in 1961, MIT students created “Spacewar!”.

However, games quickly became commercial products that would venture further than just university campuses. With the advent of personal computers, consoles and arcade machines, games became more accessible to the general public.

In its roughly seventy-year history, video games have gone from simple pixels on a screen to photo-realistic experiences enjoyed by many. Along the way, companies and teams have created many new and remarkable technologies to enhance a game’s experience.

Many consumers know about Ray Tracing, which first released with Battlefield V in 2018. But not many know about Project Acoustics that released in 2015 with AltspaceVR. Regardless of the exact dates, the recentness of both technologies was a major inspiration for OpenPL. Due to how new the technology was, and how fast hardware was improving, it was felt numerical simulations would be an important topic to research.

Ray Casting Clarified

Ray Casting	Shooting a ray to test its interactions with the world
First Generation Ray Casting	Shooting a ray to turn the 2D level into a 3D projection (Wolfenstein 3D, DOOM)
Ray Tracing	Tracing rays to render a 3D scene, similar to the propagation of real-world light rays

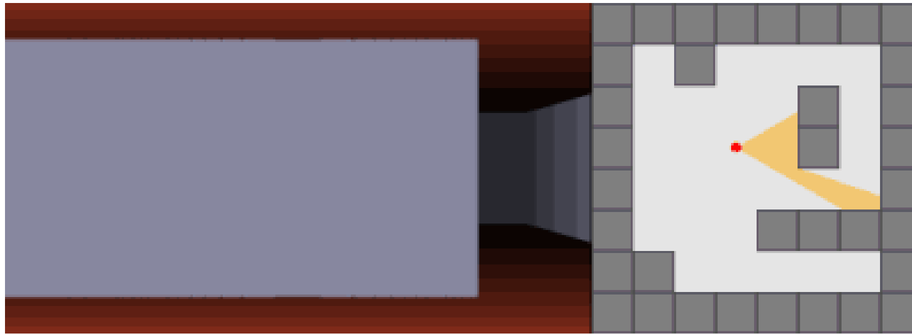

Figure 1: First Generation Ray Casting Example Similar To Wolfenstein 3D

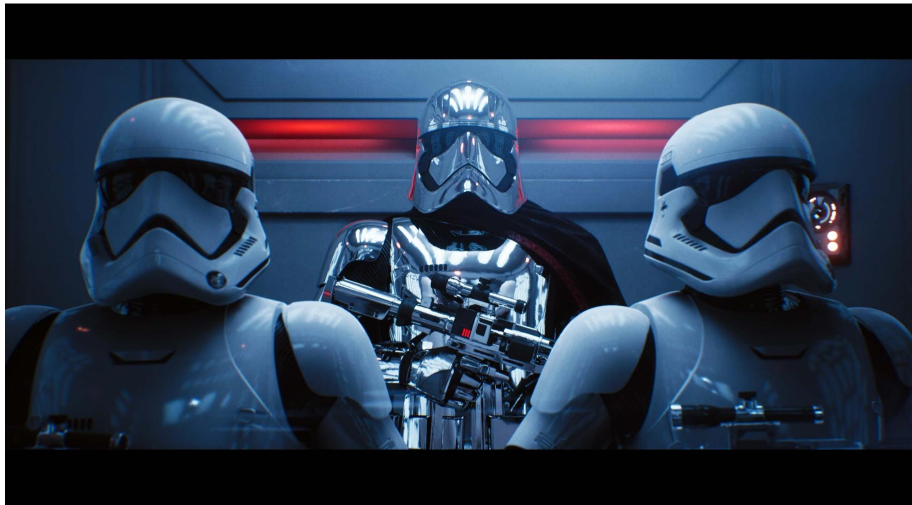

Figure 2: Ray Tracing Example In Unreal Engine

This dissertation defines ray casting as the act of shooting a ray or beam and testing its interactions with the virtual environment. Games utilise rays in many

different areas like physics, rendering and audio. Due to its prevalence, game engines make rays easy to create and, more importantly, cheap to use.

Ray casting previously defined the technology behind Wolfenstein 3D and DOOM. This dissertation will refer to this type of ray casting as "First Generation Ray Casting". In both examples, the game would shoot rays into a 2D map; the information gathered from the rays would inform the 3D renderer about the location of walls and their projection.

Ray Tracing defines shooting many rays - one for each pixel - with the results of the ray producing the final image. These rays can bounce multiple times to simulate lighting, reflections and more.

Ray Casting For Audio

```
public class RaycastingExample : MonoBehaviour
{
    public Transform listener;

    public FMODUnity.StudioEventEmitter eventEmitter;

    [Range(0,1)]
    public float occlusionStrength;

    IEnumerator Start()
    {
        yield return null;

        while (eventEmitter && eventEmitter.IsPlaying())
        {
            Vector3 listenerLocation = listener.position;
            Vector3 emitterLocation = eventEmitter.transform.position;
            Vector3 direction = (listenerLocation - emitterLocation).normalized;

            if (Physics.Raycast(emitterLocation, direction, 40f))
            {
                eventEmitter.SetParameter("Occlusion", occlusionStrength);
            }
            else
            {
                eventEmitter.SetParameter("Occlusion", 0);
            }

            yield return new WaitForSeconds(0.1f);
        }
    }
}
```

Figure 3: Raycast In Unity With FMOD

Shown above is a screenshot of the code used in the artefact to produce a quick working example of occlusion with a single raycast. Listening to the video titled "2_RaycastVersusOpenPL.mp4" in the appendix will provide the reader with an understanding of the results of raycasting.

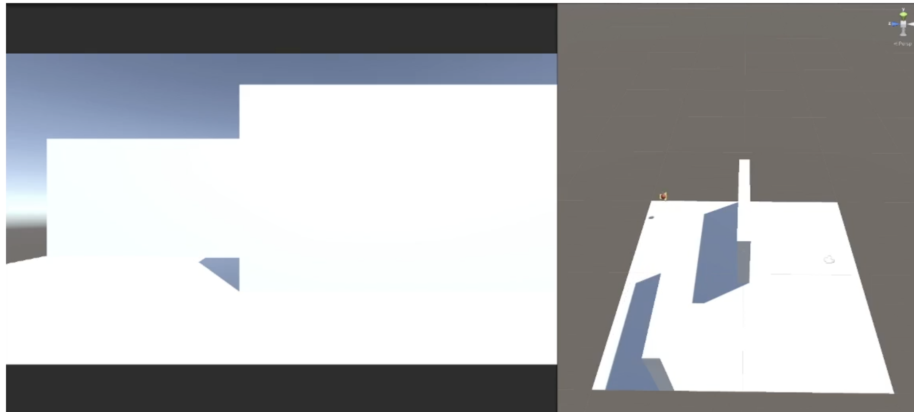

Figure 4: Scene 1. Raycasting Works Effectively

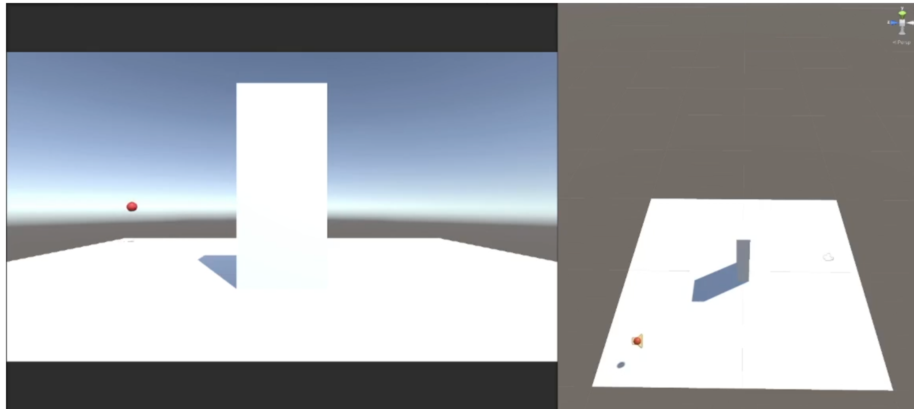

Figure 5: Scene 2. Raycasting Is Too Stong And Does Not Account For Diffraction

Two conclusions can be drawn from the video: raycasting is similarly effective to numerical options when the occluding object is completely blocking line of sight but fails to sound correct when the object is small and only partially blocking the path between sound and listener. A single raycast will create these effects because it does not account for the wave-like nature of sound.

However, the example above is primitive and tools that use rays employ different methods to hide the problems demonstrated by the video. For example, the game Nier:Automata uses multiple rays [Kohata, 2018], fired in random directions, for its reverberation calculations. Rays are successful in this instance because a single blocked ray does not drive the entire simulation; only the combination of all the rays informs the final audio values, thereby limiting the number of 'mistakes'.

Also important to mention is that the raycasting example from above took roughly thirty minutes to complete - including recording the video. On the other hand, OpenPL took months to develop, months to research and needed help from the creators of Project Acoustics and Planeverb to complete. When comparing the effort involved between a simple raycast solution, even with workarounds, and implementing a numerical simulation, the difference is staggering.

Wave Simulation Explained

Wave Simulation is an all-encompassing term that can refer to many types of acoustic simulation. However, for this dissertation, Wave Simulation will be defined as any numerical simulation. The other type of simulation is a geometric simulation that employs techniques like ray casting.

Types of geometric simulations include Ray Tracing [Cao et al., 2016,Schissler and Manocha, 2016] (not the rendering technique), Beam Tracing [Funkhouser et al., 2004] and Image Source Method [Siltanen et al., 2010]. Examples of numerical methods are Finite-Difference Time-Domain (FDTD), Finite-Element Method (FEM) and Digital Waveguide Mesh (DWM) [Siltanen et al., 2010, Raghuvanshi et al., 2009].

As explained and demonstrated previously, geometric approaches struggle to accurately simulate diffraction and require workarounds - like using multiple rays - while numerical approaches correctly account for diffraction and all audio phenomena.

As briefly touched upon, numerical methods require large amounts of computational resources. The approach taken by Project Acoustics/Triton still takes 75 minutes to simulate over a living room scene no larger than 10 meters in diameter, with larger scenes taking upwards of 350 minutes. The memory requirements are also in the 100s of MBs.

$$\frac{\partial^2 p}{\partial^2 t} - c^2 \nabla^2 p = F(x,t)$$

p	pressure
t	time
c	speed of light
$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}$	Laplacian in 3D

[Raghuvanshi et al., 2009]

The formula above accurately captures the nature of sound, as observed in the real world. Both the FDTD method and the one employed by Project Acoustics uses this formula as the basis for their equations. Reading [Raghuvanshi et al., 2009] and [Schneider, 2010] will give the reader a better understanding of both techniques than could be explained in this dissertation.

Literature Review

Research into audio simulation is extensive but is limited regarding its application within games. Project Triton stands as the first example of a numerical simulation for video games [Raghuvanshi et al., 2010]. Project Triton started with the 2009 paper titled “Efficient and accurate sound propagation using adaptive rectangular decomposition” [Raghuvanshi et al., 2009]. The paper provides a way of increasing acoustic simulation accuracy while decreasing the CPU cost and RAM usage. This feat is achieved by exploiting the solution to the acoustic wave equation on rectangular domains. In principle, the wave simulation runs over large rectangular volumes of space rather than simulating over many small voxels.

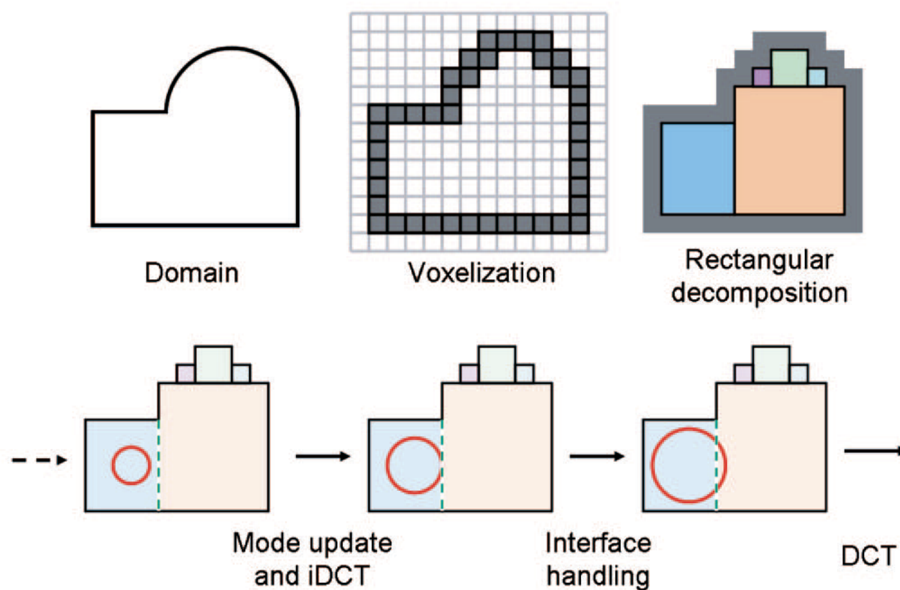

Figure 6: Project Triton Rectangular Decomposition

However, until 2010’s paper “Precomputed wave simulation for real-time sound propagation of dynamic sources in complex scenes” [Raghuvanshi et al., 2010], the research was not ready for interactive game environments. This paper separates the early reflections and late reflections of the simulated impulse response into time domain and frequency domain, and renders this binaurally at runtime for an interactive listener and source position.

Further research has expanded on these papers. 2013’s “Wave-based sound propagation in large open scenes using an equivalent source formulation” [Mehra et al., 2013] improves performance in large open scenes. 2014’s “Parametric

wave field coding for precomputed sound propagation" [Raghuvanshi and Snyder, 2014] greatly improves upon the 2010 paper by extracting salient parameters from the impulse response. These parameters are efficient to store and only take 100μs per source to decode at runtime. 2018's "Parametric directional coding for precomputed sound propagation" [Raghuvanshi and Snyder, 2018] adds support for directional sound.

However, Project Triton suffers from real-time support and cannot handle moving geometry. Matthew Rosen solved this problem in 2020 with his paper "Interactive sound propagation for dynamic scenes using 2D wave simulation" [Rosen et al., 2020]. While not strictly an advancement of Project Triton's simulation using rectangular spaces, this technique can handle moving geometry. This is solved using the FDTD method. The FDTD method is generally considered slow and was even Project Triton's main comparison for speed [Raghuvanshi et al., 2009], yet FDTD can work comparatively quickly in only two dimensions [Rosen et al., 2020]. By running the simulation ten times a second, the technique can produce a convincing audio simulation.

Ten frames a second is a relatively slow simulation speed. [Rosen et al., 2020] proves it is fast enough to create convincing audio effects but is slow to what is possible. [Allen and Raghuvanshi, 2015] shows that a 2D simulation ran on the GPU can achieve speeds of 128,000Hz.

FDTD is a prevalent acoustic simulation technique with a history going back to the 1960s. The main breakthrough comes from Kane Shee-Gong Yee's work to create the "Yee Grid". The Yee Grid staggers the electrical and magnetic fields, allowing Maxwell's equations to run more accurately.

James Maxwell published his equations in 1865. These equations took Ampere's and Faraday's Laws and turned them into working equations. The two laws were conceived in the early 1800s and only finalised by 1831.

Methodology

OpenPL is a dynamic library built using JUCE with integrations in Wwise, FMOD, Unreal and Unity. As demonstrated in earlier sections, raycasts, and geometric approaches, require workarounds and are less effective than numerical alternatives. However, they do have their benefits, namely in speed and usability. But [Rosen et al., 2020] and [Allen and Raghuvanshi, 2015] proved speed can be overcome. The final hurdle is usability.

Project Acoustics requires Microsoft Azure, a paid tool, and Planeverb only supports the Windows operating system. For these reasons, both tools are limited in their usability. On the other hand, OpenPL's simulation is ran on the user's machine. This choice makes the tool more accessible to independent developers who may not have the budget for a cloud service. Finally, the use of JUCE and its cross-platform capabilities allow developers on any operating system to use the tool.

OpenPL's premade integrations with Unreal Engine, Unity, FMOD and Wwise also help the user by decreasing the development cost of OpenPL. This ease of integration makes the tool more accessible to beginners and intermediates, who may struggle to call the library's methods correctly.

Code Structure

OpenPL uses a C Style API structure to avoid name mangling present in C++ libraries. Name mangling is avoided to ensure its compatibility with other programming languages, such as C++ for Unreal Engine 4 and C# for Unity. As libraries written in C do not mangle their function names, all library methods must have a unique name. Therefore, all OpenPL methods begin with the prefix "PL".

To handle memory management, the library exposes a PL_SYSTEM object. Using this SYSTEM object, the library user needs only to manage the PL_SYSTEM object, and not worry about freeing other parts of memory. The library exposes PL_System_Create and PL_System_Release to handle creating and destroying PL_SYSTEM objects.

The library exposes a second object to handle geometry and the simulation: PL_SCENE. The creation method requires a pointer to a PL_SYSTEM object, thereby relinquishing memory responsibility to the system. The API has many methods for adding geometry to the PL_SCENE, adding source and listener locations, simulating the geometry, and extracting information from the simulation.

All library functions return a PL_RESULT. A PL_RESULT is an enum that defines the return state of a method. Using the enum, a user can query if a method was successful or whether an error had occurred. Using an enum is a

more straightforward way of error handling, as the user need not worry about exceptions.

The library also allows the user to receive debug messages from the library. Using the PL_Debug_Initialize method, a user can send OpenPL messages to the engine's logging system.

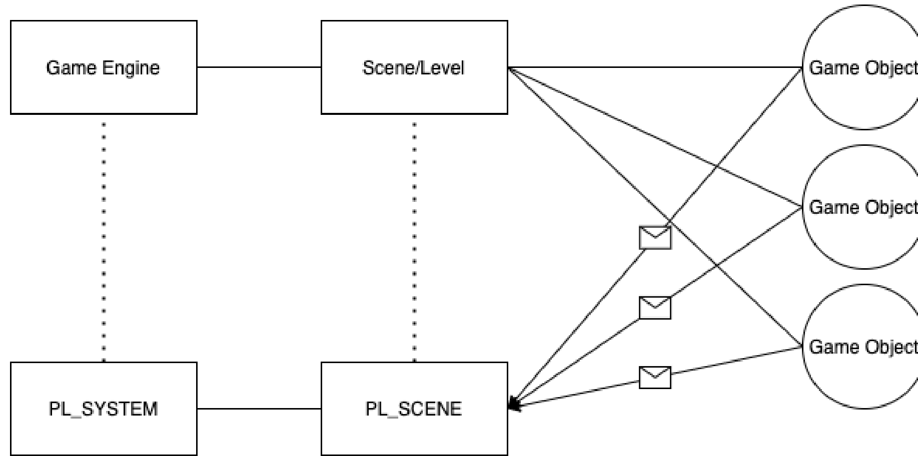

Figure 7: OpenPL Structure

OpenPL objects are designed to have similarities to their game engine counterparts. For example, a PL_SYSTEM object is similar to a game engine - the top-level manager of all objects - while a PL_SCENE object is similar to a scene or level. This design is reminiscent of similar libraries that may call objects Game Objects.

C++ And C#Wrapper

To make using OpenPL easier, a C++ and C# wrapper is included. While using the raw C API is possible, a wrapper in the user's native language is easier for development.

External Libraries And Tools

While writing all code by hand is theoretically possible, it is nearly impossible in practice. To speed up development, the library will use other libraries and tools.

For creating the final dynamic libraries for all platforms, OpenPL uses JUCE, a cross-platform audio framework powering the popular tool Max MSP.

The libraries Eigen, libigl, OpenGL and others will allow the final product to process and manipulate geometry, render results to the user and carry out

complex mathematical equations.

Project Management

Popular software development methodologies include Agile and Scrum. Some tools that support these development styles are Asana and Jira; another tool popular in indie game development scenes is HacknPlan. These tools are excellent for large teams with a consistent number of hours worked each week.

However, there is an overhead to using these tools. The tools assume users are working in a team; therefore, there are many features to keep communication open between members. However, when working alone, these features are not always needed.

For this dissertation, project management will be laxer and rely on more basic methods like to-do lists. The note-taking app Obsidian will track the completion of tasks and any extra information about the project.

Version Control

Tracking changes throughout the development of the dissertation is paramount. A common use of version control is to recover previously deleted work or quickly change between versions.

For software, this dissertation uses Git, Git Tower and GitHub. Git Tower and GitHub offer free pro licences for university students that add extra features over free accounts, making them clear choices. While other version control systems exist, Git was the preferred choice due to its support from GitHub.

Moreover, GitHub has the 'GitHub Actions' feature which allows a server to test and run code. Using these tests, a developer can find bugs faster and across multiple platforms. For this dissertation, GitHub Actions tests the main simulation code and builds dynamic libraries for all three operating systems.

Implementation

Initially, OpenPL was to research Project Acoustics and create a similar tool. However, the maths became too difficult and the goals of the project changed. The new goals became: create a tool to a professional standard, release the tool for the three major operating systems and integrate it with Wwise, FMOD, Unreal Engine and Unity.

Developing On Different Platforms

To develop for each operating system, the target operating system is required. Windows releases free virtual machines with limited-time licences installed. Using Virtual Box, a virtualisation application, developing for Windows was possible. Lastly, as Ubuntu releases its operating system for free, using Virtual Box again allowed for development on Linux.

Setup Script

To make it easy for users to install OpenPL, a setup script was created. On Mac, this script downloads JUCE, libigl, its dependencies (glad, Eigen and GLFW), GMP, MPFR, boost, CGAL and MatPlot++. Most installing is handled with git and the Homebrew package manager.

On Linux, the installation process is similar to Mac with two differences. First, additional libraries are required for JUCE to compile correctly. These number around twenty additional libraries. However, some are installed as over precaution. The second difference is that MatPlot++ cannot be installed through a package manager on Ubuntu (Arch Linux is the only distribution with package manager support). Therefore, MatPlot++ is installed with git and compiled with CMAKE.

On Windows, compilation errors were encountered, stopping the development of the setup script. These errors were found to be macros placed before function declarations. To solve this, all macros were placed after the function's return type.

'JUCE_PUBLIC_FUNCTION void FunctionName();' became 'void JUCE_PUBLIC_FUNCTION FunctionName()'.

Once fixing the macros, the vcpkg was used to install dependencies. However, installation can be slow as vcpkg installs packages from their source code. To speed this up slightly, the GitHub actions' "cache" tool was used to cache vcpkg between builds, thereby reducing the build time to around ten minutes, similar to Linux and Mac times.

GitHub Actions Automation

Continuous integration (CI) is a common approach employed by software devel-

opers. Using CI, developers can test whether their code compiles on different platforms and compile final packages for different platforms without developer intervention.

GitHub provides their “Actions” tool to support CI. The script for OpenPL is relatively simple. Past handling when the CI runs and for which platforms, the script runs the setup script for the correct platform and then invokes either xcodebuild, make or MSBuild.

This CI script is relatively simple, thanks to including the IDE files in the GitHub repository. Without this, JUCE’s Projucer app would have to be compiled from source and used to build the IDE projects. This extra step would significantly increase the build times and complexity of the project.

Building A Single DLL

Building for Windows resulted in many DLL files. Therefore, when importing into Unity, all of these extra files were required. This is a problem when shipping because sending multiple files is harder and adds more room for errors and missing files.

To try and solve these file problems, the Windows build was set to Static Runtime. It was assumed all DLLs would get bundled together but that was not the case. After much development and experimentation, OpenPL was left to build multiple DLL files.

Afterwards, it was found that Project Acoustics also builds multiple DLLs, evidenced by its Unity integration. With multiple DLL files not seen as a major problem, the GitHub Actions script was updated to bundle all DLL files in a final zip file.

Physics And Voxelising The Scene

From early research, the libigl library was shown to be a great library for geometry processing. Handily, libgil and its dependency Eigen allows for Axis Aligned Bounding Boxes (AABB). These are used frequently in games to test if two objects are colliding due to their efficiency.

The artefact used AABBs at first for the reasons above. However, they have a significant flaw: they cannot accurately handle rotated objects and complex shapes. Keeping the name “box” in mind, one can imagine a box being drawn around the extents of an object. When the object is a simple cube with no rotation, the AABB matches the object incredibly well. However, rotate that object, and suddenly, the AABB does not wrap the cube as tightly.

Now imagine a long rectangular object. At no rotation, the AABB is accurate. However, turn this 45 degrees, and there is lots of empty space. This problem is compounded when using complex shapes with multiple cuts and protrusions.

Because of AABB's flaws, some voxels were getting created for parts of the scene that should clearly be empty space/air. However, due to the AABBs, the simulation thought there was geometry where there was not.

To counteract this, an open-source physics library was searched for and ReactPhysics3D was found. The library is feature-rich but quickly ran into segmentation errors and other memory problems during use.

Through opening a GitHub Issue, debugging the issue started with the library creator. However, no solution was found. Therefore, the search began again. This time, greater effort was put into searching the libigl library. Thankfully, a solution in the form of "points_inside_component" was found.

This method correctly checked if points were inside a mesh. By passing each point of the voxel and the mesh, it was easy to find what voxels were colliding with the game meshes.

Testing this in Unity provided much more accurate results. There are still some imperfections but can be better attributed to the voxel size than the approach or collision algorithm.

The screenshot below shows the more accurate physics testing being used.

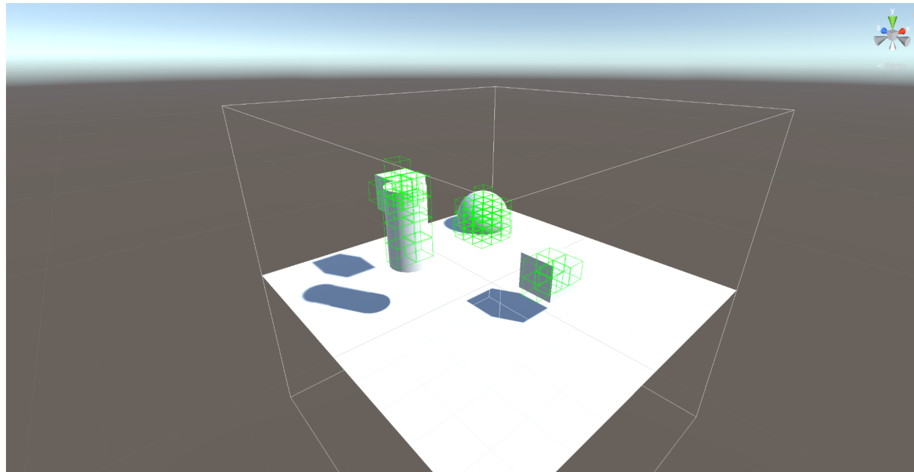

Figure 8: Accurate Voxelising

The sphere at the back of the scene is a good example of the new physics code accurately conforming to the shape of the object. If AABBs were still used, an entire "box" of voxels around the sphere would be filled.

Passing Geometry To OpenPL

Sending geometry was solved early on in development. In Unity, one can get the mesh vertices and indices. By sending the pointer of these arrays, OpenPL can read the geometry and convert the data into its internal data format, which are Eigen matrices.

```
MeshFilter mesh = currentGameObject.GetComponent<MeshFilter>();

List<Vector3> Verts = new List<Vector3>();
mesh.mesh.GetVertices(Verts);
List<int> Inds = new List<int>();
mesh.mesh.GetIndices(Inds, 0);

GCHandle VertHandle = GCHandle.Alloc(Verts.ToArray(), GCHandleType.Pinned);
GCHandle IndHandle = GCHandle.Alloc(Inds.ToArray(), GCHandleType.Pinned);

int Index = 0;

try
{
    PLVector pos = mesh.transform.position.ToPLVector();
    PLQuaternion rot = mesh.transform.rotation.ToPLQuaternion();
    PLVector scal = mesh.transform.lossyScale.ToPLVector();
    RESULT MeshResult = SceneInstance.AddMesh(ref pos, ref rot, ref scal,
VertHandle.AddrOfPinnedObject(), Verts.Count, IndHandle.AddrOfPinnedObject(), Inds.Count, out Index);
}
finally
{
    VertHandle.Free();
    IndHandle.Free();
}
```

Figure 9: Code Extracting Geometry From Unity

In the code above, it is shown that the easiest part is getting the raw vertices and indices. Once the MeshFilter object is referenced with a GetComponent call, one can access the underlying data with GetVertices and GetIndices methods.

The most complicated part of the process is pinning the arrays to retrieve a pointer that can be passed to OpenPL. OpenPL expects raw pointers to the first vertice and indice, and the length of each array. Due to C#'s design, one cannot quickly retrieve a pointer to a data structure and must write code like above to get an IntPtr type.

The screenshots below show Unity geometry successfully recreated in OpenPL. The images with a purple background are OpenPL screenshots, while the screenshots with lighting and icons are Unity.

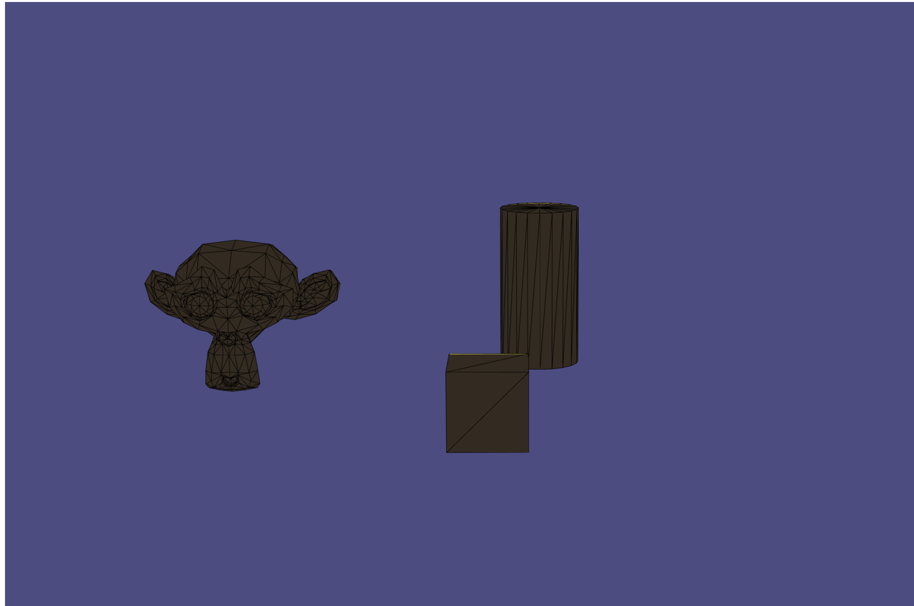

Figure 10: Scene 1 Geometry In OpenPL

Figure 11: Scene 1 Geometry In Unity

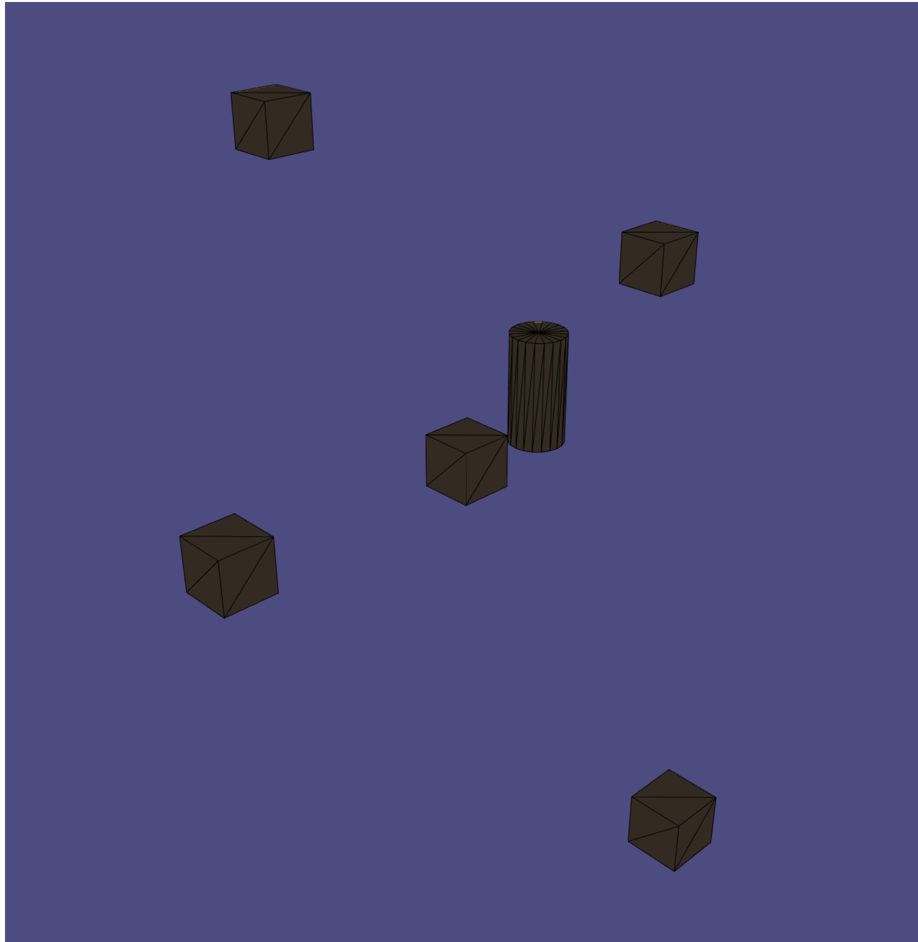

Figure 12: Scene 2 Geometry In OpenPL

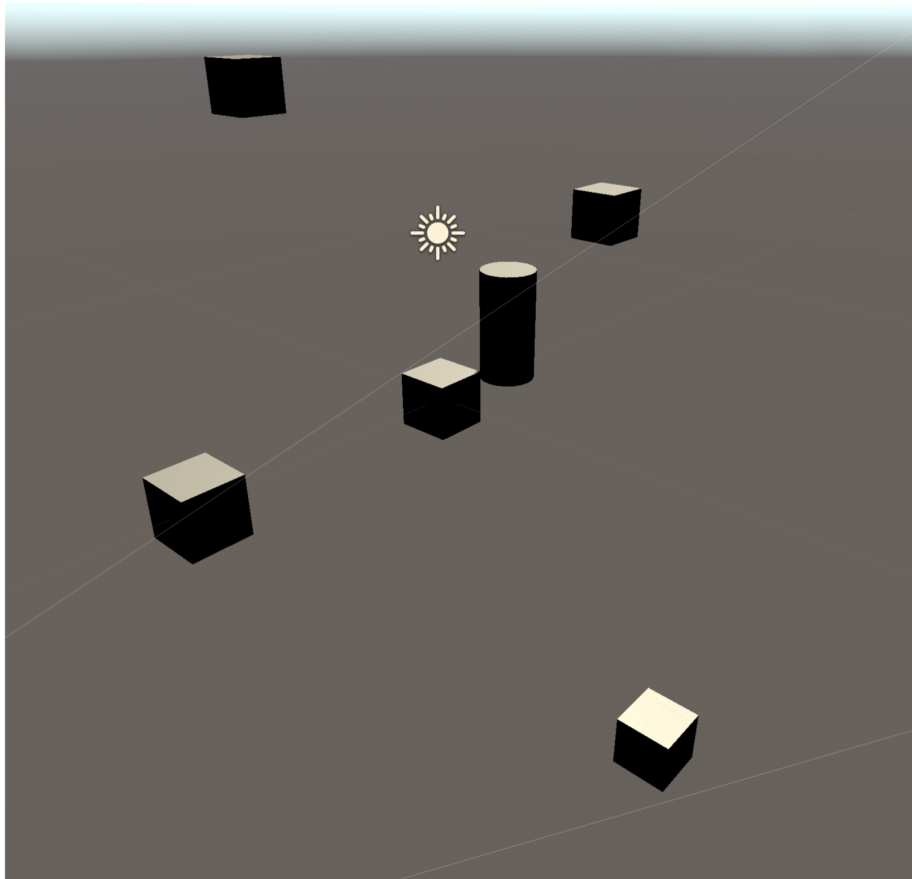

Figure 13: Scene 2 Geometry In Unity

Graph Rendering

After implementing a lot of the simulation, it became necessary to test its output. Therefore, a graph was the first consideration as broken, or failed audio files are hard to test past “it is wrong”. After searching, MatPlot++ was found, a C++ library based on MatLab’s MatPlot library.

Initial use of the tool involved testing its features and graph types. Eventually, the “waterfall” graph was found and allowed for what looked like audio or wave graph rendering. This ended up being perfect for the artefact’s uses as it allowed a user to plot the data along the entire X-axis and time axis while also plotting air pressure.

With these graphs, it became apparent when the simulation code was working or not, and these graphs were paramount to finding success.

Exporting The Impulse Response

An "Analyser" class was created to handle taking the simulated voxel data from the simulation and building a final impulse response file. Thanks to JUCE, this process is rather simple.

```
  juce::File DesktopDirectory = juce::File::getSpecialLocation(juce::File::userDesktopDirectory);
  juce::File OutputFile = DesktopDirectory.getNonexistentChildFile("TestImpulseResponse", ".wav");
  OutputFile.deleteFile();    // Delete so we can override ???

  float SamplingRate = Simulator->GetSamplingRate();

  if (std::unique_ptr<juce::FileOutputStream> FileStream = std::unique_ptr<juce::FileOutputStream>
(OutputFile.createOutputStream()))
  {
      juce::WavAudioFormat WavFormat;

      if (auto Writer = WavFormat.createWriterFor(FileStream.get(), SamplingRate, 1, 16, {}, 0))
      {
          FileStream.release(); // (passes responsibility for deleting the stream to the writer object
that is now using it)

          juce::AudioBuffer<float> AudioBuffer (1, SamplingRate);

          const std::vector<std::vector<PLVoxel>>& SimulatedLattice = Simulator->GetSimulatedLattice();

          int EncodingIndex;
          Simulator->GetScene()->GetVoxelIndexOfPosition(EncodingPosition, &EncodingIndex);

          const std::vector<PLVoxel>& Response = SimulatedLattice[EncodingIndex];

          for(int i = 0; i < Response.size(); ++i)
          {
              AudioBuffer.addSample(0, i, static_cast<float>(Response[i].AirPressure));
          }

          Writer->writeFromAudioSampleBuffer(AudioBuffer, 0, static_cast<int>(Response.size()));

          delete Writer;
      }
  }
```

Figure 14: C++ Code Constructing The Final Impulse Response File

Following JUCE's audio recording demo, a file is first created. Then, a file output stream and wave writer is created. An audio buffer filled with the air pressure from the simulation can be sent to the wave writer and saved to the file created earlier.

The final parts of the code handle getting the voxel index where the analyser was asked to make an impulse response.

Parameter Extraction

The impulse response from the simulation was ineffective when passed to middleware tools. This was primarily due to the low simulation frequency. Therefore, extracting the parameters from the impulse response like Project Acoustics and

Planeverb does was the solution.

$$DirectEnergy = \int_{\tau}^{\tau_{DS}} p^2(t)dt$$

[Rosen et al., 2020]

τ defines the start of the impulse response with $\tau_{DS} = \tau + 10ms$. The formula defines an equation that sums the squared pressure values from time 0 to 10ms, multiplied by the spatial steps (or voxel size). The output of the equation is "direct sound energy".

To create a final "obstruction gain" value that can be sent to middleware tools, two direct energy values are needed: the free energy from a simulation with no geometry and the energy from a simulation with all geometry. Both are calculated with the formula above.

With the two values, they can be plugged into the following equation to give a final gain value of g.

$$g = \sqrt{\frac{GeometryEnergy}{FreeEnergy}}$$

[Rosen et al., 2020]

$FreeEnergy$ is initially calculated by taking the energy 1 meter away from the listener. When g is calculated, $FreeEnergy$ is divided by the distance between listener and emitter.

At first, the value was used as a literal gain value that either increased or decreased the sound's volume. However, more creative control was possible by using the following formula.

$$Occlusion = 1 - g$$

Using the above, a parameter in FMOD or Wwise could be used to indicate 0 as no occlusion and 1 as full. Using effects in the middleware tool, a more creative occlusion effect can be produced.

More parameters can be extracted from the impulse response, shown in the Planeverb paper, however have not been implemented in this artifact.

FMOD Unity Integration (Impulse Responses)

Once OpenPL could build impulse responses from the simulation, it was time to use these responses in a game engine. Unity and FMOD were chosen to start as

both are friendly towards indie developers and it was assumed to be the easiest to start with.

FMOD's Unity integration is easy to use. Included by default are "StudioEventEmitter"s that allow for easy playing of FMOD events. Setting one of these up in the editor and then referencing it from the script allows easy access to all the information needed for passing impulse responses.

```
if (eventEmitter.EventInstance.getChannelGroup(out FMOD.ChannelGroup group) == FMOD.RESULT.OK)
{
    int dspNum = -1;
    group.getNumDSPs(out dspNum);

    for (int i = 0; i < dspNum; i++)
    {
        FMOD.DSP dsp;
        if (group.getDSP(i, out dsp) == FMOD.RESULT.OK)
        {
            FMOD.DSP_TYPE type;
            dsp.getType(out type);

            if (type == FMOD.DSP_TYPE.CONVOLUTIONREVERB)
            {
                int channels, sampleRate;
                float[] ir = WAV.Read(irSamplePath, out channels, out sampleRate);

                byte[] array = new byte[ir.Length + 1];
                array[0] = (byte)channels;
                Buffer.BlockCopy(ir, 0, array, 1, ir.Length);

                dsp.setParameterData((int)FMOD.DSP_CONVOLUTION_REVERB.IR, array);
            }
        }
    }
}
```

Figure 15: Sending The OpenPL Impulse Response To An FMOD Convolution Reverb

FMOD Studio Events are channel groups and groups under the hood. Therefore, one can get the master channel group from a Studio Event Instance with ease, as shown above. The rest of the code loops through all the DSPs attached to the channel group until it finds one of type "Convolution Reverb".

Once it finds the reverb attached to the event, the impulse response is loaded from disk - "irSamplePath". FMOD expects a byte array of all the data with the first byte representing the number of channels. Therefore, the byte array is created one larger than the actual impulse response length.

The first byte is filled with the channel count, then the impulse response is copied into the byte array. Once filled, the array is sent to the DSP effect with "setParameterData". "DSP_CONVOLUTION_REVERB.IR" makes sure the

array is sent to the correct parameter. The enum value represents the number "0" and could have been used in place of the enum. However, if the API changes in future versions, this will keep the code working correctly.

FMOD Unity Integration (Parameter Extraction)

The process above was the previous way integration worked when using impulse responses. However, since using occlusion values and parameters, the integration has become easier.

OpenPL defines a single method for extracting the occlusion value from a location. This value can be passed straight to FMOD through its "setParameter" function.

Wwise Unity Integration (Impulse Responses)

Currently, Wwise is not integrated due to its API not exposing functions for changing the impulse response of its convolution reverb during runtime. Because of this, a custom plugin is needed so the impulse response can change from game code.

Wwise Unity Integration (Parameter Extraction)

Implementing Wwise with OpenPL, bar the boilerplate of adding Wwise to the project and setting up an emitter, is exactly the same as it is for FMOD. However, FMOD's Studio Event Emitter method 'SetParameter' is swapped for the AkSoundEngine method 'SetObjectObstructionAndOcclusion' which takes the listener and emitter as parameters.

FMOD Unreal Integration

Unlike Unity, one cannot drag and drop a dynamic library into the project and expect the program to run. For Unreal, a natural solution is to create a plugin. Unreal works on the principles of "modules". The engine itself, as well as the game's code, are modules. Therefore, a plugin is an extra module added to the game's module. The user can define what files need to be compiled, as well as any external dynamic libraries needed.

By creating an OpenPL plugin module, the Unreal facing API, the OpenPL headers and the dynamic libraries can be defined and added to the game.

However, the first problem was the dynamic library not being found by Unreal. When using the macOS tool "otool", it was found the library file in the Unreal plugin pointed to "/usr/local/lib". As of writing, a solution has not been found. To continue development, a copy of OpenPL was placed in the expected folder. However, while development could continue, deploying to users will be a problem and require fixing.

To get the raw vertices and indices data from an Unreal mesh requires digging through different classes and data types. However, the process is still relatively simple as shown below.

```
TArray<PLVector> Vertices;
TArray<int> Indices;

AStaticMeshActor* MeshActor;
UStaticMeshComponent* StaticMeshComponent = MeshActor->GetStaticMeshComponent();
UStaticMesh* StaticMesh = StaticMeshComponent->GetStaticMesh();

const int32 LOD = 0;

if (StaticMesh->HasValidRenderData(true, LOD))
{
    const auto& RenderData = StaticMesh->GetLODForExport(LOD);
    const FPositionVertexBuffer& VertexBuffer = RenderData.VertexBuffers.PositionVertexBuffer;

    const FRawStaticIndexBuffer& IndexBuffer = RenderData.IndexBuffer;
    TArray<uint32> UnrealIndices;
    IndexBuffer.GetCopy(UnrealIndices);
    auto TriangleCount = RenderData.GetNumTriangles();
    auto VertexCount = VertexBuffer.GetNumVertices();

    for (int i = 0; i < VertexCount; i++)
    {
        const FVector& VertexPos = VertexBuffer.VertexPosition(i);
        const FVector& VertexWorld = MeshActor->GetTransform().TransformPosition(VertexPos);
        Vertices.Add(PLVector(VertexWorld.X, VertexWorld.Y, VertexWorld.Z));
    }

    for (int i = 0; i < UnrealIndices.Num(); i++)
    {
        Indices.Add(static_cast<int>(UnrealIndices[i]));
    }
}
```

Figure 16: Code Extracting Geometry From Unreal

One of the largest differences between Unity and Unreal meshes is having to retrieve the mesh for the specified LOD level.

With geometry extraction handled, the process is rather similar to Unity to trigger the simulation and retrieve occlusion values. There is a little more overhead because you cannot directly reference a ‘StudioEventEmitter‘ like in Unity or reference the player from the scene, but the process of digging through a couple class variables is not too challenging.

The final hurdle for using Unreal Engine is converting the Unreal coordinate system into a Unity/OpenPL coordinate system. This is rather simple and consists of, for example, swapping the X coordinate for the Y coordinate. Finally, Unreal’s units are in centimeters while Unity and OpenPL are in meters. The solution is to divide all of Unreal’s coordinates by one-hundred.

However, there is a final bug regarding the geometry: occlusion only works at the top and bottom of objects. This is yet to be fixed but the cause is assumed to lie in the voxelisation process.

Wwise Unreal Implementation

Integrating Wwise was a little harder than implementing FMOD. Wwise supports occlusion and obstruction by default. These settings are defined globally in the Wwise editor and its Unreal Engine implementation calls upon this functionality to update occlusion values for any object the user wants.

Following this, one could assume using the code powering this system would result in success, when swapping out certain parts for OpenPL. However, this is untrue. In Wwise's documentation, it makes note of the fact that all internal code must be exposed via the Wwise implementation and not a separate module or plugin.

Due to these problems, there were errors when creating the first implementation. The solution was to create an RTPC on the sound object and set the RTPC, ignoring Wwise's global obstruction and occlusion system. However, this is contrary to the Unity implementation where the global obstruction and occlusion system is utilised. From this, it can be concluded that implementation will vary, sometimes drastically, between different engines, middleware and platforms.

Simulation Implementation

To start learning the FDTD method, code from John B. Schneider's "Understanding the Finite-Difference Time-Domain Method" [Schneider, 2010] was used. Through reading the different chapters, a basic understanding of the method was acquired. After implementing some of the code presented in Chapter 12 "Acoustic FDTD Simulations", the artefact could run a very basic FDTD simulation over the scene.

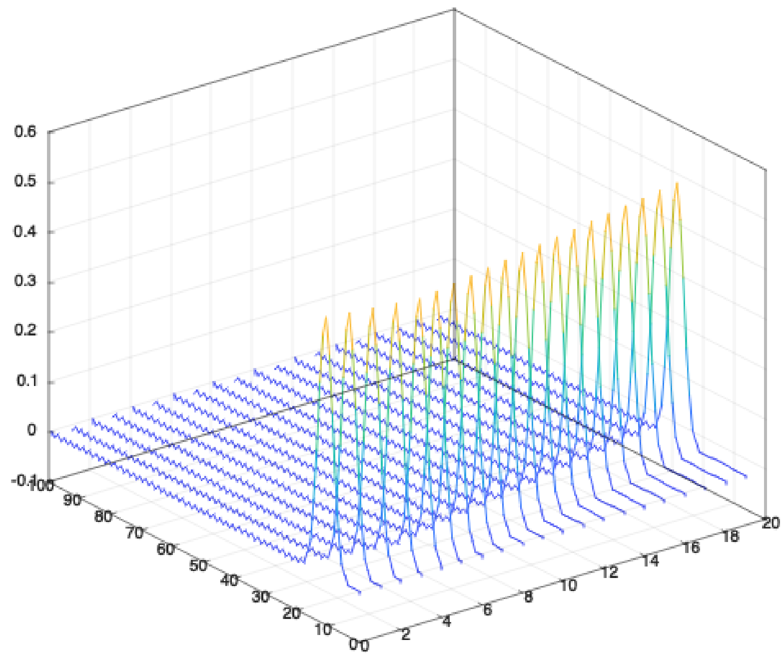

Figure 17: First FDTD Implementation

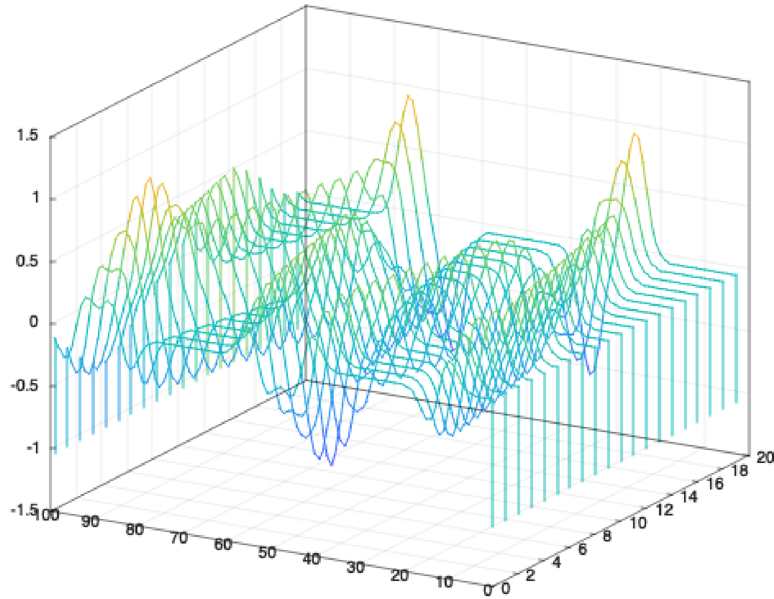

Figure 18: First FDTD Implementation With Reflecting Boundaries

The results of the first implementation are shown above. These simulations were rather primitive and weren't affected by the scene's geometry. However, getting this far was a big breakthrough.

After implementing this basic approach to understand the method, Planeverb's simulation code was implemented. This was a struggle at first due to Planeverb's code layout, but was eventually successful.

One of the first rendered graphs from using Planeverb's simulation is shown below.

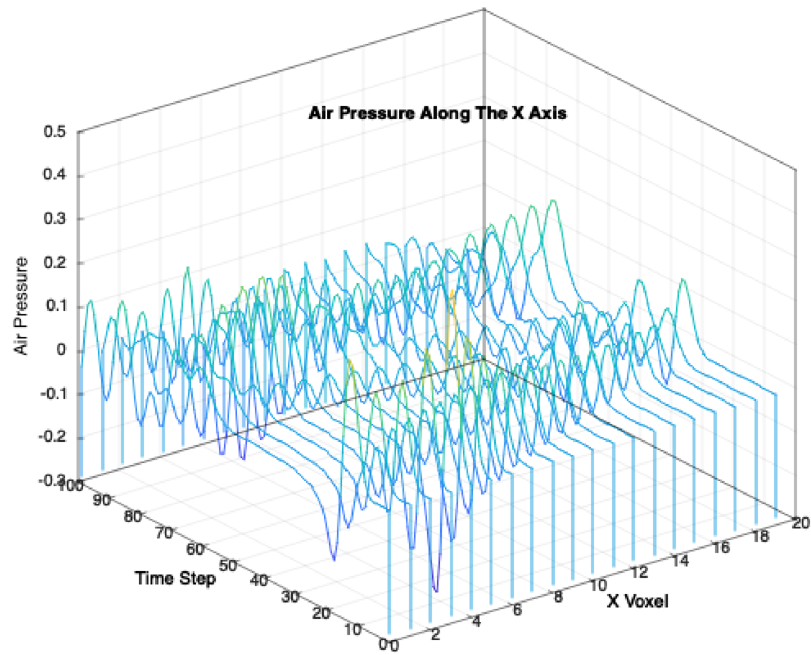

Figure 19: First Planeverb Simulation

The Planeverb code respected the geometry of the scene and was able to be affected by the geometry sent from the game. The following graph shows the simulation results at the emitter's location shown in the screenshot.

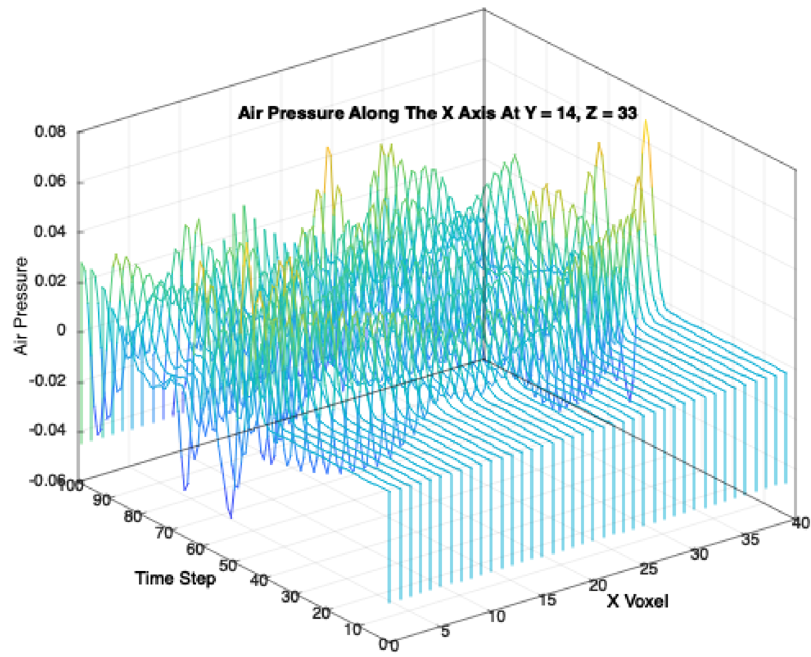

Figure 20: First Planeverb Simulation Applied To A Unity Scene

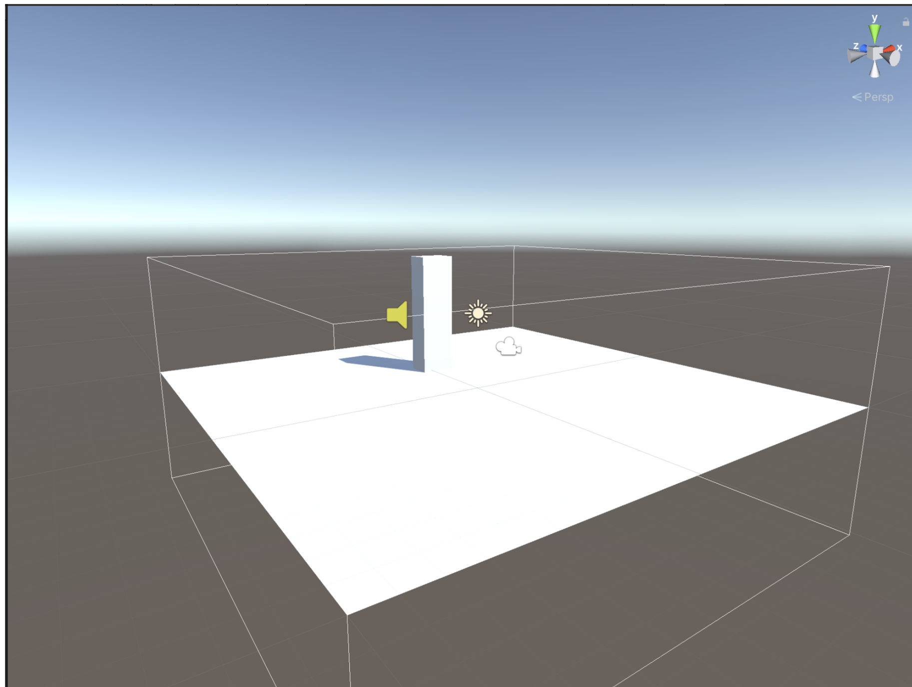

Figure 21: First Planeverb Simulation Unity Scene

The next graph and screenshot shows a larger obstacle between the listener (and the simulation start location) and the emitter.

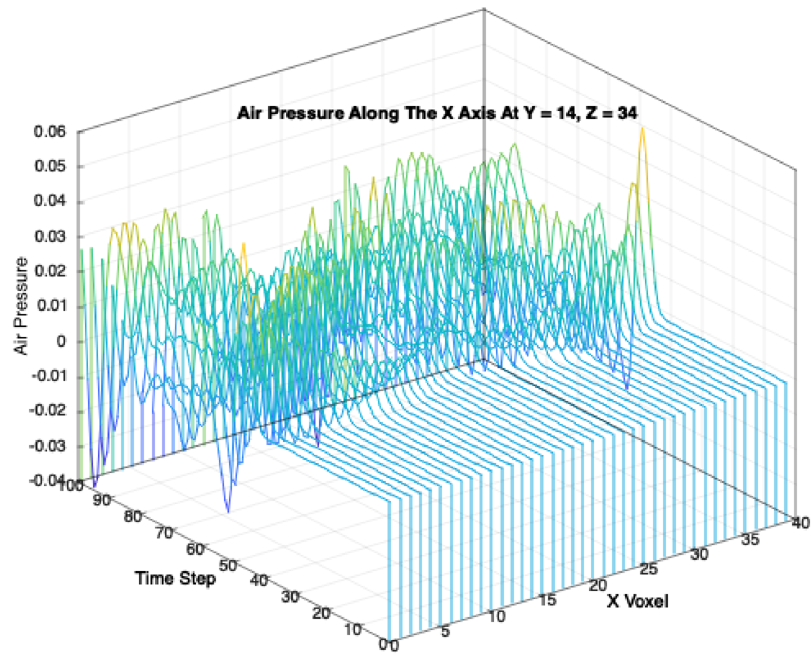

Figure 22: First Planeverb Simulation Applied To The Second Unity Scene

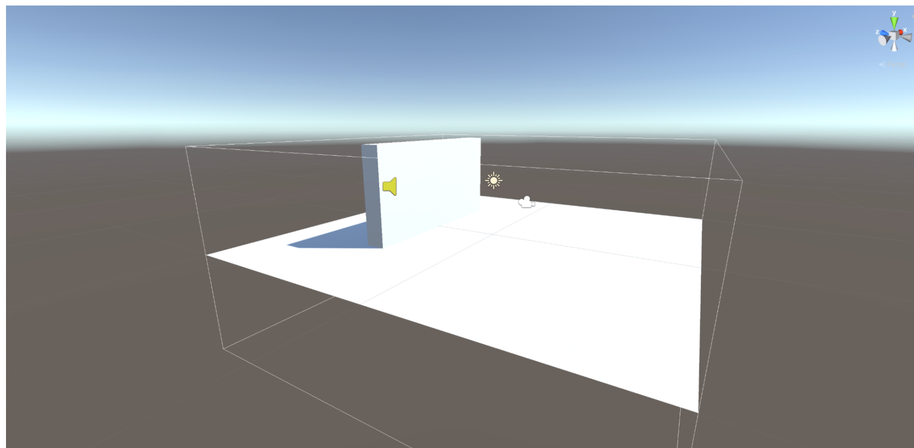

Figure 23: First Planeverb Simulation Second Unity Scene

Here is a graph showing the results of the emitter being placed closer to the

listener with no wall in between.

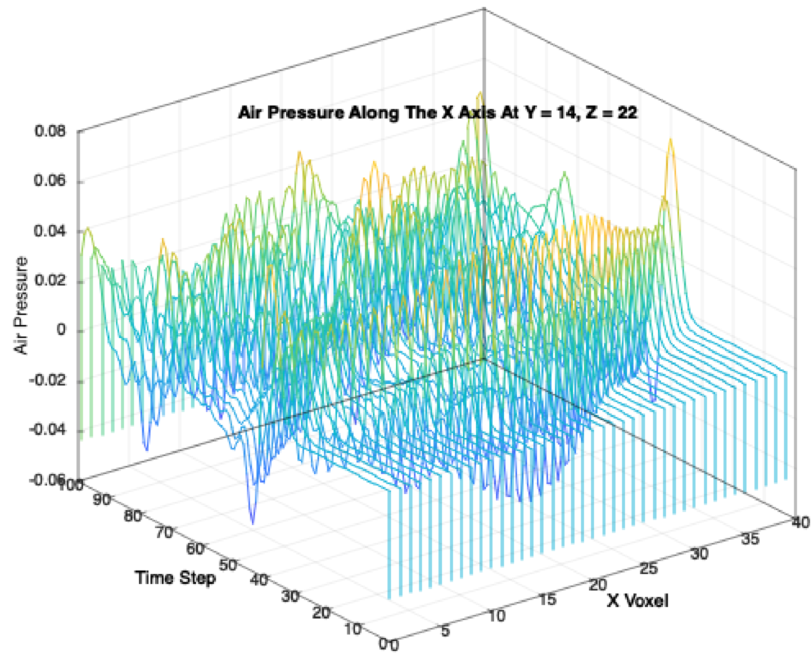

Figure 24: Simulation Output Of Close Listener And Emitter With No Geometry

Failed Simulations

However, to get to this point involved running into many bugs. For instance, this graph output shows a failed simulation. This simulation was an attempt at making the Planeverb 2D simulation work in 3D. However, not enough knowledge was known to correctly implement this.

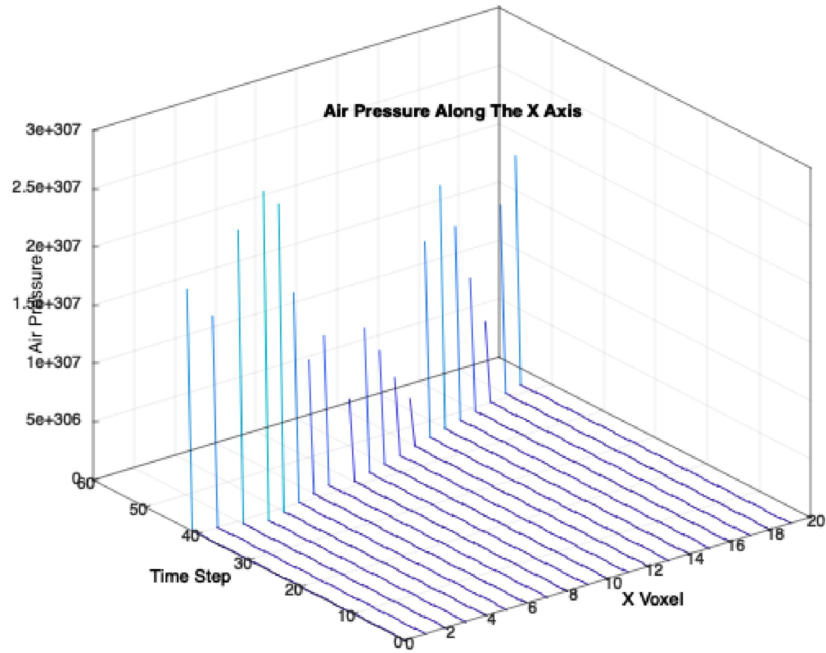

Figure 25: Failed Simulation Output

Performance And Speed

The dissertation's first implementation of Planeverb's code simulated over every voxel in the 3D lattice. This caused long simulation times as shown below.

Results are taken from a 3.2 GHz Quad-Core Intel Core i5 with 24GB of DDR3 RAM.

Simulation Size (Meters)	Voxel Size (Meters)	Number Of Voxels	Time (Seconds)
5x5x5	1	125	0.001399
10x10x10	1	1,000	0.01
20x20x20	1	8,000	0.09
40x40x40	1	64,000	0.89
100x100x100	1	1,000,000	20.95

However, Planeverb's implementation is designed to work on a 2D grid, thereby limiting the dimensions and speeding up the simulation.

Changing the code to simulate over a fixed Y-axis value results in these times.

Simulation Size (Meters)	Voxel Size (Meters)	Number Of Voxels	Time (Seconds)
5x5x5	1	125	0.000477
10x10x10	1	1,000	0.002672
20x20x20	1	8,000	0.021399
40x40x40	1	64,000	0.182111
100x100x100	1	1,000,000	2.922831

Comparing the two results, the performance benefits are clear to see; most notably at 1,000,000 voxels where there is an 18 second improvement. At smaller sizes, there are still significant performance improvements but less noticeable due to the already fast simulation speed.

The graph below shows the output of the simulation at 100 meters cubed size.

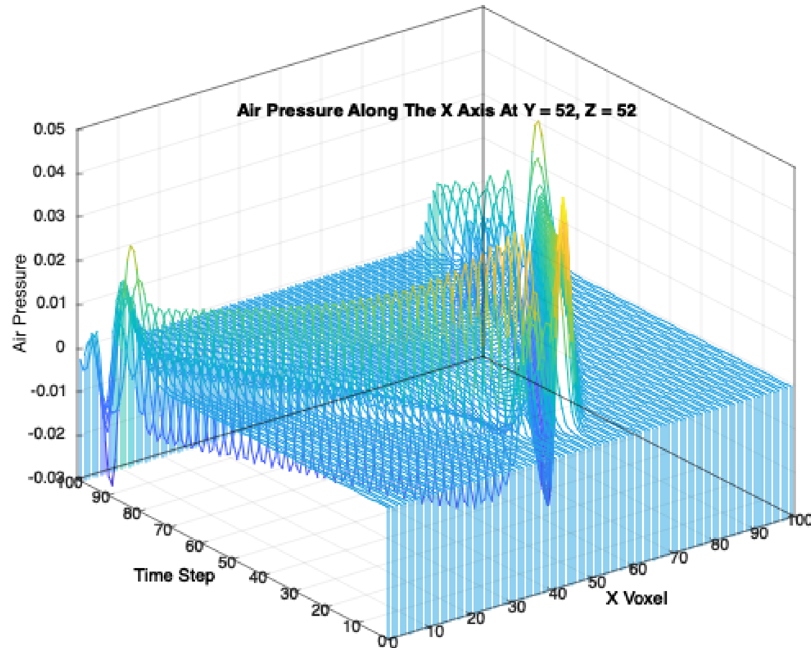

Figure 26: Simulation Result On A 100x100x100 Meter Scene

Planeverb Codebase Review

As mentioned in the literature review, Planeverb is an open-source wave simulation tool. This tool and its code is the foundation of this dissertation. Planeverb

works by running a 2D FDTD simulation over a voxel scene. Planeverb fills these voxels with AABBs informed from the game.

Planeverb takes a serious approach to speed and memory usage. Most notably, all memory is allocated at application start, and placed in a collective memory pool for its memory cache improvements.

However, these performance improvements hinder its readability and maintainability. These challenges make the tool harder to move past research domains and into production ready code that game developers can use.

Another problem in the code base is its variable naming, comments and structure. Variables are regularly named as their shorthand versions found in the mathematical equations and comments are sometimes limited - again, assuming the paper is used to understand the code. Finally, the structure is not laid out as one would expect from C++.

To explain, a method can be declared in one file, and defined somewhere else. Common practice is to place the declaration in a "<FileName>.h" and the definition in a "<FileName>.cpp". However, many functions declared in "A.h" can be found in "B.cpp" - not "A.cpp". This layout again hinders readability and maintainability.

To compare, OpenPL follows the "functions declared in A.h will be defined in A.cpp", which improves maintainability.

However, while the code is sometimes hard to read, it is highly efficient and effective in its results.

Conclusion

OpenPL set out to be an indie friendly version of Project Acoustics with cross platform support and better maintainability than Planeverb. On these points, it is partially successful. OpenPL is cross platform - but suffers from large file sizes and multiple files to juggle on Windows - and correctly integrates into both Wwise and FMOD, but does not have the same number of features as either Project Acoustics or Planeverb.

Currently, OpenPL is more a work in progress and proof of concept than a ready commercial product. However, this is possibly expected with the time frame and amount of research and learning required before development could start. Moreover, OpenPL was still successful as a learning tool and research aid into numerical simulations.

The time required to create a numerical audio simulation, compared to a geometric/raycast alternative, is quite drastic. As compared before, a simple raycast solution can be finished within hours, while OpenPL took months, if not a full year when counting research before the academic year. Because of these drastic differences, the conclusion can be made that numerical simulations are not more prevalent because of the time investment required. However, I believe Project Acoustics, Planeverb and OpenPL prove the benefits of numerical simulations and the powerful results they can yield.

While hard to predict, with Project Acoustics backed by Microsoft and seeing support in their gaming consoles, numerical simulations can be assumed to gain a larger adoption in the future. At what rate is unclear however. But with hardware improvements making the technology easier to run, there will likely be a time when numerical simulations are just as easy to run as raycasts.

References

[Allen and Raghuvanshi, 2015] Allen, A. and Raghuvanshi, N. (2015). Aerophones in Flatland: Interactive wave simulation of wind instruments. In *ACM Transactions on Graphics*, volume 34. Association for Computing Machinery.

[Ashworth, 2019] Ashworth, B. (2019). What Is Ray Tracing? The Latest Gaming Buzzword Explained | WIRED.

[Audiokinetic, 2021a] Audiokinetic (2021a). Wwise | Audiokinetic.

[Audiokinetic, 2021b] Audiokinetic (2021b). Wwise Spatial Audio | Audiokinetic.

[Brookhaven National Laboratory, 2013] Brookhaven National Laboratory (2013). The First Video Game.

[Burnes, 2018] Burnes, A. (2018). Battlefield V DXR Real-Time Ray Tracing Available Now.

[Cao et al., 2016] Cao, C., Ren, Z., Schissler, C., Manocha, D., and Zhou, K. (2016). Interactive sound propagation with bidirectional path tracing. *ACM Transactions on Graphics*, 35(6):1–2.

[Firelight Technologies, 2021] Firelight Technologies (2021). FMOD.

[Funkhouser et al., 2004] Funkhouser, T., Tsingos, N., Carlbom, I., Elko, G., Sondhi, M., West, J. E., Pingali, G., Min, P., and Ngan, A. (2004). A beam tracing method for interactive architectural acoustics. *The Journal of the Acoustical Society of America*, 115(2):739–756.

[Jones, 2020] Jones, R. (2020). Ray tracing: Everything you need to know about ray tracing.

[Judd, 2020] Judd, W. (2020). AMD unveils three Radeon 6000 graphics cards with ray tracing and RTX-beating performance • Eurogamer.net.

[Kohata, 2018] Kohata, S. (2018). An Interactive Sound Dystopia: Real-Time Audio Processing in NieR:Automata - YouTube.

[Kuttruff, 2000] Kuttruff, H. (2000). *Room Acoustics, Fourth edition.* Spon Press, fourth edition.

[Martindale and Roach, 2021] Martindale, J. and Roach, J. (2021). Here Are All the Games That Support Nvidia's RTX Ray Tracing | Digital Trends.

[Mehra et al., 2013] Mehra, R., Raghuvanshi, N., Antani, L., Chandak, A., Curtis, S., and Manocha, D. (2013). Wave-based sound propagation in large open scenes using an equivalent source formulation. *ACM Transactions on Graphics*, 32(2).

[Microsoft, 2021] Microsoft (2021). Project Triton - Immersive sound propagation - Microsoft Research.

[Milijic, 2021] Milijic, M. (2021). 45+ Video Game Industry Revenue Statistics: Game On!

[Raghuvanshi, 2017] Raghuvanshi, N. (2017). 'Gears of War 4', Project Triton: Pre-Computed Environmental Wave Acoustics. *Gdc 2017*.

[Raghuvanshi et al., 2009] Raghuvanshi, N., Narain, R., and Lin, M. C. (2009). Efficient and accurate sound propagation using adaptive rectangular decomposition. *IEEE Transactions on Visualization and Computer Graphics*, 15(5):789–801.

[Raghuvanshi and Snyder, 2014] Raghuvanshi, N. and Snyder, J. (2014). Parametric wave field coding for precomputed sound propagation. *ACM Transactions on Graphics*, 33(4).

[Raghuvanshi and Snyder, 2018] Raghuvanshi, N. and Snyder, J. (2018). Parametric directional coding for precomputed sound propagation. *ACM Transactions on Graphics*, 37(4).

[Raghuvanshi et al., 2010] Raghuvanshi, N., Snyder, J., Mehra, R., Lin, M., and Govindaraju, N. (2010). Precomputed wave simulation for real-time sound propagation of dynamic sources in complex scenes. *ACM SIGGRAPH 2010 Papers, SIGGRAPH 2010*.

[Raghuvanshi et al., 2017] Raghuvanshi, N., Tennant, J., and Snyder, J. (2017). Triton: Practical pre-computed sound propagation for games and virtual reality. *The Journal of the Acoustical Society of America*, 141(5):3455–3455.

[Rosen et al., 2020] Rosen, M., Godin, K. W., and Raghuvanshi, N. (2020). Interactive sound propagation for dynamic scenes using 2D wave simulation. Technical Report 8, DigiPen Institute of Technology.

[Sanglard, 2018] Sanglard, F. (2018). *Game Engine Black Book: Wolfenstein 3D v2.1*. Software Wizards.

[Schissler and Manocha, 2016] Schissler, C. and Manocha, D. (2016). Adaptive impulse response modeling for interactive sound propagation. *Proceedings - 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, I3D 2016*, pages 71–78.

[Schneider, 2010] Schneider, J. B. (2010). *Understanding the Finite-Difference Time-Domain Method*. John B Schneider.

[Siltanen et al., 2010] Siltanen, S., Lokki, T., and Savioja, L. (2010). Rays or Waves? Understanding the Strengths and Weaknesses of Computational Room Acoustics Modeling Techniques. In *International Symposium on Room Acoustics*, Melbourne.

Appendix

Code Base

OpenPL is hosted at https://github.com/KarateKidzz/OpenPL (28th May 2021).

The GitHub repository is publicly available and ready to download.

The user can run the installation scripts and add the dynamic libraries to their projects as needed.

SimulationVersusNoSimulation.mp4

This video shows the effects of OpenPL compared to a scene with no simulation.

RaycastVersusNoSimulation.mp4

This video shows the effects of raycasting for audio effects. The simulation sounds correct in one instance but is too strong in another.

On the other hand, OpenPL produces a believable result in both instances.

WwiseUnity.mp4

This video shows the integration of OpenPL between Wwise and Unity.

The effect is similar to that of FMOD, however, the strength of the occlusion is different.

FMODUnreal.mp4

This video shows the integration of OpenPL between FMOD and Unreal Engine.

The user will notice problems in the geometry, which is discussed in the paper.

WwiseUnreal.mp4

This video shows the integration of OpenPL between Wwise and Unreal Engine.

The user will notice problems in the geometry, which is discussed in the paper.