An Analysis of Accurate, Real-Time Reproduction of 3D Acoustics in Virtual Environments

A Dissertation

Presented to the

Graduate Faculty of the

University of Louisiana at Lafayette

In Partial Fulfillment of the

Requirements for the Degree

Doctor of Philosophy

Scott McDermott

Fall 2014

An Analysis of Accurate, Real-Time Reproduction of 3D Acoustics in Virtual Environments

Scott David McDermott

APPROVED:

C.-H. Henry Chu, Chair Professor of Computer Science The Center for Advanced Computer Studies	Christoph Borst Associate Professor of Computer Science The Center for Advanced Computer Studies
Anthony Maida Associate Professor of Computer Science The Center for Advanced Computer Studies	Mary Farmer-Kaiser Interim Dean of the Graduate School

Acknowledgments

I would like to acknowledge the many people, both friends and family, who have supported me (or patiently sat by and watched me toil) through the many, many years that it took to get to this point. You are too numerous to list here, but your collective strength and encouragement was the foundation of my efforts. We are nothing without those near us. Thank you all!

To my wife Diana, who, through the countless years of trials and chaos has learned how to be patient and yet constant, consistent and yet not insistent. Simply put, there would have been no possibility of conclusion without your love and encouragement. Thanks are simply not sufficient.

I would be remiss not to mention my daughter and son, who make my life complete. I had every intention to complete this before each of your births so that you would not need to suffer through my distraction. Thank you as well for being patient and understanding. I hope that you have learned perseverance and resolve from this experience.

Finally, to my advisor, Dr. Henry Chu, others probably cannot comprehend how patient and helpful you have been through all of these years. Simply put, I would not have finished this document, many times over, except for your understanding and backing.

Table of Contents

Table of Figures

Table of Tables

Table of Equations

Introduction

Abstract

Many of the applications, virtual environments, and video games available to average computer users integrate stunning three-dimensional (3D) graphics and real-world visualizations. Developers spend an extraordinary amount of time and effort creating these immersive, realistic virtual environments, primarily focusing on the graphics components. Within these virtual realities, the user should easily perceive the locations of sound sources accurately, as well as the acoustic nature of the environment. However, for reasons of economy and simplicity, most developers apply readily available industry standards for generating pseudo-3D sounds in their applications. This research explores the shortcomings of these standards, proposes an effective alternative, and provides a detailed analysis of the various possible approaches.

This project includes a number of computationally efficient, physics-based 3D acoustics simulations, each of which will produce realistic aural reproductions. The primary goal is to evaluate and compare these algorithms against each other, non-3D sound reproduction, and the current industry standards (e.g. Microsoft's DirectX® pseudo-3D algorithm). We will test three hypotheses. First, users will find that physics-based 3D algorithms will render improved auralization reproductions compared against industry standards like DirectX® and/or OpenAL. Second, localization and spatialization will improve with user training when using these algorithms. Finally, we should discover an unambiguous ranking system for the quality of each tested algorithm.

What is 3D Sound?

Of the five human senses, society generally regards vision as the most significant to survival. People constantly depend on sight for daily activities in order to navigate through their surroundings without accident or injury. Next in line of importance to survival would easily be the sense of sound. When one suddenly loses the ability to hear, he is almost as vulnerable as if he were blind. Audio cues constantly give volumes of information about what happens in proximity to the listener. These cues allow individuals to avoid collisions with other objects, know the relative location of something or someone, and especially facilitate communication. Keeping this in mind, to create effectively a convincing virtual 3D environment we must include sound and noise in any reproduction. Without accurate acoustics, a virtual 3D experience could easily become as frustrating as watching TV without the sound. Yet, is it enough simply to play the appropriate sounds for the environment? How accurately must we represent the 3D sound reproduction?

In order to answer this question, one must first understand the final product of what a 3D virtual environment, or virtual reality, attempts to accomplish. Concisely, a user should find himself in a completely immersive simulated experience in which it becomes difficult, if not impossible, to distinguish between reality and illusion. Ideally, this would look similar to the "holodeck" portrayed in the science fiction genre (Weinberger, 2007) where the actors, or avatars, actually walk into a generated virtual environment and completely interact with the characters and the surroundings. However, the current level of technology in computer graphics, processor speed and bandwidth, haptic responses, and audio synchronization do not allow for this level of experience, so we must approximate it. Presently, some of the more advanced 3D environments use multiple screens surrounding the user, displaying in two-dimensions a computed 3D virtual environment (Courchesne, 2007). This gives a rough, but convincing sense of visual 3D immersion. As graphical and general processing speeds increase, these simulations will become even more impressive. However, most of these applications employ decades old sound processing techniques and algorithms or simply use none at all! It is uncertain why sound has taken a backburner to visual development (Flaherty, 1998), but we see some evidence that sound processing has become more important to these immersive virtual environments. Many programming libraries currently available include at least a nod, if only trivial, to 3D sound algorithms (see “Sound API’s” on page 153).

This paper approaches the concepts and dilemmas associated with 3D virtual sound calculations. We first explore the current state of the art and industry standards for dealing with sound and 3D sound on computers. Additionally, we provide an analysis of past and current research dealing with 3D sound calculations, followed by a detailed breakdown of the algorithms used in this research. Next, a discussion of the actual implementation includes details about the simulator, computation considerations, data structure designs, and programming issues. We then employ this simulator in a formal experiment to determine the

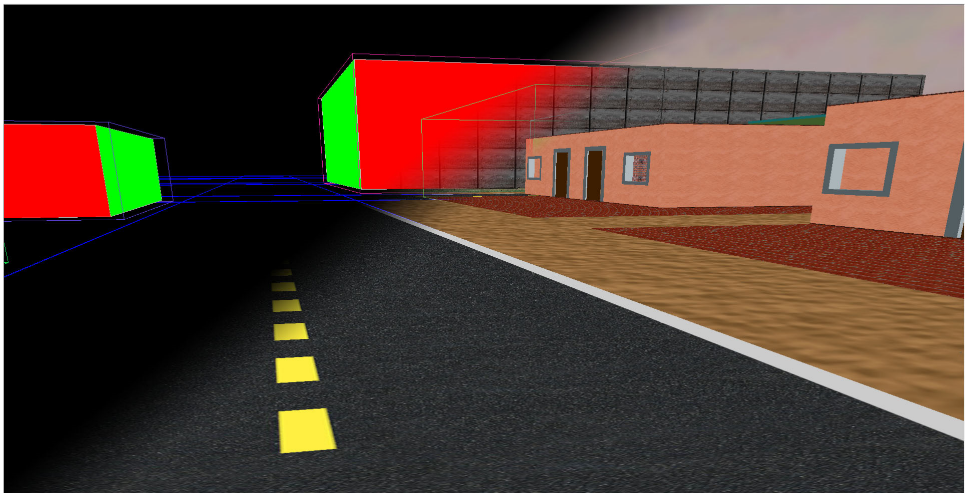


Figure 1: Immersive 3D Virtual Environment

best algorithm and methods to ascertain our conclusions. A detailed examination of the experiment and research methods follows, as well as formal analysis of the results and the effectiveness of the simulator and the algorithms investigated. Finally, we provide an appendix that includes an examination of the physical and physiological aspects of sound within the real world, and how they apply to virtual 3D sound. The appendix also contains technical aspects appropriate to the topic that might otherwise distract from the discourse at hand.

Goals

This research intends to provide a better understanding of the implementation and importance of 3D sound in virtual environments. Undeniably, both industry and academic environments have historically undervalued 3D sound. Focusing resources on this topic will encourage the devolvement of truly immersive virtual worlds which society can benefit from in uncountable ways. As always, the fundamental difficulty lies in the implementation. Algorithms for 3D visualization have evolved over decades from slow offline techniques such as ray tracing to real-time libraries and hardware realization (see “Computational Issues” on page 153) to blazing fast hardware supported libraries that can generate complex graphics in real-time. Creating analogous algorithms and eventually libraries and hardware for 3D sound will certainly lag behind what end-users will desire once the industry adopts proper standards for 3D acoustics. In a sense, the cliché “If you build it, they will come” could easily apply to 3D sound algorithms, as we can make arguments that more realistic acoustics will only improve the overall experience in any virtual environment.

Objectives

The two most important aspects of this research are the design, development, and implementation of specific 3D sound algorithms and thereafter the determination of methods for assessing and comparing the effectiveness of such algorithms. This manuscript includes a thorough description of the algorithms created for this research as well as design choices made in the process of development. We will provide pseudo code and data structure abstractions to convey the nature of the formal source code of the research. In order to analyze the appropriateness and effectiveness of these algorithms, we have developed a prescribed study involving test subjects and included a detailed explanation of this experiment. An analysis of the experiment follows this, complete with conclusions and considerations for further development.

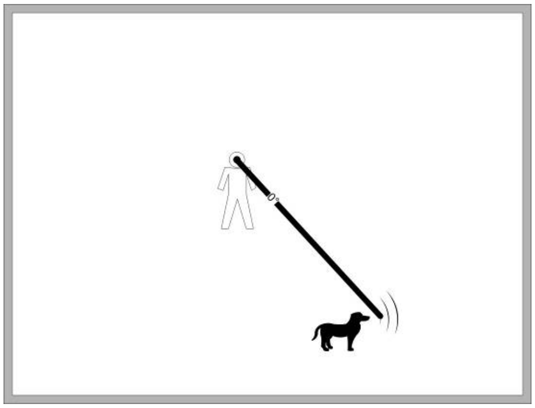


Figure 2: Zero-Order Reflections

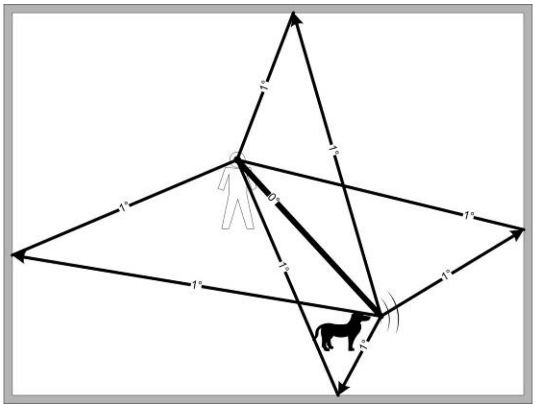


Figure 3: Adding First-Order Reflections

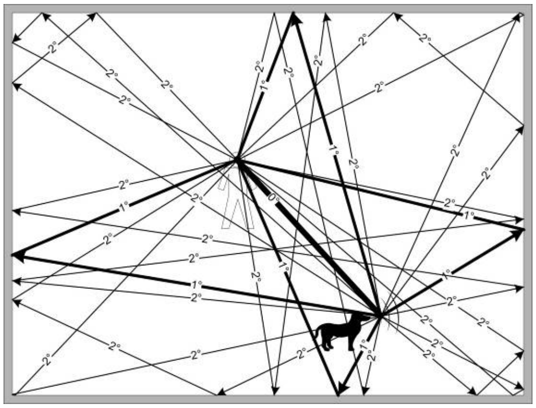


Figure 4: Up to Second-Order Reflections

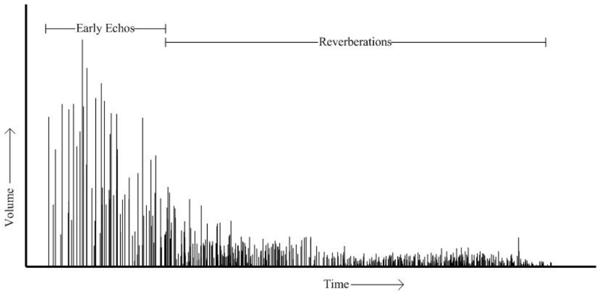


Figure 5: Finite Impulse Response

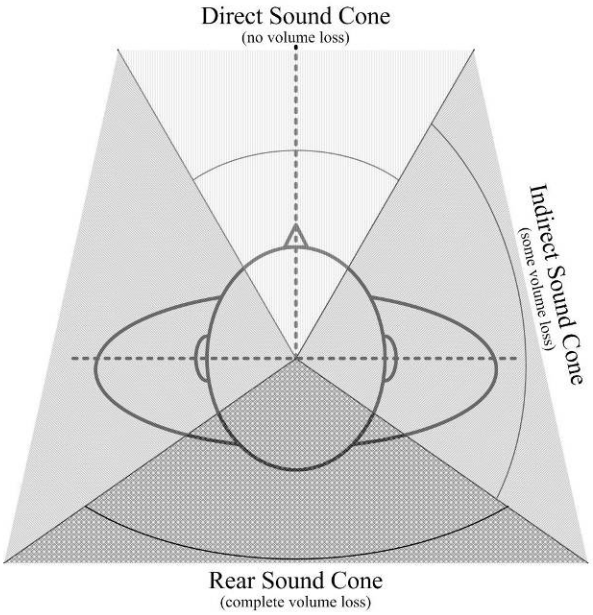


Figure 6: Sound Cones (Volume Loss)

$$y_L(t) = \int_0^\infty h_L(\tau)x(t-\tau)d\tau$$

$$y_R(t) = \int_0^\infty h_R(\tau)x(t-\tau)d\tau$$

Equation 1: Convolution of Impulse Responses

Algorithms

At the heart of any approach to the problem of providing realistic 3D sound in virtual environments will be the creation and evolution of effective algorithms. The purpose of any 3D acoustic algorithm is simple and specific. Regardless of the methods used, any valid and effective algorithm will most likely need to generate a finite impulse response, or FIR (Figure 5) that represents the acoustic relationship between the sound source, the virtual geometry and spatialization, and the listening avatar (see “3D Sound Perception” on page 148). Since we store most digital sound in basic mono or stereo formats, reproductions must determine, generate, and apply all acoustic spatial properties. For this research, we begin with only 8-bit or 16-bit monophonic PCM audio files (see “Sound Storage” on page 72) and transform the audio stream to stereo with a 3D nature.

Consider a geometrically simple virtual space (Figure 2 through Figure 4) and sound emanating out from a source equally in all directions. Sound will reach the listener avatar in three manners. First, sound can travel line-of-sight straight to the listener. This direct path sound is the most obvious path and the most important acoustically. Also called the zero-order path (Figure 2), the direct path is almost always the loudest, first to arrive, and the clearest. It gives the listener the most relevant information about the nature of the sound, including clear content, initial directionality and distance, and many other cues (see “3D Sound Perception” on page 148). The next type of sound the avatar perceives consists of the early echoes, or initial reflections. As detailed in the appendix, an approximate point in time exists when sound changes from early echoes to reverberation. The acoustic nature of the two has significant impact on sound localization and especially spatialization and therefore we consider important to the algorithms. Figure 3 and Figure 4 illustrate some of the first-order and second-order paths, respectively. These diagrams, by no means, present all appropriate paths emanating from the sound source.

Implementation and the formal data structure design may vary between approaches, but the concept and how to utilize a FIR typically remains constant. When each path, from zero-order to second-order and beyond, reaches the listener from the sound source, we record a hit, or impulse, at the appropriate time and volume level. Since sound travels at perceptible speeds and the volume depreciates as it propagates through air, the resulting set of pings produces an array of impulses that to some extent can describe the room. Figure 5 diagrams an example of a FIR, graphed volume versus time. The y-axis in the example represents the volume of the impulse, but we can invert it to determine the volume level loss of the original sound for playback (see "Early Echo Response & Reverberation" on page 152). With this FIR in hand, we simply replay the original sound file repeatedly at the proper times and according to the calculated (generally exponentially decreasing) volume levels. Discussion of both our design and implementation follows as well as consideration of the acoustic engine design and algorithms.

For the purposes of virtual environments, we can assume the sound sources exist as generally infinitely small points in space with sound radiating out in all directions equally (Stephenson, 2013) and without a physical dimensional sound field based on the shape of the object. Initially, we treat a person talking in the same manner as a dog barking or a door closing: just a point in space, centered on or in the object, radiating sound outward. Our approach can compensate some for these assumptions by considering the direction and orientation of the sound source within the algorithm. Furthermore, we can, in effect, apply some of the spatial interference from the mass and shape of the object through directional volume level cones. These cones consist of three encompassing regions where we can dampen the sound volume depending on direction emanating from the source (Figure 6) from full volume in front of the object to none or almost no sound behind the object. These assumptions significantly simplify calculations while still allowing directionality of the sound source.

In the next section, we describe and analyze other proposed approaches to the problem of creating virtual environments with 3D sound. This will include both academic and industrial approaches. Subsequently, we will illustrate in depth the various approaches we have taken and then investigate and compare the results of these implementations.

Previous Work

Though relatively little direct development (Vorländer, 2011) (Stephenson, 2013) has occurred in this specific field, a number of related works have contributed to 3D sound generation. In the appendix of this paper (see "Computational Issues" on page 153), we will explore the industrial technical developments that have supplemented this research. Effectively though, the computer industry has contributed little to nothing appreciable to this field in the past few decades. Programming libraries such as Microsoft's DirectX DirectSound® and Creative Lab's OpenAL® have demonstrated an aspiration toward 3D sound enrichment. However, industrial follow-through, investments, and development in these utilities have truly fallen short of inspirational.

Despite the direction (or lack thereof) of industry development in this field, a small set of dedicated researchers have applied some degree of effort in developing true 3D sound algorithms. Here we discuss some academic enhancements and consider their impact on the field of 3D sound reproduction.

Geometrical vs. Wave Equation Based Methods

Fundamentally, almost all development in this field boils down to one of two approaches, or some hybrid thereof (Antani, Chandak, Savioja, & Manocha, 2012) (Raghuvanshi, Snyder, Mehra, Lin, & Govindaraju, 2010). Relatively visually and computationally simple, geometrical models attempt to treat sound as particles, emanating from a sound source in all directions. This approach includes ray-tracing, image source reflections, and beam trees as well as the algorithms developed in this research. Though the specifics of design and efficacy might vary dramatically between systems, geometric models typically suffer from the same concerns. Since they are not frequency dependent, geometrical models by nature do not provide refraction or phase control and generally remain accurate only for higher frequency signals. Geometrical techniques can generate low-ordered reflections extremely fast and accurately, but have questionable performance with respect to reverberation and other sophisticated acoustical properties.

On the other hand, true wave equations methods theoretically avoid these issues. This set of solutions treats sound from a physics-based perspective, attempting to solve second-order partial differential equations, in real-time. Algorithms utilizing finite-element and boundary-element methods as well as finite-difference time-domain comprise this group. Wave equation methods do not suffer from frequency related constraints and should offer a superior, generalized solution to 3D virtual acoustics. However, they do so at a dramatic cost of extreme processing and storage requirements, rendering them almost impractical for real-time implementation on modern processors.

A small, third set of research includes statistical models. These approaches generally produce efficient, but inferior results compared to geometric or wave equation methods.

We make note that, regardless of the approach, almost all research in this field begins with the fundamental premise of the goal to calculate a finite impulse response (see "Algorithmic Goal" on page 46) in some form. After calculation, the research uniformly agrees that the system should convolve the FIR with a monaural sound signal to produce a 3D acoustic effect. Unsurprisingly, the difference lies in computing this impulse response.

Beam Trees (2004)

A group of researchers at Princeton (led by Thomas A. Funkhouser) historically were some of the very few who have spent any amount of time on this topic. They have approached this challenge from a data structure point-of-view. By using beam trees to optimize retrieval, they have created a system that pre-computes direct paths of what they call beam tracing. Their system is generally efficient with real-time results but makes some major assumptions. First, they assume that beam tracing will produce accurate results, even though they state early on that similar algorithms are subject to "aliasing and errors in predicted room" (Funkhouser, Carlbom, Elko, Pingali, & Sondhi, A Beam Tracing Approach to Acoustic Modeling for Interactive Virtual Environments, 1998). It is still not evident that the use of beam trees alleviates this. Second, the algorithm they designed is dependent on the environment not changing. Though this is a logical assumption, it would be nice to have a dynamic algorithm. Finally, the sound source and listener locations must be stationary. Unfortunately, this research group has published nothing for this topic since 2004. However, other groups have attempted to extend this approach.

Accelerated Beam Tracing (2009)

The developers of this approach (Laine, Siltanen, Lokki, & Savioja, 2009) significantly enhanced the previously described beam tracing method by incorporating common sense algorithm optimizations. This new system reorganizes and simplifies the precalculation data structure, making considered and logical assumptions on the required data. Additionally, they have dramatically optimized the run-time algorithm by eliminating unnecessary beam propagations. The authors have even included predictive methods (buckets of path nodes to skip) to further scale processing requirements.

Unfortunately, this method still suffers from speed and resource limitations. The authors attempt to allow movement of the listener by forcing updates to the "precalculations" as the listener moves. Assuming the system can process these calculations fast enough, this approach has merit. However, results show that it can only handle spaces with "moderate model complexity" in real-time. Finally, the article does not include any evaluation of the efficacy, subjective or objective, of this research.

Precomputed Wave Simulation (2010)

Though a wave equation approach should produce far superior results than a geometrical model, this research (Raghuvanshi, Snyder, Mehra, Lin, & Govindaraju, 2010) lies more in the realm of proof of concept than a potential solution. The researchers have attempted to produce a purely wave equation based solution that runs in real-time. To make this extremely complicated solution run in real-time, the main contribution of this approach is to divide the space into a grid and interpolate FIRs at run-time.

To be fair, the final product does in fact run extremely fast. However, the many major assumptions call into question the correctness of the solution. Most notable of these assumptions include interpolating impulse response over a 2D grid, locking the listener to a 2D plane, employing a "frequency trend representation" for impulse responses, and the confusing statement that "in many applications, the listener's position is more constrained than the sources'." Even anecdotal review of the simulation supports these concerns, as the reverberation often becomes distracting, if not disturbing. In fairness, to date, this research constitutes the first and possibly only wave equation based real-time approach and certainly begs for future development.

Augmented Reality (2011)

In Finland, the Department of Media Technology at Aalto University School of Science has come across a unique approach to immersive virtual environments. They have brought together the critical aspects of virtual reality, both visual and audio, into one research group. Ostensibly, this could afford a more balanced and enhanced virtual experience. Unfortunately their primary focus resides in virtualization (3D sound reproduction) or as they describe it, augmented reality.

The approach in their audio augmented reality research has applicable similarities but does not directly address the dilemmas put forth in this paper. They are primarily concerned with the "cocktail party effect" (Gamper & Lokki, 2011) and virtually placing sound sources relative to a listener in a virtualized acoustic space. A listener using proprietary earphones is able to virtually position multiple audio streams in order to separate and distinguish conversations. He accomplishes this by snapping his fingers in front of him, causing a sound source to appear to originate from that location.

Though impressive and potentially useful, the main portion of the research resides in stereoscopic reproduction. However, the manner of determining the relative locations of the snapping somewhat parallels the research in this paper. The specially designed earphones include small microphones that allow a mixer to find the snapping sound. Using various audio processing techniques, the system generates an impulse response of the physical room from the location of the snap. The system then convolves a monaural audio signal with the processed response to create a convincing virtualization in a physical space. In a sense, this research lies somewhere between HRTF refinement and true 3D virtualization.

Reverberation Shading (2011)

Though the authors of this paper (Cowan & Kapralos, A GPU-Based Method to Approximate Acoustical Reflectivity, 2011) do not address the problem of sound localization in 3D virtual environments, they provide a unique approach to 3D sound spatialization. This research falls into the statistical category with respect to algorithm approach. Instead of calculating numerous mathematically intensive sound paths from sources, they use readily available graphics shading algorithms to estimate the global reverberation effect of the room. The system analyzes the relative distance and orientation of each surface, while considering reflection/absorption values, and computes an overall room reverberation factor. Clearly, this heuristic model cannot accurately account for all properties of a given space, but when dealing with only the reverberation portion of sound virtualization, the approximation may have merit. Given the relative simplicity of this approach, it is certainly tempting to include it in future research. Unfortunately, the authors do not offer any formal analysis of the efficacy of this procedure.

Beam Tracing with Refraction (2012)

This research extends the previously described beam tree algorithm by focusing on refraction. Ray tracing approaches such as beam trees, image reflections, and those presented in this document suffer from lack of refraction because they do not consider phase and frequencies of the source signal and propagation through diverse boundary conditions. The authors of this paper (Sikora, Mateljan, & Bogunović, 2012) attempt to address this deficiency by creating a complicated, robust data structure and algorithm sensitive to these concerns. The primary difference between this research and generic beam trees lies in the subdivision of beams when the algorithm encounters boundary surfaces between two media. While ostensibly effective, this algorithm does not run in real-time and requires extensive resources. However, future refinement and optimizations could lead to practical hybrid applications. Unfortunately, the authors do not offer any formal analysis of the efficacy of this procedure other than numerically comparing to other known models.

Precomputed Acoustic Transfer Operators (2012)

This relatively new approach (Antani, Chandak, Savioja, & Manocha, 2012) to 3D sound in virtual environments has incredible potential for future implementation. Precomputed Radiance Transfer method encodes light propagations between surfaces in a scene, which allows for quick estimation of diffuse lighting during rendering. Based initially on this method, this approach attempts to predetermine the effects and impact of reverberation from the virtual surfaces on the listener and sound source configuration. Though the designers have made quite a number of assumptions to optimize the efficiency of the algorithm, since they only intend to use this method to calculate reverberation, the results may be sufficient. Even in extremely complex environments and allowing common ray-tracing approach to determine low-ordered reflections, this method clearly runs at very close to real-time. The authors clearly have exerted much effort in carefully considered optimization of precalculations and data storage routines. Regardless, the demands on processor and storage outside of run-time are substantial.

Unfortunately, this paper only describes the technical merit of this approach. Though extremely detailed in the mathematics and rationalization behind the algorithm design, the authors neglect to examine the effectiveness of the algorithm. Does this produce superior or even effective results? How would altering the allowed variables formally affect the auralization? The researchers indicate that the system can scale to processor load, but leave the analysis in the mathematical realm only. Finally, as the authors note, this approach still suffers from the same dilemmas that plague all geometric designs. Informal evaluation of the video demonstration provided by the authors leads us to believe this approach has strong potential and merits further exploration.

Spatial Sound and Visual Fidelity (2013)

The authors of this recently published paper (Cowan, Rojas, Kapralos, Collins, & Dubrowski, 2013) fully acknowledge the importance of proper 3D acoustics in virtual environments and yet have only recently begun to explore implementation and efficacy. They repeatedly state that the audio component must "go far beyond traditional stereo and surround sound techniques."

Unfortunately, this experiment has numerous fundamental flaws. First, the authors set the stage in a virtual operating room, asking the user to perform knee arthroplasty. Unquestionably, the average computer user could find this type of task intimidating and unapproachable. Second, the experiment allowed for variation of shading levels—distracting the focus from the auditory aspect. Third, the 3D acoustic portion of the experiment incorporated a generic HRTF approximation on an actual drill sound. Finally, the experiment task was incredibly simple, the focus group too small (ten people), and evaluation was based on a user inputted ranking system. The authors unsurprisingly did not find a correlation between sound spatialization and visual fidelity.

Hybrid Approach - Acoustic Radiance Transfer (2013)

This very recent paper (Southern & Siltanen, 2013) discusses the inherent problems with 3D acoustics in virtual environments. Though the authors do not offer any substantial evaluation or analysis of their proposal, the concepts brought forth certainly require further examination. The paper asserts that, due to the frequency dependent nature of sound propagation and reflections, one algorithm for computing an impulse response is insufficient. Rather, the authors propose a combination of three divergent models. By exploiting the strengths of three different methods, this approach has the potential to generate superior acoustic realizations. This research group suggests using beam trees, or some geometric propagation model for early, general reflections. The Finite Difference Time Domain (FDTD) algorithm handles lower frequencies since it approximates pure wave equations. Finally, the Acoustic Radiance Transfer (ART) takes over for higher frequencies caused by later reflections, or reverberation. The latter method uses an energy-based boundary element algorithm that lies somewhere between ray-based and wave equation approaches (Siltanen, Lokki, & Savioja, Rays or Waves - Understanding the Strengths and Weaknesses of Computational Room Acoustics Modeling Techniques, 2010).

The authors acknowledge the inherent awkwardness of combining multiple approaches into one FIR for convolution purposes. They do offer an “empirically derived” modifier to mitigate this issue. However, the research offers no evaluation or analysis to justify this approach. In fact, reviewing the numerous publications regarding the Acoustic Radiance Transfer method, it is apparent that this algorithm cannot solve the problem alone as the authors repeatedly offer hybrid approaches that seem to mitigate the shortcomings of

ART. Once again, we find no formal or even subjective analysis of the efficacy of this approach.

Wave-Based Sound in Open Scenes (2013)

Recently, the research group at UNC has put forth another solution based on wave equations. This paper (Mehra, et al., 2013) in some sense compliments the 2010 approach by focusing on spatialization in large, open spaced scenes instead of smaller, enclosed spaces. The mathematics and physics inherent to the different spatial environments require significantly different approaches to solving wave equations efficiently. In this case, the designers have divided the scene into objects and enclosed each object in boundary areas. They then generate transfer functions to predict the "acoustic behavior" of objects with respect to each other and the source and listener objects. At runtime, the system parses and combines these transfer functions to generate meaningful sound spatialization.

Due to the extremely complicated math involved, most of the calculations occur offline. Even using a 64-node CPU cluster, precomputations require on the order of hours to process. However, storage requirements fall in the more reasonable range of tens of megabytes. Unfortunately, mostly due to the nature of the processing requirements, the system has quite a few significant limitations. Either sources or listeners must remain static and it requires static scenes in general. Furthermore, the algorithm cannot model Doppler effects and larger scale outdoor scenes (larger than kilometers) require too much memory. Yet, the system does run smoothly, apparently in real-time. However, the authors only provide mathematical analysis of validity and offer no evaluation of efficacy. Anecdotally, the provided simulations in the video suggest that the approach works sufficiently, but these have limited scope of content. Unlike the previous wave equation model for enclosed spaces, we generally find the spatialization enhances perception, with some exceptions. Often, though, the simulator actually exaggerates some effects to the point of distraction.

Current Work

The primary goal of this research is to develop our own 3D sound algorithms and measure them against each other and at least one industry standard algorithm. To do this, we have developed a robust 3D virtual environment simulator (see "The TDS Simulator" on page 49) from the ground up. This simulator allows us to implement any number of 3D sound routines and even design an experiment to evaluate the effectiveness of each.

This work has many avenues for further development. Future work may include adding and refining algorithms, calculations over multiple rooms and multiple structures, taking into effect other acoustical properties or cues, and allowing for run-time virtual environment changes. Though the simulator and algorithms run on a Windows® platform, the basic code remains strictly portable (see "Portability" on page 90) to other programming environments. Furthermore, we plan enhance methods of objectively comparing the results of concurrent algorithms. Because the simulator allows for analysis of multiple algorithms simultaneously, the logical next step would include development and investigation other

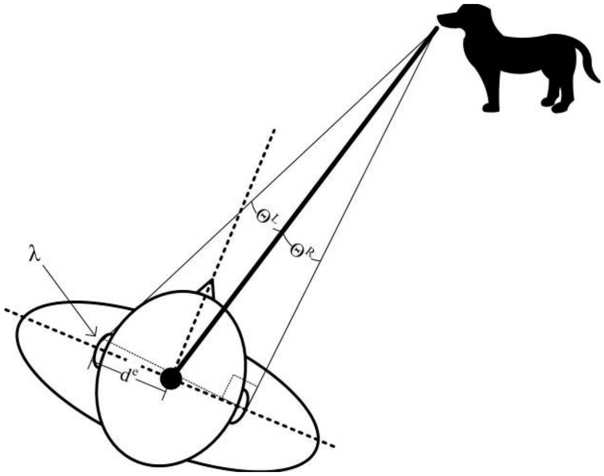


Figure 7: Direct Paths Algorithm

methods of generating 3D sound. Below we describe our algorithms, both implemented and proposed.

Direct Paths Algorithm

The first algorithm we developed specifically compares to those used by industry standards such as Microsoft's DirectX DirectSound® and Creative Lab's OpenAL®. This routine calculates only the zero-order path (see Figure 2 on page 17) where sound travels directly from the source to the listener. We actually consider two paths separately, one for each ear. This innately allows for stereoscopic auralization based on interaural delay time, head shadow, and head motion (see "3D Sound Perception" on page 148). Sound will naturally arrive at one ear a few milliseconds before it reaches the other ear, depending on the proximity between the listener and sound source and the orientation of the listener relative to the source. See Figure 7 for an illustration.

This approach consists of only the most basic requirement for generating 3D sound. It does not take into account any aspects of the virtual environment except the locations and orientations of the generating sound source and the listening avatar. The algorithm ignores room geometry, surface properties, and all other forms of interference and obstructions. We include this algorithm in our development and experiments for one reason: comparison to industry standard algorithms. Both Microsoft's DirectX DirectSound® and Creative Lab's OpenAL® only consider the zero-order path. Any effects of reverberation, echoes, or other acoustic enhancements come from uncorrelated filters. The current OpenAL® specification [http://connect.creativelabs.com/openal/Documentation/OpenAL%201.1%20Specification.htm] states:

> OpenAL (for "Open Audio Library") is a software interface to audio hardware. The interface consists of a number of functions that allow a programmer to specify the objects and operations in producing high-quality audio output, specifically multichannel output of 3D arrangements of sound sources around a listener…
>
> OpenAL does include extensions compatible with the IA-SIG 3D Level 1 and Level 2 rendering guidelines to handle sound-source directivity and distance-related attenuation and Doppler effects, as well as environmental effects such as reflection, obstruction, transmission, and reverberation.

Note that OpenAL® treats important acoustic information such as reverberation and reflections as secondary "effects" and makes no formal effort to calculate them. We explore these shortcomings in more detail in the appendix "Sound API's" on page 153.

```
startTimeCheck();
setVariables();
locateListenerEars();
getDistanceToEars();
getTimeToEars();            // (distance/SOUND_SPEED)
getDelayToEars();           // (SamplesPerSec*Time)
getOrientationToEars();     // (SourceOrientation-VectorFromSource)
getOrientationWeights();    // (1-OrientationWeight)+(orientation*OrientationWeight)
getAttenuatedSignals();
endTimeCheck();
```

Table 1: Direct Paths Algorithm Pseudo-Code

Computationally and conceptually, our algorithm contains no significant challenges. It does not require loops or nested functions and therefore runs linearly and extremely fast, with practically no delay. All calculations such as dot products and cross products use basic trigonometry and other straightforward math. The pseudo-code follows in Table 1 above and we describe the implementation later (see "SoundPlayDirectPaths Algorithm Object" on page 63). Our initial impression and assessment of this algorithm shows promise, despite its inherent simplicity. Unlike other potential algorithms, the direct paths algorithm does not suffer from aliasing or other artifacts directly because of the uncomplicated nature of the method. For this reason as well as the logical comparison with industry standards, we have chosen to implement this algorithm in the experimental test (see “Testing Details” on page 99). Notably, this algorithm does not generate a formal finite impulse response like most other approaches. In effect, though, the two calculated paths comprise the most basic of possible FIR’s.

Brute Force Algorithm

The most simple and obvious starting point to more complicated algorithms would be a basic brute force algorithm. This technique would consist of calculating an almost infinite number of acoustic paths emanating out from the sound source. Not all paths would reach the listener, making this algorithm extremely inefficient. Furthermore, the complexity and sheer number of calculations required would make this approach impractical at best. Run time on this type of method certainly would not fall within acceptable requirements for our purposes, so we chose not to pursue this methodology. However, we can consider modifying a brute force algorithm with common-sense optimizations to create effective and exploitable routines.

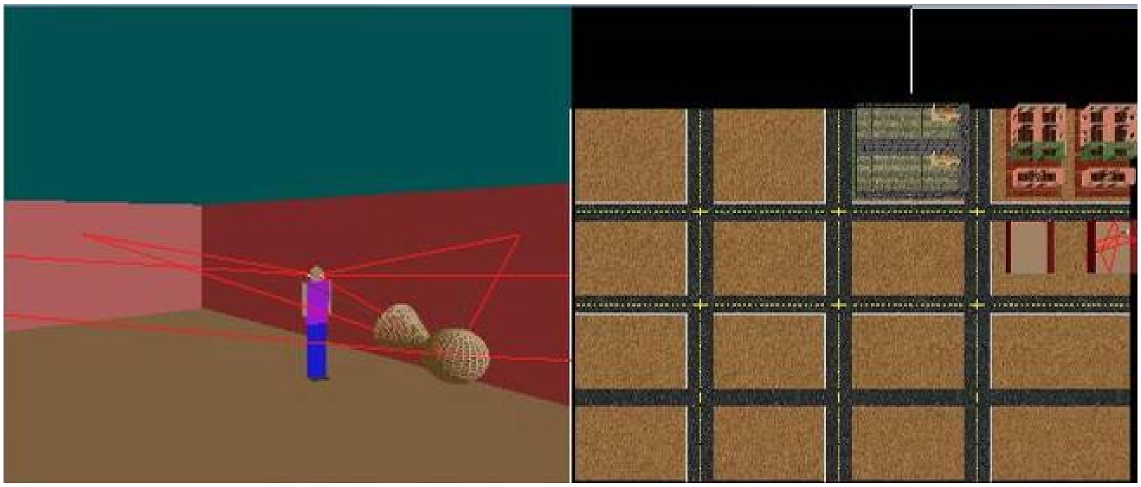


Figure 8: TDS Running the Reflected Paths Algorithm

Reflected Paths Algorithm

Another algorithm we developed for this research, the reflected paths algorithm, begins as more or less a brute force approach to the problem. We limited consideration to only sound paths that traverse from the source to the listener. Figure 2, Figure 3, and Figure 4 (page 17) diagram these for the zero-order, first-order, and second-order reflection paths respectively for a room with just four walls. This algorithm calculates these paths via simple, semi-optimized mathematical techniques. Simple trigonometry and algebra dominate the math required for this algorithm.

The depth complexity level setting for this algorithm controls how many reflections, or level of order, to parse. As the order of reflections increases, the calculations needed to determine the sound paths also become exponentially more complex, approximately bounded by $O(4^n)$, requiring dramatically more processor time. Table 2 shows an example of this progression.

d	r_p	n_p	p(w=4)	r_t	n_t
0	0	1	1	0	1
1	1	2	4	4	8
2	2	3	12	12	36
3	3	4	36	108	144
4	4	5	108	432	540
5	5	6	324	1620	1944
6	6	7	972	5832	6804
7	7	8	2916	20412	23328
8	8	9	8748	69984	78732
9	9	10	26244	236196	262440

d - Order of reflections (depth)
w - Number of walls (e.g. {w = 4})
r_p - Total reflections/path [$r_p(d) = d$]
n_p - Total segments/path [$n_p(d) = r_p(d) + 1$]
p - Total paths [$p(d) = p(d - 1) * (w - 1)$]
r_t - Total reflections [$r_t(d) = p(d) * r_p(d)$]
n_t - Total segments [$n_t(d) = p(d) * n_p(d)$]

Table 2: Reflected Paths Complexity Levels

Currently, this algorithm only works well when both the sound source and the avatar reside in the same room of the same building. Expanding the algorithm to handle multiple rooms should not prove difficult. However, it will dramatically increase the algorithmic complexity, and consequentially computation time, thus making it impractical without further serious optimizations. Additionally, we account for sound source direction by gain-weighting sound paths emanating from directions closest to its orientation vector.

We list the pseudo-code for the initial algorithm in Table 3. Most of the algorithm uses simple proportions to find reflection points. Because of this, and the fact that the algorithm does not currently allow for breadth complexities, the code runs almost instantly with the present limited functionality.

Due to the finite nature of the number of paths to utilize and non-random character of these paths, this algorithm suffers from severe aliasing artifacts during sound reproduction. The fabricated signals tend to sound choppy and uneven when the avatar moves around, even slightly. To be sure, we could overcome some of this by significantly increasing the depth (number of paths) or other optimizations and considerations. However, the increase in complexity and loss of elegance would render the algorithm ineffective, not to mention inefficient. Therefore, for the purposes of the experimental study, we elected to leave this algorithm out and focus on the other two that we developed.

```
startTimeCheck();
setVariables();
for w...walls {
        findWallGeometry();
        closestPtSource = findClosestPointOnWall(source);
        closestPtAvatar = findClosestPointOnWall(avatar);
        D = distance(closestPtSource, closestPtAvatar);
        L = (avatar_wall_dist * D) / (source_wall_dist + avatar_wall_dist);
        bouncepoint = closestPtAvatar + L * wall_direction;
        checkIfIsOpening(bouncepoint);
}
endTimeCheck();
```

Table 3: Reflected Paths Algorithm Pseudo-Code

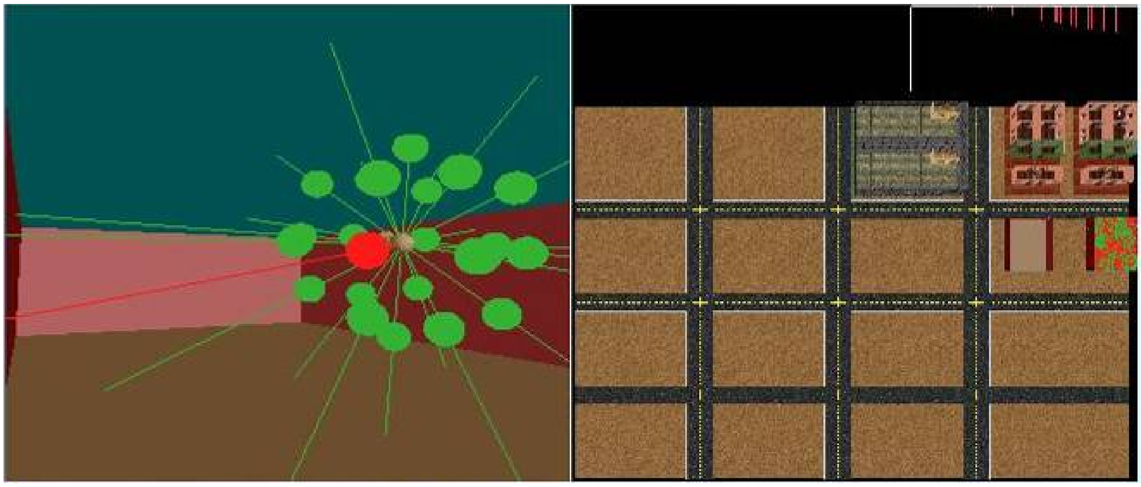


Figure 9: TDS Running the Bouncing Reflections Algorithm

Bouncing Reflections Algorithm

This algorithm differs from reflected paths algorithm above in that it calculates paths that do not necessarily reach the source. In this sense, it is more random and inefficient than the direct paths algorithm; however this makes it easier to optimize and eventually more accurate. The concept is simple enough. When a source generates a sound in the 3D environment, the algorithm calculates the paths emanating from the source in all cardinal directions and some or many random directions and then determines consequential reflections off the walls. Think of a large number of ping-pong balls shot out from the sound source in all directions. These balls bounce around the room (or rooms), unaffected by gravity. They eventually stop by either hitting the listener or traveling past a certain threshold distance. This maximum distance traveled corresponds to the dissipation of sound traveling through air. In fact, each path generally travels much less than the predetermined threshold distance since some dissipation occurs at each bounce, or reflection point on the walls and objects in the room. Most offline architectural acoustic modelers use this same concept, with a few key differences. First, none of these modeling programs run in real-time. For their purposes, architectural modelers have no need to run on the fly. Rather, they sacrifice speed for detailed accuracy, computing significantly more rays paths than our system could allow. Second, the process for offline architectural acoustic modelers effectively performs the reverse of what we try to accomplish. These programs intend to minimize undesirable acoustic properties of a given virtual space by analyzing how samples will sound throughout. Our system looks to estimate and determine all acoustic properties of a space and reproduce sound using this data. We attempt to reproduce even undesirable acoustic properties of a given virtual space. Moreover, we care about sound reproduction in only one location. The distinctions might seem subtle, but they are important.

The depth complexity level determines the number of reflections in a path to allow. On the other hand, the breadth controls how many directions emanating from the source to parse (i.e. 1, 6, 14, 22, etc.). We can initially set directions as strict cardinal vectors (x, y, z) or determine them based of the sound source orientation. The current algorithm uses the initial directions (ahead, back, left, right, up, and down) to further granulate the depth via subdivision of the vector spaces. Furthermore, we can select an almost infinite depth of randomly generated directions. Table 4 diagrams the first three sets of orientation-based vectors (on the right) and includes the actual code for determining these rays. Our present implementation begins with the sound source orientation and calculates 22 directions before randomly selecting directions.

Table 5 also contains the pseudo-code for the remainder of this algorithm and the formal implementation follows (see “SoundPlayBouncingReflections Algorithm Object” on page 65). See Figure 9 for an example run screenshot. With proper control of the breadth and depth of this algorithm, we can generate an intuitive, logical, and hopefully accurate impulse response. Since we currently calculate the direction vectors by uniformly dividing the space geometrically, sampling artifacts can certainly occur. The combination of structured calculations and randomized directions might mitigate some of these hot spots or empty spaces. Given enough depth and/or breadth, we should generate reproductions of the desired acoustic signals even more accurately. We will randomize these variables in the experiment (see “Research Methods” on page 93) in order to, hopefully, determine an effective cutoff point for complexity of the algorithm with respect to performance. Future research could show that employing an even more randomized distribution pattern would obtain better results and avoid possible anticipated sampling artifacts.

We should note that we determine intersection with the avatar by checking each path against two parallel bounded planes, one at each ear. Assuming these faces are properly sized this should catch almost all paths that the avatar would hear. We would only miss segments that run parallel or almost parallel to the planes or that hit the head, between the planes. If necessary, we can add a third plane through the head, between and perpendicular to the ear planes, and test against intersection for a more accurate model. However, this adds another degree of computational complexity to the algorithm, which we do not desire. Another possible method would test each path against the two perpendicular planes centered in the middle of the head. This could receive a more accurate set of path hits, but at the expense of the stereoscopic nature inherent in testing ear planes.

```
i=0;
dirs[i++] = sourceDir;                  //  0->Forward (already calculated recorded)
dirs[0].cleanVector();
dirs[i++] = -dirs[0];                                                 //  1->Behind
dirs[i++] = crossProduct(dirs[0],UP);                                   //  2->Left
dirs[i++] = -dirs[2];                                                  //  3->Right
dirs[i++] = crossProduct(dirs[2],dirs[0]);                                //  4->Up
dirs[i++] = -dirs[4];                                                   //  5->Down
dirs[i++] = (dirs[0]+dirs[2]).normalize();      //  6->Angle between forward & left
dirs[i++] = (dirs[0]+dirs[3]).normalize();     //  7->Angle between forward & right
dirs[i++] = (dirs[0]+dirs[4]).normalize();        //  8->Angle between forward & up
dirs[i++] = (dirs[0]+dirs[5]).normalize();      //  9->Angle between forward & down
dirs[i++] = -dirs[6];                           // 10->Angle between behind & right
dirs[i++] = -dirs[7];                            // 11->Angle between behind & left
dirs[i++] = -dirs[8];                            // 12->Angle between behind & down
dirs[i++] = -dirs[9];                              // 13->Angle between behind & up
dirs[i++] = (dirs[0]+dirs[2]+dirs[4]).normalize();    // 14->AB forward & left & up
dirs[i++] = (dirs[0]+dirs[2]+dirs[5]).normalize();  // 15->AB forward & left & down
dirs[i++] = (dirs[0]+dirs[3]+dirs[4]).normalize();  // 16->AB forward & right & up.
dirs[i++] = (dirs[0]+dirs[3]+dirs[5]).normalize(); // 17->AB forward & right & down
dirs[i++] = -dirs[14];                      // 18->Angle between behind & left & up
dirs[i++] = -dirs[15];                    // 19->Angle between behind & left & down
dirs[i++] = -dirs[16];                     // 20->Angle between behind & right & up
dirs[i++] = -dirs[17];                   // 21->Angle between behind & right & down
// Now, add some random orientations... //
for( d=i; d<NUM_DIRECTIONS; d++ )             // 22 through NUM_DIRECTIONS->Random
   dirs[i++].randomNomalized();
// First set the breadth (number of directions) //
if( env->m_iBreadth <= 1 ) i = 4;
else if( env->m_iBreadth <= 2 ) i = 6;
else if( env->m_iBreadth <= 3 ) i = 14;
else if( env->m_iBreadth <= 4 ) i = 22;
else i = NUM_DIRECTIONS;
// Next set the depth (num reflection points to parse for each direction) //
if( env->m_iDepth <= 1 ) totalreflections = 1;
else if( env->m_iDepth <= 2 ) totalreflections = 10;
else if( env->m_iDepth <= 3 ) totalreflections = 20;
else if( env->m_iDepth <= 4 ) totalreflections = 40;
else if( env->m_iDepth <= 5 ) totalreflections = 80;
else if( env->m_iDepth <= 6 ) totalreflections = 160;
else totalreflections = MAX_REVERB_SEGMENTS - 2;
```

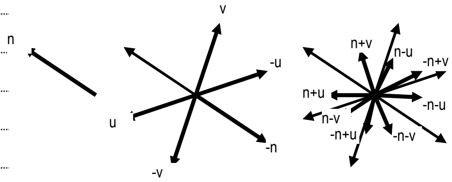


Table 4: Code for Generating Directions in Bouncing Reflections Algorithm

```
startTimeCheck();
setVariables();
calculateDirectPaths();
calculateDirections();
for dir...breadth {
        resetVariables();
        setInitialPoint();
        for ref...depth {
                if( checkListenerPlanesHits() ) addImpulseToData();
                for w...all_walls_in_room {
                        if( checkWallHit(w) ) {
                                addIntersectionPoint();
                                getNextDirectionFromIntersectionTest();
                        }
                }
        }
}
endTimeCheck();
```

Table 5: Bouncing Reflections Algorithm Pseudo-Code

While the section “Efficiency Validity” on page 104 investigates the actual running times for this algorithm, here we consider the theoretical intricacy of our model. Both the breadth and depth settings within the simulator dramatically affect the complexity of this algorithm. Since the directional determinations based off the breadth setting compute linearly and in exactly the same quantity for each run, we do not consider this in our analysis. The same goes for setting the number of reflections to parse via the depth setting. The true impact from this algorithm comes from how we use these variables. The algorithm contains two main loops. We traverse each path, via breadth, and within each path, we work out a maximum number of reflections, via depth. Within these nested loops, we currently check two intersections of planes against line segments and all wall intersections. Like the reflected paths algorithm, we presently only allow for a geometrically simple room with four walls. However, we must also consider the floors and ceilings as reflection points. Assuming we have a small, limited number of faces this gives an order of complexity bounded by $O(n^3)$.

Wall Dispersion Algorithm

Derived from the Bouncing Reflections algorithm, this method takes into account that sound does not always directly reflect off of walls (Schroder & Pohl, 2013), but rather some energy randomly disperses at each hit. When a sound wave encounters a barrier, much of its energy reflects geometrically as expected and described above. However, some power transmits into the medium while the remainder disperses in almost every possible direction. This typically derives from the fact that surfaces on a microscopic level are not completely smooth (Figure 10) relative to the wavelength of the sound. Statistically speaking, this dispersion looks practically random and therefore we can consider it uniform over a hemisphere radiating out from the wall (Kinsler, Frey, Coppens, & Sanders, 2000). For the purpose of the algorithm, we regard each surface as another sound source, starting at a lower volume. Obviously, this will geometrically increase the complexity of the computations if we do not consider further refinements and optimizations. We could additionally use the breadth complexity level to determine how many dispersion paths at each reflection point to pursue. The absorption coefficient of the material surface property at the reflection points could also influence or controlled the dispersal complexity.

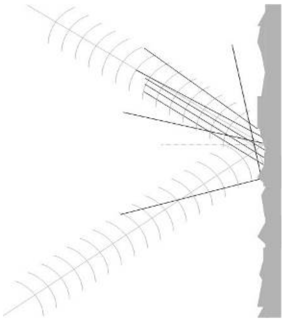


Figure 10: Dispersion Due to Microscopic Effects on Reflections

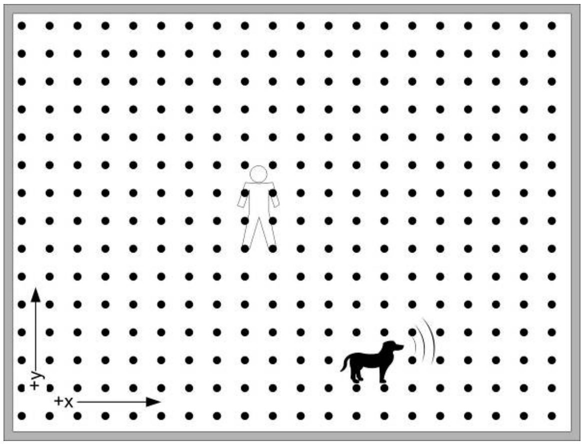


Figure 11: Division into Possible Locations

Due to the intuitive nature of this algorithm with respect to how sound transmits and reflects off surfaces it would seem that this algorithm would easily fall within the purview of this research. Extending the currently implemented Bouncing Reflections algorithm to include this approach would not overly complicate matters and should take relatively minor effort and time. However, for the initial experiment and analysis of the subsequent data, we decided to leave this approach for future development. With the limited subject pool and number of tests dictated by the experiment (see “Testing Details” on page 99), including another algorithm with more variables would significantly diminish from the statistical validity we desire.

Matrices of Impulses Algorithm

This algorithm would use a specified combination of the above routines to perform calculations at runtime. Ideally, it would accurately produce a matrix of matrices of sound impulse arrays from all possible sound source locations to any potential listener position. Because of the extremely computationally expensive nature of this algorithm, it would require a very complex structure and large amounts of data storage and memory. We cursorily address these issues below. The simulator must first divide the virtual environment into a set of evenly spaced points (Figure 11) based off user-specified granularity. Then, for every possible (sound source) location within the grid, we calculate the impulse response for every (listener) point in the grid. When it comes time to play the sound, it is simply a matter of a quick lookup of the current locations in the matrices and applying the pre-generated impulse response to reproduce the sound. For locations not exactly matching the grid, we might select or interpolate between nearby points and the subsequent impulse responses.

If the algorithm uses a granularity of g_x, g_y, and g_z, we must compute and store approximately $(g_x \times g_y \times g_z)^2$ impulse arrays. This is an enormous amount of data. Computing it every time the simulator initializes might take minutes, hours, or even days. We therefore propose caching the previously generated results and running calculations only when the geometry of the environment has changed.

In order to account for orientations of the sound source and listener, we must store the impulse in a modified form. Specifically, we need to know for each impulse that reaches the potential listener the distance or time traveled, the original orientation from the sound source, the arriving orientation, and any attenuation loss incurred from absorption. With these details, reconstructing the final impulse at run-time should require minimal effort.

To expand this algorithm so it can handle multiple rooms, we expect to employ solutions discussed in the previous sections for the basic algorithms and simply expand the grid to cover the entire structure. This same approach should also allow the merging of multiple structures. Clearly, this approach capstones any and all other algorithm developments. When we finish evaluations and refinements of our other works, we can pursue this technique in earnest.

Physics Algorithm

Some previous works (Stephenson, 2013) (Southern & Siltanen, 2013) (Raghuvanshi, Snyder, Mehra, Lin, & Govindaraju, 2010) have alluded to the potential of using physics-based wave equations to solve the 3D sound challenge. Though no solution has yet come forth using this approach, it has the potential as an effective, if not efficient solution. Problems involved with this track include converting complex generalized differential equations into practical run-time computer algorithms. We intend to explore this option in future work. Many recent articles concerning 3D acoustics in virtual environments have suggested some form of a hybrid approach to the problem. This generally entails using a geometric model for lower-ordered reflections and employing wave equations for higher-ordered reflections, or reverberations. The trend toward this combination presently dominates academic journals for this topic.

Stochastic Algorithm

Another (Stephenson, 2013) (Cowan & Kapralos, A GPU-Based Method to Approximate Acoustical Reflectivity, 2011) approach employs a statistical method. This solution, though extremely complex in nature, could possibly closest approximate our understanding of how sound actually propagates. We perceive sound microscopically and macroscopically, measuring the transmission of pressure variant waves. These waves formed by particles in spaces move in a general direction pushing neighboring particles in a wave-like manner. Since these particles always move in an unpredictable chaotic fashion, physicists often analyze the statistical nature of the movements. Our simulator could employ simplified mathematics to calculate the movement of sound on an atomic level. We leave this and the physics-based approaches for future consideration.

Design and Implementation

Implementation Overview

In order to explore and evaluate the algorithms described in this research, we developed an extensive virtual 3D environment that allows for quick and easy design of dynamic 3D virtual worlds, complete with structures containing rooms with walls and other objects. Avatars and sound sources have unencumbered movement throughout the space. The virtual environment application, or TDS Simulator (short for Three Dimensional Sound Simulator), can play sounds via any number of acoustic algorithms and even includes testing modes for subject experimentation (see "Testing Modes" on page 89). For further details about how the simulator runs and loads information, see the section "The TDS Simulator" on page 49.

When reproducing acoustic virtualizations, the application must first read and then interpret the geometrical information of the current environment, structures, and rooms. The simulator uses this same information to produce the 3D visualization via the OpenGL® programming library. The TDS Simulator then exploits this information to determine the acoustic nature (see "Early Echo Response & Reverberation" and Figure 34 on page 152 and "Objectives" on page 16) of the environment and feeds this data into the audio engine for playback. The audio engine then convolves (see "Convolution Algorithm" on page 77) the finite impulse response with the original monaural sound file and plays the generated sound reproduction. In the case of running off-the-shelf algorithms (such as Microsoft's DirectX DirectSound®), the simulator simply skips the intermediate steps and sends the sound file straight to the audio engine for appropriate reproduction. Otherwise, the simulator uses standard audio processing algorithms such as convolution and fast Fourier transform to produce the desired auralization. Figure 12 highlights an extremely simplified overview of the basic flow of data within the simulator and the relationship between the core components. Since we modularized the various workings of the system, we can easily maintain portability and compatibility on multiple platforms.

Algorithmic Goal

Our goal for this project is to generate the impulse response (Figure 5 on page 17) between the sound source and the listener for any set of locations in the entire room or environment, and do this efficiently. The response will consist of an array of echo volume losses that we can use to play back the sound. As mentioned previously, the sound can travel numerous paths, reflecting off of walls and other objects, finally reaching the listener at various times and strengths. After we calculate a set of these impulses, the chosen API (see

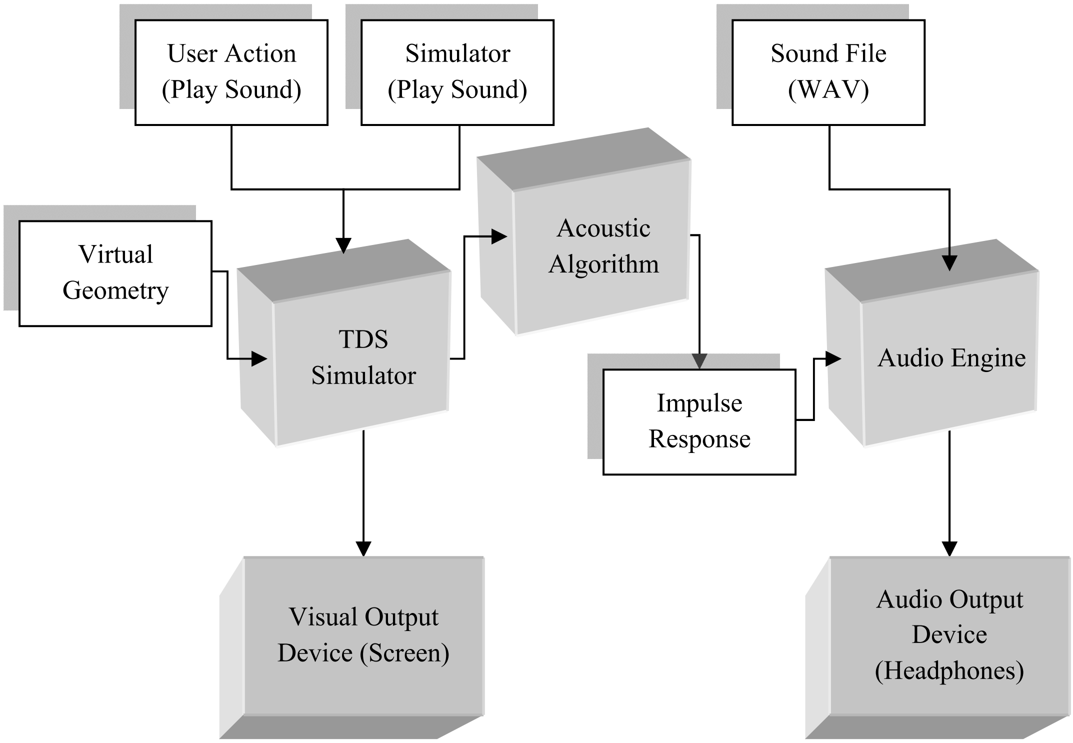


Figure 12: TDS Simulator Algorithm Pipeline

"Sound API's" on page 153) will replay the original sound multiple times while applying the appropriate attenuation and other desired effects based on the impulse response. Impulse data handling and audio processing can occur either by the hardware mixing multiple instances of the specific sound or convolving it at the software level (see "Convolution Algorithm" on page 77).

One question overrides all others: how do we calculate an impulse response efficiently and accurately? Inherent difficulties include the depth or order of reflections to compute, allowing for dynamic listener and sound source locations and orientations, multiple-roomed or complicated structures, dynamic changes in the virtual environment, object interference, open-spaced environments, material absorption and dispersion, refraction, and latency. We will address all of these challenges below, but it is worthwhile now to delve into latency and its impact.

Latency

We consider latency, or the amount of delay incurred due to the computations before sound begins playback, the primary determining evaluation of the correctness of an algorithm. Presently, offline applications (e.g. EASE [http://www.auralisation.com/], CATT Acoustic [http://www.catt.se/], and Odeon [http://www.odeon.dk/]) which developers did not design to run in real-time situations can analyze spaces such as concert halls and sound rooms for acoustic anomalies. The algorithms used in these programs, though considered comprehensive, run far too slow for real-time applications such as virtual environments. The human ear can tolerate up to a 150 ms delay (Wu, Duh, Ouhyoung, & Wu, 1997) from the visual or perceived initiation of a sound to the actual time that the user first hears it. Thus, any valid algorithm must have a maximum latency below this threshold. We demonstrated in the Brute Force algorithm above the impracticality of generating an impulse response in real-time using techniques standard in offline architectural applications without significant optimizations and considerations. Therefore, we must develop alternative solutions to this problem and any viable approach must fall within a low latency threshold, ideally running with something close to no latency. We will revisit latency in a formal setting with the analysis of the experiment results (see “Efficiency Validity” on page 104). Next, we will detail some of the data structures and algorithms common to almost any solution to this problem, considering their impact on efficiency and latency.

Object-Oriented Design

As mentioned previously, the TDS Simulator is an extensive application that manages data flow and information, 3D visual data and rendering, 3D audio pipeline, simulation and subject testing, and more. We chose to employ an object-oriented, hierarchical design to the simulator’s libraries. This decision allows for easy encapsulation, modularization, and portability. Written with Microsoft’s Visual Studio® 2008 C++ programming environment, the code consists of approximately 25,000 lines of code for the entire project, so we will only highlight basic design and functionality here. The following sections describe the workings of the simulator and a few of the more relevant objects designed for this application.

The TDS Simulator and Algorithms

Fundamentally, for the algorithms developed for this research, we treat sound as a spherical wave increasing in diameter from the source. We consider sound to radiate out equally in all directions from the sound source and use this to calculate the various orders of reflections. Assuming that we have the geometry of the scene readily available, these computations generally break down to simple trigonometry. Even when traversing through different rooms, the math, if properly considered, does not become overly complicated. We also allow the flexibility to compute only the depth of reflections that the processor can handle which eventually we will dynamically set, presumably to account for speed and processor load. Furthermore, we can set the breadth to limit computation complexity by modifying the number of directions, granularity, stereo vs. mono, cut-off level, or any aspect specific to the desired algorithm.

Currently, the simulator includes limited executions of the two algorithms: the Direct Paths algorithm and the Reflected Paths algorithm (see pages 31 and 34, respectively). Future development should include other algorithms such as the physics and stochastic models.

The TDS Simulator

The TDS Simulator (short for Three Dimensional Sound) runs as an OpenGL® virtual environment application. Figure 8, Figure 9, and Figure 13 show screenshots of the simulator in action. We developed the simulator to allow for easy specification of almost any hierarchical building or structure. These structures can contain numerous rooms, each with walls, doors, windows, and other objects in almost any conceivable geometry. Parsed during

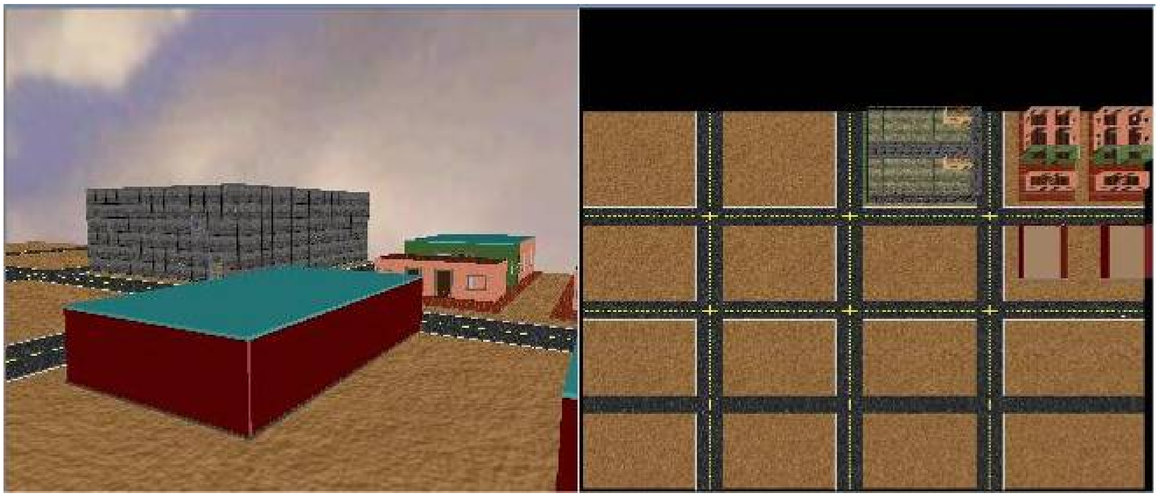


Figure 13: Screenshot of the TDS Application

the initialization of the application, we store the environment details, including structures, sound sources, and avatars in local text files. The hierarchical design of the environment affords quick and simple expandability and efficient access to the virtual landscape. For example, user-specified materials compose each surface. We have expanded these materials not only to include graphical information (e.g. color, texture, light reflection level, etc.), but also acoustic properties like absorption and dispersion. We make these properties available for any of the acoustic algorithms to utilize in the form of coefficients variables.

Both avatars and sound sources have complete autonomy and freedom of motion within the virtual world. As well, we permit multiple instances of either. The user sees the world through the eyes of any of the avatars and controls the locations, orientations, and playback of the various sound sources. Presently, scene geometry remains static for each time we start the simulator, but this limitation strictly resides with the simulator, and eventually we will not require it. Neither algorithm currently uses precompiled information based on the run-time environment geometry beyond common properties or calculations.

The simulator can run any number of algorithms to achieve the desired results: an impulse response of the sound (Figure 34). Each algorithm must generate the impulse response and the simulator instructs it to take care of mixing and playing the result. At playback of a sound source, the simulator, using the currently selected algorithm, performs the desired calculations on the original sound file and plays the generated result. Since Microsoft's DirectX DirectSound® API gives near low-level access to the sound mixing hardware, we have chosen it to handle the mixing and playback (see "DirectSound® Library" on page 83). However, as mentioned earlier, the algorithms remain completely independent of this exploited API, so future iterations could employ other audio engines.

We should note that while we do allow the movement of both sound sources and avatars, the environmental geometry remains fixed from the initialization of the simulator. Some algorithm optimizations used in this research require basic pre-computations based off the environmental data like plane locations and normal vectors. Future algorithms such as the Matrices of Impulses algorithm (see page 42), would certainly benefit from this assumption of precompiled values. However, we are certain we can overcome this assumption with further refinements to the algorithms and simulator. Furthermore, we currently only allow the relative speed between the avatar and sound source to remain far less than the speed of sound. Thus, we do not contend with the Doppler Effect or other advanced acoustical concepts. When a sound source or listener moves, the simulator simply reruns the algorithm with the new geometry.

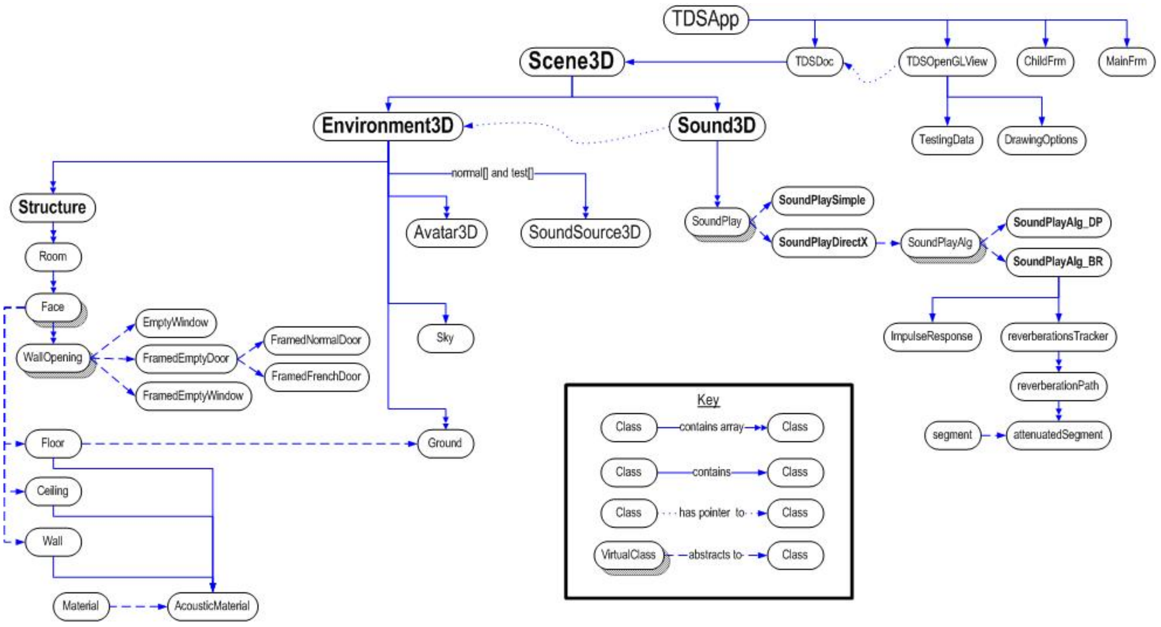


Figure 14: TDS Data Structure

Scene3D Data Structure

The Scene3D data structure entirely describes the complex organization of the environmental geometry. Used for the three-dimensional visualization as well as the acoustic realization fundamental to this research, this library of objects must allow sufficiently rapid storage, manipulation, and retrieval of virtual geographical information. Essential to the TDS Simulator, we require a data structure that accurately and efficiently stores the virtual geometry and design of the buildings and other objects of the environment. We need this data structure robust enough to handle almost any conceivable building layout while still allowing for adaptation to unforeseen elements and future developments. It must handle large sets of data with numerous rooms and structures while still offering efficient access to specific elements at a moment's notice by any part of the simulator. Furthermore, it should easily expand to numerous rooms and structures, possibly allowing for dynamic changes while the simulator runs. Clearly, the graphical API (in our case OpenGL®) must swiftly read this data to generate the visuals, but the design should generically allow for any choice of API on any platform. The same should hold for the sound API (in our case currently Microsoft's DirectX DirectSound® library). As we will describe in the sections below, the data structure will incorporate drawing and other routines specific to each object it contains, but these functions must allow for easy adaptation to alternative environments. Clearly, we will choose a specific programming language and environment as well as various API's, but the Scene3D design should follow standard, cross-platform programming techniques and conventions.

Environment3D Hierarchy

Central to the Scene3D library system, the Environment3D object stores all objects within the scene. Figure 14 illustrates the basic design flow of the objects within the simulator, including the `Environment3D` object and subsequent entities. Within the `Scene3D` object, we store a single instance of this data structure as well as the audio engine, `Sound3D`. Through this parent object, these two divisions of the simulator have some degree of access to each other via C++ pointers and friendly functions.

The `Environment3D` object contains an array of `Structure` objects, which in turn hold the rooms, walls, and all other components (see "Structure Objects" on page 57) of a building. This encompassing `Environment3D` object also stores the grounds and sky for the entire simulator. The sky simply consists of an enormous cloud-textured sphere bisected by the ground, or terrain objects.

Finally, the `Environment3D` object includes the `Avatar3D` and `SoundSource3D` object arrays. Both of these objects function in a similar manner except that one originates the sound while the other listens to it. They both can freely move through the environment and draw representations within the simulator. An avatar, however, also incorporates all camera functionality within the simulator. The `Avatar3D` objects have the responsibility of presenting everything the user sees on the screen based on the location and orientation of its eyes. On the other hand, the `SoundSource3D` objects do little more than draw and maintain the locations of the sound sources. The audio engine, `Sound3D` takes care of all sound processing and generation, as we will describe shortly.

Structure Data File

The TDS Simulator, when initialized, reads and parses a set of user-editable text data files that contain the design and layout of the virtual geometry of the environment, including the `Structure` object and all ensuing components. Initially, the simulator looks for the scene data file that lists all of the subsequent data files to load as well as the initial positions of each object. The scene file specifies all buildings, avatars, sound sources (including sound sources for the experiment), and the grounds and the sky universal to the environment. We have included an example of a scene file at the end of this document in Table 32 on page 163. The simulator, specifically the `Scene3D` object, parses this file creating appropriate objects that it instructs to load the necessary data. Each child object handles reading and storing information from the various data files. The simulator can use a data file multiple times to create separate objects of the same type.

The text data files follow an extremely simple specification. The `Avatar3D` and `SoundSource3D` objects exemplify this (see Table 31 on page 161) and require no further explanation. However, the `Structure` object and the associated data file have a much more complicated blueprint. Table 6 on page 57 lists a simplified example of a structure data file that illustrates various aspects of this multifaceted design. The simulator incorporates a common robust text file reading library that tokenizes data as it delves into it. This library ignores comments, denoted by anything following two forward slashes, as well as ignores white space. It also reads and converts text into appropriate storable data formats. We force data to follow a prearranged bracketed flow, embedding objects within objects.

When it encounters a structure data file, the simulator fist checks that the version number of the files matches up with what it expects. Next, it requires certain global variables such as units of measurements and geometrical offsets. It then will read any materials used by objects within the building as well as any ground plains that the structure rests upon. Finally, it loads all rooms contained within the construction. In Table 6, we have designed a structure with only one room, but clearly, it could manage many more. This room, however, contains a unique dividing wall that effectively separates it into two spaces. We included this only to illustrate the different manners of specifying faces.

When the simulator loads a `Room` object, it can load it via a one of two methods. The simplest technique assumes that the space follows a typical rectangular flow. Within the parsed data file, the user specifies the room dimensions and then labels the `Face` objects as either `StandardWall` or `ceiling` or `floor`. The library parses these and automatically initializes and locates the appropriate `Face` objects. No geometry needs setting for these standard objects with the exception of which wall to create (`negahead` or `negy` or `south`; `ahead` or `posy` or `north`; `negside` or `negx` or `west`; `side` or `posx` or `east`). Custom faces have the label wall and the user must specify dimensions, location, and orientation. This allows for easy design of standard rooms and buildings, while still accepting complicated models. The `splitWall` in Table 6 demonstrates this by subdividing the room with a large opening in the middle. Figure 15 is a screenshot of the structure described above.

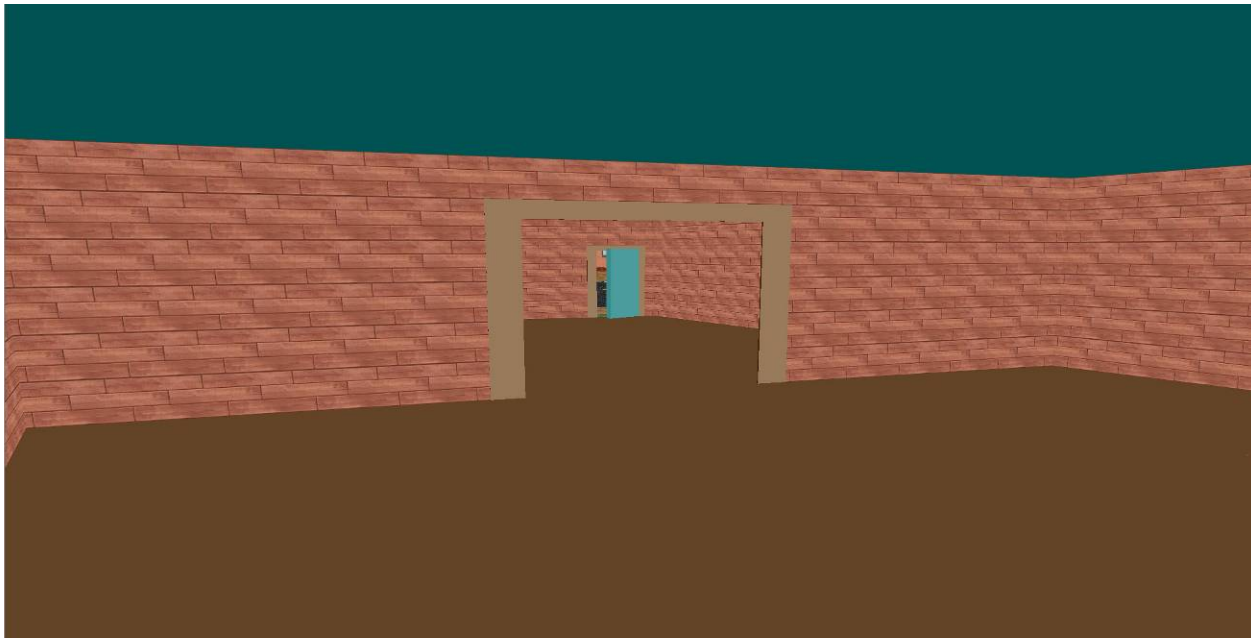


Figure 15: Structure 1.3 Data File Example: An Example House Structure

```
Structure 1.3
// examplehouse.txt,
// Created for TDS Simulator
// Measurements are in meters, feet, etc
   Units              inches
// The thickness of the walls
// and ceilings are counted or not
   Measurements       outside
// Move entire structure from (0,0,0)
   Offset             0.0 0.0 0.0
// Rotation of the whole structure
// around axis's (after translation)
   Rotation           0 0 0
NumMaterials          4
Material              Floor
{
   Color              BROWN
// Lower value makes perfect reflection
   Absorption         0.0005
   Dispersion         0.0001
}
Material              WallTextured
{
   File Textures\Wood5.tga
   RepresentedSize    20.0 20.0 20.0
   Absorption         0.10
   Dispersion         0.05
}
Material              Ceiling
{
   Color              CYAN
   Absorption         0.0005
   Dispersion         0.0020
}
Material              Grass
{
   File               Textures\Ground.tga
   RepresentedSize    1.0 1.0 0.1
   Absorption         0.50
   Dispersion         0.40
}
NumGrounds            1
Ground  Grass
{
   Thickness          0.5
   Dimensions         -30 330 -30 590
   Offset             -1.0
   Material           Grass
}
NumRooms              1
Room MainRoom
{
   Offset             0.0 0.0 0.0
   Dimensions         300.0 550.0 120.0
   Rotation           0 0 0
   NumFaces           7
   Face StandardWall
   {
      Name            farWall
      Thickness       3.0
      Side            ahead
      Material        WallTextured
      NumWallOpenings 0
   }
   Face               StandardWall
   {
      Name            rightWall
      Thickness       3.0
      Side            side
      Material        WallTextured
      NumWallOpenings 0
   }
   Face               StandardWall
   {
      Name            leftWall
      Thickness       3.0
      Side            negside
      Material        WallTextured
      NumWallOpenings 0
   }
   Face               StandardWall
   {
      Name            entranceWall
      Thickness       3.0
      Side            negahead
      Material        WallTextured
      NumWallOpenings 1
      WallOpening     FramedNormalDoor
```

```
    {
      Type          FramedNormalDoor
      Offset        8.5 0.0
      Dimensions    36.0 80.0
      Material      Ceiling
      TrimMaterial  Floor
    }
  }
  Face              ceiling
  {
    Name            theCeiling
    Thickness       3.0
    Material        Ceiling
    NumWallOpenings 0
  }
  Face              floor
  {
    Name            theFloor
    Thickness       3.0
    Material        Floor
    NumWallOpenings 0

  }
  Face              Wall
  {
    Name            splitWall
    Thickness       0.5
    Height          0.0 120.0
    Length          0.0 300.0
    Rotation        0
    Offset          0.0 300.0 0.0
    Material        WallTextured
    NumWallOpenings 1
    WallOpening     FramedEmptyDoor
    {
      Type          FramedEmptyDoor
      Offset        105.0 0.0
      Dimensions    90.0 100.0
      Material      Ceiling
      TrimMaterial  Floor
    }
  }
}
```

Table 6: Structure 1.3 Data File Example: An Example House Structure

Structure Objects

Contained as an array within the `Enviroment3D` object, the `Structure` objects abstract buildings. Each building can include one or more rooms and each room typically has six faces. As with all `Enviroment3D` objects, this object maintains its location and orientation, which the graphics engine uses as a starting point for the rooms within the `Structure`. However, the `Structure` object does not technically draw anything. Rather, it propagates drawing commands to the `Room` objects that it possesses. We should note that the `Structure` object stores all materials (see “Acoustic Materials” on page 59) used by the objects within its scope. When the simulator initializes, it reads and loads these materials and sends them as pointers to subsequent objects.

Room Objects

`Room` objects only exist as an array within the `Structure` objects. Certainly, a `Structure` can contain just one `Room` object, but this does not occur to frequently. Typically, a `Structure` will load multiple rooms, side-by-side. Just like the `Structure` objects, `Room` objects do not actually draw anything through the graphics library. Instead, they pass on the drawing commands to all `Face` objects contained within the data structure. A `Room` object can contain a large number of `Face` objects, but typically only has six faces—four walls, a ceiling, and the floor. However, a `Room` object has the ability to store any number of faces, in about any conceivable configuration.

Faces Objects

Using a common C++ programming convention, we do not formally create instances of the `Face` objects at any time. Rather, this virtual class will abstract to a `Wall`, `Ceiling`, or `Floor` object. Unlike its parent objects `Room` and `Structure`, children of the `Face` object include formal drawing routines. `Floor` and `Ceiling` objects do not contain any openings and have extremely basic drawing routines. On the other hand, a `Wall` object must progressively draw itself from one side to the other, leaving space for the various openings.

Wall Openings Objects

Like the `Face` objects, a `WallOpening` object exists only as virtual class that must substantiate in the form of an `EmptyWindow`, `FramedEmptyWindow`, `FramedEmptyDoor`, `FramedNormalDoor`, or `FramedFrenchDoor` object. See Figure 14 on page 51 for the hierarchal design of these objects. With the exception of the `EmptyWindow` object, these classes include functions for drawing appropriate representations. An algorithm has the option of ignoring reflections that bounce off a face but fall within these possibly empty intervals. The `WallOpening` objects can even influence this decision based on its physical nature or current state. For instance, an empty window should allow no reflections, while a closed door might.

Acoustic Materials

Each substantial object contained within `Structure` uses one or more acoustic materials. The `AcousticMaterial` object derives from the `Material` object that integrates drawing of either simple flat colors or visually stimulating textured patterns. The difference between the two is that the `AcousticMaterial` object stores the two variables `absorption_coefficient` and `dispersion_coefficient` (see "Absorption & Dispersion" on page 146). The acoustic algorithms can include these variables in their calculations.

<u>**Sound3D Data Structure**</u>

We isolated the audio engine and all acoustic manipulation routines into `Sound3D` data structure. The TDS application need only create one instance of this object. We send all audio events to the `Sound3D` object for it to process and manage. Though we do store some information (e.g. breadth and depth settings) in the `Enviroment3D` object to allow for global access, all manipulations and modifications occur through this library. Most importantly, we store a virtual array of `AcousticsAlgorithm` objects through which the `Sound3D` object can instantiate objects in the form of any of the current possible algorithm classes. These algorithms each have the responsibility to load and play the sound when prompted in a manner appropriate to the algorithm. We should note that each algorithm may independently employ its own audio engine (MCI, DirectX, OpenAL) to reproduce the sound.

AcousticsAlgorithm Hierarchy

Nested within the `Sound3D` hierarchy, the design and structure of the acoustic algorithms code begs for further explanation. Fundamental to this research, this construction integrates intensive and important core programming in the form of the various acoustic algorithms. Presently, the algorithms include `SoundPlaySimple` (or no algorithm), `SoundPlayDirectX`, `SoundPlayAlg_DP` (the Direct Paths algorithm), and `SoundPlayAlg_BR` (the Bouncing Reflections algorithm). See Figure 14 on page 51 for the hierarchal design of these objects. With minimal effort, we can quickly expand the list of available algorithms by deriving new objects from the parent virtual `SoundPlay` class. The only complication lies in the formal implementation of the algorithm and not the integration within the simulator. The virtual `SoundPlay` class serves mostly as a focal point for the subclasses and includes little functionality other than checking if sound files exist and managing generally useful variables like the type of algorithm of the child class and the algorithm's current playback state. All capabilities of the algorithms reside in the derived subclasses.

SoundPlaySimple Algorithm Object

The `SoundPlaySimple` algorithm object does nothing more than playback the sound file. It considers no 3D geometry or acoustic properties at all. Rather, it merely exploits Microsoft Windows® standard Media Control Interface, or MCI, routines to read and playback an audio file. It plays the sound straight from the indicated sound file, not even bothering to preload any information. Understandably, this "algorithm" runs in real-time without any delay or lag. We call this and include this as an algorithm for baseline comparisons and controls within the experiment. Cleary, in any experiment, all proper 3D sound algorithms must perform significantly better than the Simple algorithm with respect to 3D virtualization and localization.

SoundPlayDirectX Algorithm Object

Parent to the two developed algorithms in this research, the `SoundPlayDirectX` algorithm object utilizes Microsoft's DirectX DirectSound® library for reading and playing sounds. This library offers us somewhat low-level access to the hardware while still maintaining a higher level of programming abstraction (see "DirectSound® Library" on page 83). Both currently implemented research algorithms, Direct Paths and Bouncing Reflections, inherit from this class in order to take advantage of the ease of efficient access to the audio hardware. The DirectX algorithm itself executes the 3D acoustic algorithm in Microsoft's DirectX DirectSound® library. This algorithm decidedly compares with the Direct Paths algorithm in that it only considers the relative distance and orientations between the sound source and the listener and ignores the room geometry entirely. In fact, the version used for this research (DirectX® 9.0) includes major undocumented flaws and bugs, which required significant effort to overcome. For instance, the library cannot handle listener objects moving through the environment. Rather, we must mathematically reposition the listener to the origin and rotate everything around it. Despite this and other bugs, unquestionably and depressingly so, this library still sits at the top of available API's for sound processing and 3D auralization. Simply put, no (viable) alternative presently exists.

The DirectX algorithm loads and stores the sound files into memory. It can process almost any format, but currently we only allow 16-bit monaural PCM WAVE files for the sake of simplicity. We certainly could include capability for reading 8-bit sound files and convert them to 16-bit for processing, but the resolution and quality of higher fidelity original sound files should prove worthwhile. However, we must require single channel files to begin with since we wish to translate them to stereo signals using convolution and the calculated impulse response for our algorithms. This logical constraint holds as well for the DirectSound® library, which can only apply 3D effects on monaural data.

We should note that the DirectX algorithm and thus succeeding objects retain the previous locations and orientations of the avatar and sound source objects. When the `Sound3D` object calls the algorithm to run, the library quickly compares these variables against the current environment and can opt to use old, previously calculated results. Not presently implemented, we could set a threshold for change of orientations and locations in which to force the algorithm to recalculate (see "Orientation and Location Thresholds" on page 141).

SoundPlayAlg Object

The DirectX algorithm includes extra functionality not required by the DirectSound® 3D algorithm. It can store sound data in a number of extra secondary and mixing buffers for processing. The DirectX algorithm can play directly from the secondary buffer while it allows unregulated access to the mixing buffers. Through the virtual class `SoundPlayAlg` (child object of the `SoundPlayDirectX` algorithm object), our algorithms use these buffers and can send them back to the DirectX® engine for sound generation. In this aspect, the algorithms we developed do depend on a specific API library, but at some point, we cannot avoid this anchor. Therefore, we have decided to work with the already reliant DirectX® object and bridge the proper algorithms through this intermediary object. All derived algorithms in this research (currently only two) inherit directly from this virtual class to obtain this functionality.

The `SoundPlayAlg` object uses the parent DirectX algorithm to parse and load the sound file, then reads and copies the data into accessible buffers. It stores the sound data as arrays of individual samples (see "Sound Sample Storage" on page 75). This object also creates left and right channel buffers to mix the impulse responses and the original data into for the final product of the algorithm. Finally, when it loads a sound file, it also normalizes the original sound signal globally so that convolving it will not produce excessively loud samples or popping noises.

SoundPlayDirectPaths Algorithm Object

Implementing the Direct Paths algorithm (page 31), the `SoundPlayAlg_DP` object contains nothing of considerable note other than the algorithm. This algorithm is analogous to the DirectX algorithm in that it only takes into account the orientations and distance between the sound source and the listener objects. It does not bounce the sound off the walls or otherwise consider the environment. We intend to extend this algorithm to account for the walls possibly even consider a few orders of direct reflections like the Reflected Paths algorithm on page 34, but we leave this for future work (see "Direct Paths Algorithm Expansion" on page 140). For now, this object simply runs the algorithm described in the pseudo code in Table 1 on page 32.

SoundPlayBouncingReflections Algorithm Object

Significantly more complicated than the previous algorithm object, the `SoundPlayAlg_BR` object runs the algorithm represented in the pseudo code in Table 5 on page 40. Since this algorithm must track numerous reflection paths in addition to the zero-order direct path from the sound source to the listener, we require two additional complex data structures: the `ImpulseResponse` object (see "Impulse Response Data Structure" on page 66) and the `reverberationsTracker` object (see "Reverberations Data Structure" on page 64). We store all paths emanating from the sound source in the `reverberationsTracker` object, while the `ImpulseResponse` object only maintains the times and volume levels of the paths that actually hit the listener's ears. The algorithm stores both of these data structures as single instances and keeps them as separate, external objects to allow other potential algorithms to include them.

Furthermore, in order to merge the generated finite impulse responses with the original sound signal, the algorithm must perform Fourier transforms and therefore we link it to the third party FFTW libraries (see "FFTW Library" on page 84). Linking and utilizing these libraries necessitates storage and manipulation of library specific variables such as "plans" and arrays of complex numbers with imaginary components. With one exception (wisdoms), this algorithm object and the embedded impulse response object manage all of the FFT functionality. We will describe, in detail, the incorporation of the FFTW libraries below.

Reverberations Data Structure

Though only currently utilized by one algorithm, the `reverberationsTracker` object has the flexibility to enhance the implementation of many algorithms and applications.

This object employs a hierarchal, object-oriented design that allows the program to track numerous acoustic paths of varying lengths. The `reverberationsTracker` object contains and array of `reverberationPath` objects each of which in turn manages an array of `attenuatedSegment` objects. The `attenuatedSegment` objects inherit from the parent segment object, which simply stores the start-point and end-point of a segment along with the calculated length. The `attenuatedSegment` object also preserves the strength, timestamp, and reference distances for the beginning and ending of the segment as well as variables for animation. See Figure 14 on page 51 for an illustration of the hierarchal relationship of these objects.

Consider a large number of ping-pong balls exploding out in all directions from the sound source. The Bouncing Reflections algorithm must determine and set these initial directions (see "Table 4" on page 39) of the ping-pong balls and has the responsibility of calculating all subsequent reflection directions, or bounce paths. It then enters each ball's path into the tracker by adding more end-points. Initially, the application makes a call to the `addPoint` function, sending the start-point and setting the appropriate path index number. It then calls this function again with the next point and the tracker establishes a line, or segment. Each subsequent call to this function adds another segment that begins where the previous one ended. We can include multiple ball paths by setting the path index variable as appropriate. So far, though, we have described few aspects of any acoustical nature in this data structure.

Before we add any paths, we must set two important acoustic variables. The `initialReferenceDistance` and `initialReferenceStrength` help determine the acoustic properties, specifically the attenuation, of the set of paths. When adding a point, we send the attenuation coefficient variable, calculatedCoefficient—defined as the volume loss for the sound due to its bouncing off a surface. Currently, we compute this value by simply adding the two variables absorption_coefficient and dispersion_coefficient from the acoustic material of the object. The tracker then multiplies the inverse of this value to determine the starting volume level of the next path. We reconsider this simplification for future expansion (see "Acoustic Assumptions" on page 140). See Table 7 and Figure 16 for an example use and breakdown of this variable and attenuation calculations.

The reverberationsTracker data structure includes a complex and powerful drawing routine that uses the C++ built in <time.h> library to animate the object. The SoundPlayAlg_BR object sends out many spheres, or ping-pong balls, emanating from the sound source, all traveling the same speed. Some paths will certainly end earlier than other paths, due to the extra volume loss from wall reflections and collisions with environmental objects. A path that reaches the active listener automatically ends and the reverberationsTracker highlights it with a different color. Finally, the tracker represents the strength of the signal by proportionally decreasing the size of the traveling spheres as they drop in volume. Through the simulator's powerful interface, the user has the option of showing the full animation or even just separate parts like the spheres, lines, highlights, and/or bounce points.

Impulse Response Data Structure

Like the reverberationsTracker object, we embed the ImpulseResponse object in the Bouncing Reflections algorithm. We can employ this data structure in any future algorithm due to its portable and utilitarian nature. This data structure links to the

FFTW libraries (see “FFTW Library” on page 84) in order to initialize and store data. In fact, almost all interaction, including formal calls to run the discrete Fourier transform functions ensues through this data structure. This logically flows from the natural relationship of the impulse response and Fourier analysis. However, we maintain portability to other platforms and libraries by noting that the third-party FFTW libraries already run on almost any system.

This data structure has a number of important arrays of data. The left and right impulse response arrays contain the strength values of the all hits sent to the reverberationsTracker object. Similarly, the left and right impulse time stamp arrays store the time of each processed hit. We need not make calls to the addStereoImpulse function in the order the hits transpire temporally. Rather, we allow the algorithms to parse an entire path before proceeding to the next one and call the hit function when required. Most importantly, though, the reverberationsTracker object shelves two arrays of the fftw_complex variables, one for each ear. This variable type, which the FFTW library defines simply as a “struct” of two floating-point variables, abstracts the real and imaginary components of complex numbers. The ImpulseResponse object stores the reorganized response data into the real component of these arrays in order to facilitate speedy convolution with the sound source by the algorithm. After the algorithm sends all of the appropriate data to the ImpulseResponse object, it calls the function postCalculate which simply sorts the previously sent impulse data in order of time and puts the results into these complex number arrays for use in the convolution process.

Attenuation Algorithm

Central to the algorithms in this research, the simulator includes procedures for determining the attenuation of sound propagating through the environment (see

“Attenuation” on page 147 for extended definitions and explanations). We consider only the most common environmental situations and constrain sound to travel at 344 meters per second. Certainly, factors such as air temperature, humidity, wind, and altitude can affect sound speed, but since the simulator currently does not allow changes to these controls, our algorithm does not account for them. We also simplify matters by embracing the approximation of the inverse-square law that states that sound level loses about 6dB for every doubling of distance from the source. This approximately means that the perceived volume level drops by half each time the distance doubles. Furthermore, we do not consider the effect of non-uniform volume loss at different frequencies that can occur primarily due to humidity levels in the atmosphere. These assumptions allow us to design an extremely efficient library for processing acoustics. Implementation of some of these simplifications could significantly influence the effectiveness of our methods. However, we opted to focus on developing algorithms comparable with industry standards. However, the simulator requires none of these assumptions and we leave them for future expansion and analysis (see “Acoustic Assumptions” on page 140).

Together, Table 7 and Figure 16 thoroughly illustrate the implementation of attenuation in our project. The `attenuatedSegment` object (part of `reverberationsTracker`) calls the attenuation function frequently during the course of each path. After the simulator determines the length of any segment, it immediately calculates the attenuation for that segment. These computations occur at each intersection point, whether the path bounces off a wall or hits the listener. Attenuation effect compounds as sound propagates further through the environment, so we calculate it on the fly with each path. Since we know the approximate volume level of a path at any given point, we can easily apply a cutoff threshold (the depth of the algorithm) to force a path to end prematurely when it becomes irrelevant due to it becoming too quiet. The `reverberationsTracker` data structure tracks the volume level loss in terms of strength of the signal. At full strength, or 1.0, the sound has not decreased in volume. As sound traverses space and loses volume, the strength variable decreases accordingly to zero. When appropriate, the impulse response data structure stores this variable and then the algorithm can use it to convolve the impulse response with the original sound data.

In effect, we consider two conditions that influence the sound path with respect to attenuation. First, we use the inverse-square law to determine volume loss. This law, common to physics and basic acoustics, states that as the distance doubles, the sound signal loses approximately half of its perceived volume (see “Attenuation” on page 147). In order to utilize this axiom, we must first know the initial distance traveled, or reference distance. Since each sound file has an approximate or average initial volume level, we must assume that all sounds playing without any acoustic manipulations start at a specific distance from the listener. We can call this length the “base reference distance” and we arbitrarily set it to 2.0 meters. This assumption holds for all sound files globally, regardless of the perceived initial volume level of the original data, as we must have some starting point to upon which to base our calculations. When we establish a set of reverberation paths, the acoustic algorithm must first examine the direct or zero-order path from the sound source to the listener. The algorithm has the responsibility of calculating the distance from the source to each ear of the listener. It sets the shortest distance as the initial reference distance (D^{IR} in Figure 16). With the initial reference distance, the base reference distance, and the strength of the signal from the sound source to the base reference distance (1.0, or full strength), the algorithm can send these factors to the attenuation function to find the initial reference strength level (S^{IR} in Figure 16). The algorithm then has a starting point to pursue all other attenuation calls. Table 7 and Figure 16 give a clear example of this confusing mathematical process.

The second aspect of attenuation to consider, reflection loss, occurs only as a path bounces off a wall. When a path encounters a wall, the `reverberationsTracker` data structure first uses the general attenuation function to find the strength before the hit. The path will lose a certain amount of strength from the interaction—some due to absorption and some from dispersion (see “Absorption & Dispersion” on page 146). The `calculatedCoefficient` variable combines the two acoustic properties `absorption_coefficient` and `dispersion_coefficient` for each material by simply adding them together (see “Acoustic Assumptions” on page 140). The tracker then calculates the resulting strength (S^2 in Figure 16) by multiplying the pre-reflection strength (S^1) with the inverse of the calculated coefficient variable. Again, Table 7 and Figure 16 give a clear example of this process. The combination of these two aspects of attenuation creates an effective and efficient approach to 3D acoustics.

The simulator does not have an object directly associated with attenuation. Rather, we chose to employ a globally available function to perform these calculations. We store most of these functions, along with routines and data structures for convolution and sound file storage in a general global library called `SoundAlgorithms`.

Attenuation Pseudo-Code:

```
ATTENUATION = 0.5             // The amount of volume loss for doubling of distance
atten(distance, startStrength, referenceDistance) {
   if( distance <= referenceDistance ) return startStrength;
   real endStrength = startStrength;
   // Don't forget to travel the referenceDistance first!!!
   distanceTraveled = referenceDistance;
   i = 0;
   while(true) {
      // distanceTraveled=distanceTraveled+(referenceDistance*2^i)
      pow2 = referenceDistance * pow(2.0, i);
      distanceTraveled += pow2;
      if( distanceTraveled > distance ) break;
      endStrength *= ATTENUATION;
      i++;
   }
   real remainder = distance - (distanceTraveled - pow2);
   endStrength *= 1.0 - (ATTENUATION * (remainder / pow2));
} // The line above is an approximation for efficiency!
```

To calculate S^{IR}:

```
Given: distance=4.0 [=D^IR], startStrength=1.0 [=S^0], referenceDistance=2.0
After traveling the first referenceDistance:
   distanceTraveled=2.0, endStrength=startStrength=1.0
After traveling referenceDistance again:
   distanceTraveled=4.0, endStrength=startStrength*ATTENUATION=0.5
After traveling referenceDistance again:
   distanceTraveled=8.0, endStrength=startStrength*ATTENUATION*ATTENUATION=0.25
```

To calculate S^{1}:

```
Given: distance=6.0 [=D^1], startStrength=0.5 [=S^IR], referenceDistance=4.0 [=D^IR]
After traveling the first referenceDistance:
   distanceTraveled=4.0, endStrength=startStrength=0.5
After traveling remainder=2.0:
   distanceTraveled=6.0,
   endStrength=startStrength*(1-(ATTENUATION*remainder/distanceTraveled))=0.416̅
After traveling referenceDistance again:
   distanceTraveled=8.0, endStrength=startStrength*ATTENUATION=0.25
After traveling referenceDistance again:
   distanceTraveled=16.0, endStrength=startStrength*ATTENUATION*ATTENUATION=0.125
```

To calculate S^{2}:

```
Given: startStrength=0.416̅ [=S^1], referenceDistance=4.0 [=D^IR],
   absorption_coefficient=0.006, dispersion_coefficient=0.004
calculatedCoefficient=absorption_coefficient+dispersion_coefficient=0.01
endStrength=startStrength*(1.0-calculatedCoefficient)=0.4125
```

Table 7: Calculating Attenuation Pseudo-Code and Example Calculations

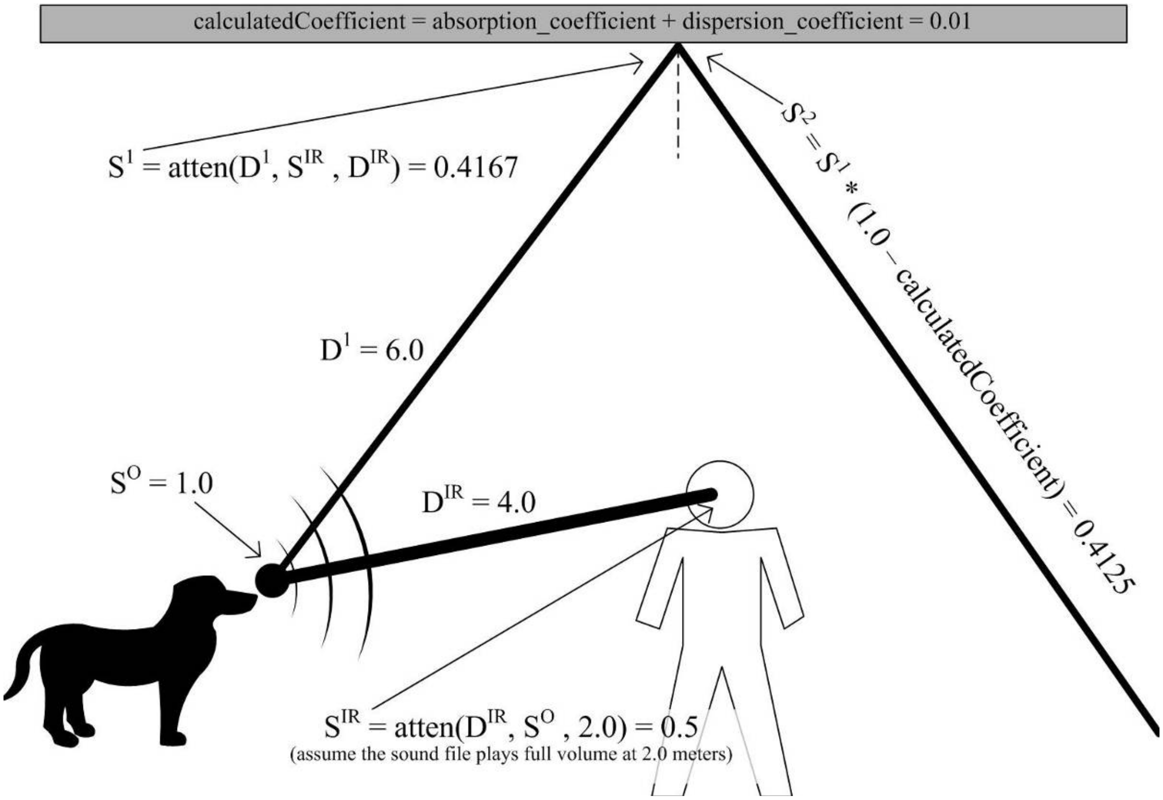

calculatedCoefficient = absorption_coefficient + dispersion_coefficient = 0.01
S^1 = atten(D^1, S^{IR} , D^{IR}) = 0.4167
$S^2 = S^1$ * (1.0 – calculatedCoefficient) = 0.4125
D^1 = 6.0
S^O = 1.0
D^{IR} = 4.0
S^{IR} = atten(D^{IR}, S^O , 2.0) = 0.5
(assume the sound file plays full volume at 2.0 meters)

Figure 16: Calculating Attenuation

Sound Storage

We must consider two aspects with respect to sound data: permanent and temporary storage. Digital sound typically begins as a stable file located on an accessible portion of a hard drive. It generally has one of a number of standard formats applied to it. An application, in our case the TDS Simulator, then reads and interprets this audio data and consigns it into temporary memory storage. Since we have little control over standards involving the persistent sound file storage, we will focus mainly on transient sound data stored in computer memory. First, though, we must cover some of the basics of local storage.

Sound File Storage

In order to keep the simulator generic and universally portable, it must conform to industry standards when reading sound data stored locally on a hard drive or other media. However, we do make certain assumptions about the format of the original permanent sound data in order to facilitate simplicity. First, the simulator only accepts WAVE data stored as PCM files. Developed and standardized by Microsoft™ in the early 1990's, the WAVE format (denoted by the ".wav" extension) describes a subset of the RIFF (Resource Interchange File Format) specification that typically contains sound data in a bitstream format [https://ccrma.stanford.edu/courses/422/projects/WaveFormat/]. We further require the bitstream data to follow a pulse-code modulation (PCM) form and only consider 16-bit, monaural data. We certainly could enhance the simulator to parse 8-bit data files; however, the loss in fidelity could have a dramatic impact on the results. Presently, this research does not explore this assumption, but logic dictates that starting with higher quality data will provide better results.

As well, the simulator cannot handle stereo sound files since it must eventually generate its own stereoscopic rendering of the original data. It uses the basic, signal-channel sound and eventually convolves it to produce a stereo effect (see "Convolution Algorithm" on page 77) so ability to read stereo sound files becomes irrelevant. This again follows simple logic and, in fact, industry standard 3D sound API's also make this assumption. The simulator could read only one channel of a stereo sound or in some manner merge two channels together, but this would defeat the purpose of generating a realistic 3D sound.

Finally, the simulator only reads WAVE PCM files. Files of this format best represent sampled analog signals. Furthermore, we do not read streamed data or allow losslessly and compressed designs like MP3 sound files. We must assume that we have the entire data at our disposal for convolution, so we cannot accept streamed or similarly formatted sound data. In order to execute the convolution algorithms, the simulator absolutely must know the maximum size of the signal data. As well, for the sake of elegance and simplicity the simulator only reads WAV files. Certainly, a digital sound can reside in any number of formats. The most popular format, MP3 files, countenances a losslessly storage model. This means that each time a user saves a file in this format, the data changes and actually effectively loses some aspect of the original source. In our research, technically, this would not directly affect the results, as we do not resave the generated stereo sounds. However, the simulator would have to convert the data into samples for processing which adds another level of undesired complications. As well, the MP3 format uses compression algorithms to read and store the data. Though this obstacle is not insurmountable, we can find no reason to clutter the simulator with this capability.

Instead, the simulator uses standard DirectX® routines based off sample libraries to read and process the stored WAVE files. This procedure breaks down to two complicated elements: reading the header and reading the data. Fortunately, Microsoft® has standardized and surprisingly simplified the reading of the header data. The simulator encapsulates all of the necessary code for this in the `WaveFile` object which uses the Windows MMIO functions for parsing the data [http://msdn.microsoft.com/en-us/library/windows/desktop/ee418775(v=vs.85).aspx]:

> WAV files are in the Resource Interchange File Format (RIFF), which consists of a variable number of named chunks containing either header information (for example, the format of sound samples) or data (the samples themselves). The Win32 API supplies functions for opening and closing RIFF files, seeking to chunks, and so on. The names of these functions all start with "mmio”.

Though most of this follows standard conventions, the `WaveFile` object does check the “`AudioFormat`” portion of the “`fmt `” of the header to force only PCM data.

The most difficult part in reading a WAVE PCM sound file lies in parsing the actual data. For incomprehensible or possibly just archaic reasons, the WAVE format specifies that the data reside as little-endian bytes. Furthermore, 8-bit samples start as single unsigned bytes from 0 to 255. However, we read 16-bit samples as pairs of bytes, 2's-complement signed integers, ranging from -32768 to 32767. See Table 8 for the way the simulator handles these differing data types. Our parsing algorithm must convert the two-byte little-endian 16-bit data to something useful to the simulator (see “Sound Sample Storage” on page 75). Fortunately, the processor can ignore complexity arising from stereo channels, but if required, we note that samples simply alternate back and form from right to left in the data.

Sound Sample Storage

Before examining how the simulator stores sound data in memory, we must first further consider the details of the data stored in the original sound file. By convention, the data in most WAVE files exists as pulse-code modulation (PCM) samples. This design corresponds appropriately to the manner in which we perceive and create sound. Consider a speaker reproducing digitized sound. The speaker has a configuration of magnets and coils that upon receiving an electrical impulse moves a membrane a subtle distance. This membrane movement pushes air molecules, causing pressure differentials, and thus perceivable sound. The amount of the distance moved directly corresponds to power or strength of the electrical impulse as well as the volume of the perceived sound. Running the process in reverse, using the membrane to detect small movements in the air, the system can record transmitted sounds. A PCM sound file simply stores these electrical impulses as discrete digitized samples. Clearly, this explanation oversimplifies a complicated process and does not address issues such as fidelity, aliasing, quantization, and sampling rates (Nyquist theorem). However, for the purposes of this research, we can assume we have access to properly sampled sound files that already factor these details. Individual samples in the sound file correspond to separate impulses for a speaker. We consider this data as stored in the time domain since playing the sound file means reading the data in a linearly temporal manner.

```
typedef signed char samp8;        // -128 to 127 (8b samples originally
                                  // stored unsigned bytes)
 typedef signed short samp16;     // -32768 to 32767 (16b samples stored
                                  // as 2's-complement signed integers)
typedef unsigned char sampData;   // 0 to 255
typedef float sampScaled;         // -1.0 to 1.0
void sample16ToReversedString(samp16 num, sampData &data1, sampData &data2) {
   data2 = (sampData)(num/256);   // data2 is the high byte, data1 is the low byte
   data1 = (sampData)(num-(256*data1));
   if( (num <= 0 && (data2 != 0 || data1 != 0)) && (data1 != 0) ) data2 -= 1;
}
samp16 reversedStringToSample16(sampData data1, sampData data2) {
   samp16 result = 0;
   int tmp;
   bool neg = false;
   tmp = (int)data2;
   if( tmp >= 128 ) {
     tmp -= 128;
     neg = true;
   }
   result = tmp*256.0;
   tmp = (int)data1;
   result += tmp;
   if( neg ) result -= 32768;
}
```

Table 8: Sound Storage Data Types and Conversion Functions

The WAVE sound file uses a complicated design to store audio information. Data points in the WAVE file start out as unsigned character bytes (ranging from 0 to 255), which the simulator reads in as the data type `sampData`. A sample consists of one or two of these unformatted bytes. Depending on the bit-rate, the `SoundPlayAlg` object then converts this data to standard, meaningful information using the functions listed in Table 8. In memory, a sample can exist as either a signed byte ranging from -127 to 128 (8-bit) or a signed integer from -32768 to 32767 (16-bit). The simulator must convert these variations to something standard and useful, in this case the defined data type `sampScaled`. Variables of this type can range as floating-point numbers from -1.0 to 1.0.

Once in the `sampScaled` format, the `SoundPlayAlg` object stores the sample data into arrays for later manipulation. It retains a copy of the original sample data in a non-modifiable array and allocates memory for two mixing arrays, one for each ear. These arrays must each have at least twice the memory allocated as the original data in order to facilitate convolution.

Convolution Algorithm

As mentioned previously, the sound API ideally would have the option to apply the computed impulse response on the original sound file through accelerated hardware components of the sound card. Indeed this has an analogous case in 3D graphics. In the visual realm, the API effectively sends the geometrical information to the 3D graphics card, which has specialized processors to handle and manipulate the data and complex calculations. While some sound cards do boast some 3D acceleration, the reality of current technology dictates only accelerated HRTF processing (see “Head Related Transfer Function (HRTF)” on page 157). For the time being, the simulator must convolve the sounds at the software level (see Table 9 and Equation 2). The methods used in this step are common sound processing techniques, implemented throughout the computer music field for decades.

After the system processes the sound, the resulting signal passes through the DirectX® sound API without any further manipulations.

For purposes of modularity and simplicity, we keep the convolution algorithm in the generic global library `SoundAlgorithms`. This short algorithm needs no further explanation. However, the algorithms in the simulator do not currently run this code. Executing this process proves exceedingly inefficient because of the increasingly large number of additions and subtractions required. According to common understanding (Smith, 1997), convolution in the time domain does not work for real-time applications. However, we have a simple alternative: multiplication in the frequency domain.

$$Y(n) = i_0 * X(n) + i_1 * X(n-1) + i_2 * X(n-2) + i_3 * X(n-3) + ... + i_k * X(n-k)$$

Equation 2: Convolution of Sound [X] with Impulse [i]

```
for n=impulse_size...sample_size {
        output[n-impulse_size] = 0.0;
        for k...impulse_size
                output[n-impulse_size] += impulseResponse[k]*waveForm[n-k];
}
```

Table 9: Convolution Algorithm Pseudo-Code

Initially, the data for a sound file resides as pulse code modulation in the time domain. This means that data stores the pulses according to increasing time at specified sample rates. We can convert this discrete information into the frequency domain by a common process call Fourier transform. Once we have both the original signal and the impulse responses in the frequency domain, we quickly multiply the signals and transform the result back to the time domain for playback. Clearly, the cost of converting back and forth between the domains and the necessary math in the frequency domain must offset the expense of convolution in the time domain. Unquestionably, we far surpass this threshold because of three factors. First, the simulator can covert the original data samples into the frequency domain when it first loads the sound files at application initialization. This means that this part of the process occurs only once, regardless of how many times the simulator runs the algorithm. Second, we use the same data, in the same loop (see Table 10) for both channels, thereby minimizing processing cycles. Finally, instead of “reinventing the wheel” we have linked to a set of third party libraries to perform fast Fourier transforms (FFT). The FFTW library (see “FFTW Library” on page 84) operates these transforms tremendously fast, far more efficiently than we could program within the scope of this research.

We can confirm the efficiency of this process by comparing the two developed algorithms in this research. The computational differences between the Direct Paths algorithm (page 31) and the Bouncing Reflections algorithm (page 36) dwell in the process of calculating the reverberation paths and impulse responses and primarily the convolution aspect of the latter algorithm. We examine this in detail in the section “Validity” on page 104, but for now note that the convolution portion of the algorithm runs in real-time but does principally influence the speed of the acoustic algorithm.

```
MAXSAMPLES = numberOfOriginalSamples * BUFFER_SIZE_MULTIPLIER;
SoundPlayAlg_BR::convertAlgorithmToSamples() {
// The FFTs have been calculated. Multiply each sample in FFTs together linearly.
   for( n=0; n<MAXSAMPLES; n++ ) {
      outLeft[n][R]=inOriginal[n][R]*FIR_Left[n][R] - inOriginal[n][I]*FIR_Left[n][I];
      outLeft[n][I]=inOriginal[n][I]*FIR_Left[n][R] + inOriginal[n][R]*FIR_Left[n][I];
      outRight[n][R]=inOriginal[n][R]*FIR_Right[n][R] - inOriginal[n][I]*FIR_Right[n][I];
      outRight[n][I]=inOriginal[n][I]*FIR_Right[n][R] + inOriginal[n][R]*FIR_Right[n][I];
   }
   // Finally, now multiply each to get the data back...
   fftw_execute(planOutLeftREV);
   fftw_execute(planOutRightREV);
   // Copy the data back into the buffers.
   for( n=0; n<numberOfOriginalSamples*2; n++ ) {
      modifiedDataSamplesLeft[n] = outLeft[n][R] / MAXSAMPLES;
      modifiedDataSamplesRight[n] = outRight[n][R] / MAXSAMPLES;
   }
}               // end SoundPlayAlg_BR::convertAlgorithmToSamples() //
```

Table 10: Multiplication in the Frequency Domain Pseudo-Code

Table 10 shows the pseudo-code for performing these operations. Multiplication in the frequency domain actually requires two operations per channel. It must contend with both the real and imaginary components of the complex number. The FFTW library predefines this data type as simply a "`struct`" of two floating-point variables. Reconstructing the resulting data into something meaningful does warrant further scrutiny. In order to avoid circular convolution, the algorithm simply pads the number of samples to parse with a multiplier. This multiplier indicates the proportional amount of extra space to allocate larger than the number of sound samples. For example, if set to 3.5, the simulator will set the size of the sound buffer and subsequent FIR buffers to 3.5 times the number of samples from the original. The algorithm must finally divide the results stored in the real component of the outputs by the number of samples parsed (multiplier and number of samples from the original) to return to the `sampScaled` data type. See (Smith, 1997) for a detailed explanation of the FFT process.

Currently, the simulator does not generically store the code for multiplication in the frequency domain with the convolution algorithm in the `SoundAlgorithms` library. Rather, since presently only the Bouncing Reflections algorithm (see page 36) incorporates it, the code remains in the `SoundPlayAlg_BR` object. This avoids the messiness of sending arrays by reference to a generic function that we only call from one location in the code.

Integrated Libraries

Eventually, the goal of this research is to produce a fully functional integrated library, or API, that will provide true 3D audio for any application. For design and simplicity purposes, the simulator currently exploits a number of off-the-shelf API's to handle the more mundane aspects of the program. First, the Microsoft Foundation Class® handles the front-end graphical user interface. This API manages all of the basic functionality of the actual simulator application such as window creation and keyboard and mouse event handling. Just behind and integrated with MFC, the graphics library OpenGL® controls the entire visual 3D graphical pipeline. The simulator sends the data in the `Environment3D` object to the OpenGL® API, via various function calls, which then renders the graphics on the display screen. One can readily find both of these libraries commonly implemented in a multitude of settings and applications and thorough documentation of their workings and design.

Buried within the hierarchy of the `Sound3D` object, Microsoft's DirectX® library offers audio reproduction support through its DirectSound® interface. Presently, all audio feedback tunnels through this library. The simulator does not depend on the specifics of this API. It would not take much effort to utilize a completely different audio or graphical interface, and we have incorporated this feature, or ability, in the current state of the application by segmenting the dependent code with processor directives. The programmer need only provide support for an alternative API, embedding the instructions in appropriate locations, and then switch off any undesired API code with preprocessor definitions.

However, for the present, Microsoft's DirectX DirectSound® provides sufficient low-level support for the algorithms while allowing the simulator to compare them against DirectSound® 3D reproduction. As well, OpenGL® offers strong graphical hardware rendering support and unquestionable platform portability (see "Portability" on page 90). Below, we will explore the inclusion, advantages, and any shortcomings of these Application Programming Interfaces.

OpenGL® Library

Integral to the simulator and the experiment, the OpenGL® library allows us to provide a complex and accurate visual reproduction of the environment. The main hypothesis of this research, to approach the concepts and dilemmas associated with 3D virtual sound calculations, would have no meaning without an immersive 3D virtual environment that properly represents the expected visuals. In fairness, the visual component of Microsoft's DirectX® library, Direct3D®, does compare with if not outperform OpenGL® and we certainly could have programmed the simulator with this API. As well, we seriously considered several more sophisticated, high-level environments such as the Unreal® Engine [http://www.unrealengine.com/]. Three points led to the implementation of the OpenGL® library over alternatives.

First, the final version of this research contains approximately 25,000 lines of code. During previous research, we had developed generic libraries for reading files, loading and applying visual textures, enumerating three-dimensional math, and other non-trivial functions. Incorporating and extending these existing libraries seemed natural as well as continuing with the familiarity of a known environment. Furthermore, the learning curve associated with a different system undoubtedly outweighs any other possible benefits.

Second, OpenGL® has a proven reputation to programmers as a stable and easy to understand library. It profits from extensive global support with examples and instructions across a wide variety of applications. Considering the issues that we encountered using Microsoft's DirectX DirectSound® API and the incredible lack of acknowledgement of (needless to say, support of) these serious bugs, we feel quite justified in our choice. Furthermore, Microsoft® has a deserved reputation of imposing dramatic changes to their libraries that depreciate previous versions and disregard backwards compatibility. Other libraries, such as the Unreal® Engine have better reputations then DirectX®, however we considered a prebuilt engine based off gaming environments to have too high level of programming abstraction. With OpenGL®, we could create a generic environmental system not only usable by the graphics library, but also available to the audio portion of the system.

The third reason we selected the OpenGL® API above others possibilities, GLUT (OpenGL Utility Toolkit [http://www.opengl.org/resources/libraries/glut/]), considerably extends the OpenGL® library with easy to use functions and characteristics. These tools vary from simple, portable functions for drawing basic shapes like boxes and spheres to sophisticated windowing and callback routines.

DirectSound® Library

Previously described in a number of sections, the simulator employs Microsoft's DirectX DirectSound® API for loading and reproducing most sounds. Along with common runtime libraries for end-users to appropriate, Microsoft® regularly provides updated software development kits that include programming components such as libraries, symbols, and header files required by the compiler for coding and linking to the API. During the course of this research, Microsoft® has updated DirectSound® a number of times and we have attempted to maintain concurrency with these changes. However, at some point, we found it necessary to settle on a specific version for the remaining development. The version used for this research, DirectX® 9.0 works with "Microsoft DirectX SDK (November 2007)". Presently, Microsoft offers DirectX® 11.0 which links to "June 2010 DirectX SDK" [http://msdn.microsoft.com/directx/].

As mentioned previously, Microsoft has a tendency to make sweeping changes to their libraries that depreciate previous versions. The current version of DirectX® does not even include DirectSound®. Rather, Microsoft® has chosen to completely rewrite the audio stack and put the 3D sound portion in a sub-library of the new XAudio2 called X3DAudio. According to Microsoft® [http://msdn.microsoft.com/en-us/library/ee415813(v=vs.85).aspx]:

> XAudio2 is a low-level audio API. It provides a signal processing and mixing foundation for games that is similar to its predecessors, DirectSound and XAudio. XAudio2 is the long-awaited replacement for DirectSound. It addresses several outstanding issues and feature requests.

Despite this change in nomenclature, the X3DAudio library still does little more than account for the direct path between the sound source and the listener, "to position sound in 3D space to create the illusion of sound coming from a point in space relative to the position of the camera" [http://msdn.microsoft.com/en-us/library/windows/apps/ee415717.aspx].

As with the graphics library, we did have options other than DirectSound®. Considering the historical obstacles with DirectX® and DirectSound® [http://www.gamedev.net/page/resources/_/technical/directx-and-xna/directsound3d-r593], other sound libraries might have held certain appeal. Historically, alternatives included OpenAL [http://www.openal.org/], Environmental Audio Extensions (or EAX), or Aureal 3-Dimensional (A3D) [http://www.soundblaster.com/eax/]. All of these, Creative Technology Ltd. has acquired over the past decade and either no longer supports or has significantly less functionality than DirectSound®.

FFTW Library

Critical to the core of the sound processing algorithms, the simulator must perform computationally intensive Fourier transforms and reverse transforms frequently and therefore efficiently. Fourier transforms provide an alternative to the impractical convolution process by converting two large data sets into the frequency domain, allowing the program to multiply the streams linearly, and translating the result back into the time domain for consumption. A basic Fourier transform would have an unacceptable time complexity of O(n^2) while fast Fourier transform (FFT) algorithms have computational complexity as little as O(n log n). Some derivations such as fast cosine transform (FCT), Cooley–Tukey FFT, or other specialized discrete cosine transform (DCT) algorithms can execute even faster in controlled circumstances.

Programmers have used these techniques for decades, so formulas and even direct code examples abound for our use. Since the speed of the algorithms in the simulator depends so heavily on the efficiency of this segment of code, we carefully considered available options. It quickly became apparent that the FFTW library [http://www.fftw.org/] would perfectly suit the needs of the simulator. By all accounts, this library runs as quick, if not significantly faster than its competition. Moreover, FFTW boasts the added features of being easy to use, well-documented, extremely portable, processor optimized, and most importantly free. The simulator does include a version of the common FFT algorithm within the `SoundAlgorithms` library; however, no process in this project employs it.

In order to exploit the FFTW library efficiently, the simulator must take a few steps to properly setup the system. First, the library prefers to involve a set of mechanisms called plans and wisdoms. It uses plans to store all the internal data that FFTW needs to compute a given transform. These plans can take incredible amount of processing time to create, but they allow the library to execute the actual FFT exceedingly fast. When creating a plan, we must specify the input and output arrays of complex numbers, the size of these arrays, and the direction of the transform. For each instance of a loaded sound file, the simulator must manage the five plans listed in the top of Table 11. This happens each time the scene file lists a SoundSource3D object, regardless if it occurred previously. Clearly, we could minimize the number of plans required by recognizing this fact. However, the only significant drawback in the current implementation occurs when the simulator first generates the required plans. On a modern dual-core processor, this process takes up to an unacceptable six to ten minutes! Fortunately, we have wisdoms at our disposal, which provide a process for saving plans to the local disc and restoring them the next time simulator loads.

In the Scene3D loadScene function, before it encounters any sound files, the simulator looks to see if a properly formed wisdom exists on the hard drive. If it does, the loadScene function proceeds to load the sound files and restore the associated plans from the wisdom file. This takes almost no time. Otherwise, the simulator warns the user of the impending time-consuming process and then continues on, loading the listed sound files and generating sets of new plans for each instance. Once it finishes loading all sound files, the Scene3D object stores the processed plans in a new wisdom file if needed. Each time the simulator runs on a new computer, the list of sound files in the scene changes, the compiler switches from debug to release mode, or we utilize a different FFT method, the simulator must rebuild the wisdom file. Otherwise, this deplorably prolonged process need only run one time for a given scene and not each time the simulator initializes. Table 11 lists the complicated process for loading the FFTW library.

Each time the simulator loads a sound file into the system it associates one of each of the algorithm objects to it and these objects have the option of creating and storing plans. Currently, only the SoundPlayAlg_BR object (see “SoundPlayBouncingReflections

Algorithm Object" on page 64) requires the FFT process so only it retains plans for the different sound files. Each instance of this object also contains an `ImpulseResponse` object, which in turn keeps two forward plans, one for each ear.

The simulator loads and stores these uniquely named wisdom files that dramatically speed up the loading of plans. These wisdom files sit within the working directory and have tags in the filename to help maintain concurrency. They depend on factors such as the specific scene name, the mode debugging mode of the compiler, the computer name, and the mode of the FFTW library. Since the FFTW system can run in a number of different modes, the simulator universally hard-codes this variable for all FFT calls. It currently sets the mode to `FFTW_MEASURE` that the documentation [http://www.fftw.org/fftw3_doc/Planner-Flags.html#Planner-Flags] describes as a moderately optimized plan.

```
MAXSAMPLES = numberOfOriginalSamples * BUFFER_SIZE_MULTIPLIER;
// For Each Instance of a Loaded Sound File:
//   For Each Algorithm that Requires FFT:
//     Original Mono Sound Data FORWARD                    [size=MAXSAMPLES, inOriginal]
//     Left Channel Impulse Response Data FORWARD          [size=MAXSAMPLES, FIR_Left]
//     Right Channel Impulse Response Data FORWARD         [size=MAXSAMPLES, FIR_Right]
//     Left Channel Modified Sound Data REVERSE            [size=MAXSAMPLES, outLeft]
//     Right Channel Modified Sound Data REVERSE           [size=MAXSAMPLES, outRight]
Scene3D::loadScene() {
   // Load FFT Plans...
   // basename contains "TDS_" & [METHOD] & ["_D_"|"_R_"] [filename]
   bool bNeedToCreateWisdom = true;
   getUniqueFilename(uniqueName, basename, computerName, "fftplan", true);
   if( fftw_import_wisdom_from_filename(uniqueName) ) bNeedToCreateWisdom = false;
   else MESSAGE("Wisdom for FFT was NOT found! This will take a LONG while...");
   // Next load all sounds! For each sound, call Sound3D::loadSound()
...
// Called within the Scene3D::loadScene() function //
int Sound3D::loadSound() {
   soundObjects[sndIndex][3DCALC_SIMPLE] = new SoundPlaySimple();
   soundObjects[sndIndex][3DCALC_DIRECTX] = new SoundPlayDirectX();
   soundObjects[sndIndex][3DCALC_DIRECT_PATHS] = new SoundPlayAlg_DP();
   soundObjects[sndIndex][3DCALC_BOUNCING_REFLECTIONS] = new SoundPlayAlg_BR();
   for( i=0; i<NUM_ALGORITHMS; i++ ) {
      soundObjects[sndIndex][i]->initialize();
      soundObjects[sndIndex][i]->env = env;
      soundObjects[sndIndex][i]->parseSoundFile();
   }
}               // end Sound3D::loadSound() //
// Called within the Sound3D::loadSound() function //
SoundPlayAlg_BR::parseSoundFile() {
   // Now allocate the memory for the FFTs to come...
   inOriginal = (fftw_complex*)fftw_malloc(sizeof(fftw_complex) * MAXSAMPLES);
   outLeft = (fftw_complex*)fftw_malloc(sizeof(fftw_complex) * MAXSAMPLES);
   outRight = (fftw_complex*)fftw_malloc(sizeof(fftw_complex) * MAXSAMPLES);
   planInFWD = fftw_plan_dft_1d(MAXSAMPLES,inOriginal,inOriginal,FFTW_FORWARD,METHOD);
   planOutLeftREV = fftw_plan_dft_1d(MAXSAMPLES,outLeft, outLeft,FFTW_BACKWARD,METHOD);
   planOutRightREV = fftw_plan_dft_1d(MAXSAMPLES,outRight,outRight,FFTW_BACKWARD,METHOD);

   // Finally, put stuff into the FFT and do an initial run!
   for( n=0; n<numberOfOriginalSamples; n++ )
      inOriginal[n][R] = originalDataSamples[i];
   for( n=numberOfOriginalSamples; n<MAXSAMPLES; n++ )
      inOriginal[n][R] = 0.0; // Pad with the original the zeros before doing the fft...
   fftw_execute(planInFWD);
   m_FIR.setMaxSize();
}               // end SoundPlayAlg_BR::parseSoundFile() //
// Called within the SoundPlayAlg_BR::parseSoundFile() function //
ImpulseResponse::setMaxSize() {
   FIR_Left = (fftw_complex*)fftw_malloc(sizeof(fftw_complex) * MAXSAMPLES);
   FIR_Right = (fftw_complex*)fftw_malloc(sizeof(fftw_complex) * MAXSAMPLES);
   planFIRLeftFWD = fftw_plan_dft_1d(MAXSAMPLES,FIR_Left,FIR_Left,FFTW_FORWARD,FFT_METHOD);
   planFIRRightFWD=fftw_plan_dft_1d(MAXSAMPLES,FIR_Right,FIR_Right,FFTW_FORWARD,FFT_METHOD);
}               // end ImpulseResponse::setMaxSize() //
...
   // Finally save the FFT Plans...
   if( bNeedToCreateWisdom )
      if( !fftw_export_wisdom_to_filename(uniqueName) )
         EXIT("ERROR! Unable to create the wisdom for the FFT!!!", 4523352);
}               // end Scene3D::loadScene() //
```

Table 11: FFTW Loading Process

Experiment Structure

In the section “Research Methods” on page 93, we will go into much more detail about the processes involving the experiments for this research. However, in the following sections we briefly describe the design choices and implementations of the experiment side of the TDS Simulator.

Data Field	Example	Comment
SubjectID	8396	Randomly generated (unique)
Date	11/28/2011	
Time	10:16:12	
Algorithm	DIRECT PATHS	or SIMPLE, DIRECTX, BOUNCING REFLECTIONS
Number of Sources	4	Random from 2 to 7 sources
Attempts	1	Number of attempts made before correct
Guessed Correctly	T	T or F
Correct Answer	2	Index of Source for the correct answer
SoundFile	bounce.wav	
Depth	1	Random from 1 to 10
Breadth	4	Random from 1 to 10
OrientationWeight	0.452	Random from 0.0 to 1.0
Listener LastX	3.889	
Listener LastY	32.787	
Listener LastZ	1.93	
Listener LastT	78.5	
Listener LastP	0	
Source1X	1.468	
Source1Y	37.237	
Source1Z	2.613	
Source1T	188.037	Horizontal orientation Theta, from 0 to 360
Source1P	354.056	Vertical orientation Phi, from 0 to 360
Source2X	5.404	
Source2Y	33.037	
...		Continues through "Number of Sources"

Table 12: Experimental Data Storage Format

Testing Modes

In addition to the basic user-interface, which allows the user to control all aspects of the simulator, the TDS Simulator also provides a testing environment for subject-based experiments. As highlighted in the “Research Methods” section on page 93, the simulator provides two methods of tests. A test subject sees the data either with or without feedback. The simulator presents the subject with ten questions with feedback followed by ten questions without feedback. See “Experiment Analysis” on page 109 for a thorough analysis and comparison of the testing mode techniques.

Experimental Testing Data Structure

Stored through the `TestingData` object, the simulator collects and outputs a great deal of information during the course of the experiment. Each subject runs through twenty questions, ten with feedback and then ten without. The simulator records many variables for each question (see Table 12). We discuss the data and analyze the results in section “Results” on page 102. After each subject completes the experiment, the `TestingData` object appends the collected data to the end of a comma delimited text file for later processing and analysis.

Portability

Even though we have not explicitly tested this at this point, the system and designed algorithms should retain general portability to other compilers and platforms. Addressed in many of the previous sections, the TDS Simulator connects to a diverse set of external API libraries, including MFC®, OpenGL®, GLUT®, DirectX®, MCI®, and FFTW®. In all fairness, certain components of the simulator do depend on portions of these libraries and therefore portability does not flow throughout the system universally. However, we have made significant effort within the 25,000+ lines of code to designate and segment dependent sections in order to facilitate easy transition to alternative libraries. Most of the functions that require linkage to external libraries have surrounding processor directives such as

#ifdef GRAPHICSENGINE_OPENGL and then #endif. Before building the program, we simply set the necessary preprocessor definition variables to tell the compiler to include the desired sections or not. Enhancing the simulator to attach to different API libraries simply requires copying these sections of code and modifying appropriately. This could even easily allow for support of multiple side-by-side competing options of certain library types.

Some external libraries, however, need not have alternative options as they already purport portability to different platforms. These include OpenGL®, GLUT®, and FFTW®. Unfortunately, and not surprisingly, the libraries from Microsoft® do not natively afford portability to other systems. Since no reasonable alternative audio library currently exits (see "DirectSound® Library" on page 83), the TDS Simulator loses portability on a fundamental and critical level. Fortunately, fault for this does not fall on the design of the simulator, but rather lack of substitutes in the industry. Once we have alternatives to the sound API available, the simulator will readily embrace them.

The TDS Simulator currently compiles without error or difficulty through Microsoft's Visual Studio® 2008 C++ programming environment. Although this is not the current version of Visual Studio [http://www.microsoft.com/visualstudio/eng], during the course of development of this project, we resolved to lock down and stay with this implentation. However, nothing inherent in the program requires this specific programming choice other than the main initial application creation process. Most of the code falls into modular, typically object-oriented libraries, independent of the main application. Switching to another C++ programming environment would require a significant amount of effort, but undoubtedly, most of this would occur at the top level of the simulator application. This research does however rely on a programming environment that uses the C++ programming language. Conversion to another, even similar language would require a complete overhaul. Happily most serious developers still work with, if not exclusively, this programming language.

Research Methods

Introduction

Through this study, we intend to provide a better understanding of how we perceive sound. We wish to explore virtual 3D perceptions, concentrating on the relationship between aural awareness and visual interpretation. We have developed a virtual 3D simulator (see "The TDS Simulator" on page 49) that allows a user to move through a graphical setting while listening to sounds from objects positioned throughout the virtual environment. The simulator reproduces these sounds using a variety of techniques including simple playback (i.e. no 3D virtualization), implementation of the Microsoft's DirectX DirectSound® library, or utilization of the 3D acoustic algorithms developed in this research. Table 13 lists the three primary hypotheses for this experiment. In the sections below, we will provide further explanation and analysis of the testing details.

In order to examine the issues of generating 3D sound, we mandated an experiment to assess a reasonably large population sample. See "Experimental Environment" on page 97 and "Experimental Procedure" on page 99 for an extensive breakdown of the final experiment. After receiving approval from The Institutional Review Board, we commenced this set of experiments involving college student subjects in controlled settings. Initially, in

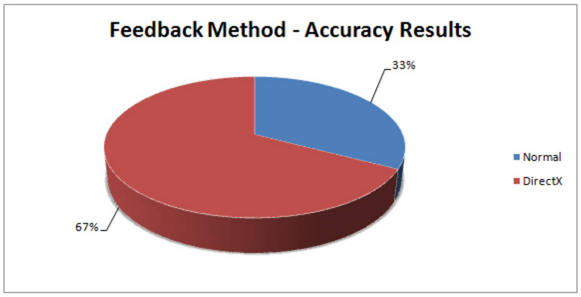


Figure 17: Initial Results for Testing with Feedback

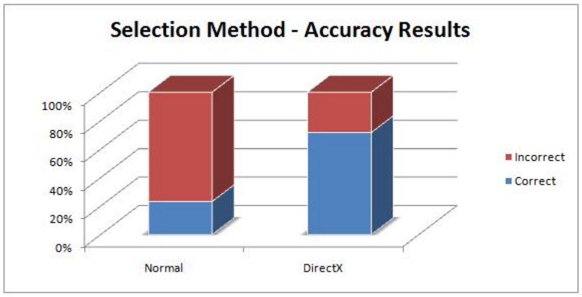


Figure 18: Initial Results for Testing without Feedback

July of 2008, we ran the experiment with a small set of test subjects to obtain a proof-of-concept for the basis of the project. This portion of the experiment simply demonstrated that we could establish that the DirectX DirectSound® industry standard 3D sound algorithm performs better than reproducing the same sounds with no 3D acoustic algorithm at all (the Simple algorithm). Running this preliminary experiment also allowed us to evaluate and refine the testing procedure and analysis techniques for future experiments. Figure 17 and Figure 18 show the results of this prototype experiment. The experiment included just four test subjects and transpired in a small room on the campus of University of Louisiana, Lafayette. Since we covered this small pseudo-experiment in past research, we simply highlight the results. The methods of analysis of the data have not changed dramatically and so we leave further explanation for other sections.

After competition of the initial proof-of-concept experiment and having concluded the validity of the simulator and testing procedures, we pursued the design and development proprietary algorithms for comparison. Through this process, we developed two strong candidates for consideration. The Direct Paths algorithm (see page 31) correlates to the industry standard DirectX DirectSound® method in many ways while the Reflected Paths algorithm (see page 34) significantly extends these two relatively simple approaches. The final experiment design compares these three methodologies with each other and again incorporates the control of generating sound with no 3D acoustic algorithm.

After some trial and error, in October of 2012, we concluded a formal round of experimental tests involving over one-hundred student test subjects. Described in detail in the following sections, this experiment took place on the campus of Southeastern Louisiana University in the College of Business. The test subject pool comprised mostly of college students from this department, with varying degrees of technical experience and backgrounds.

Hypotheses

The theories and premises behind this research maintain that current algorithms and techniques for producing 3D sound in virtual environments fall measurably short of more advanced potential practices. Table 13 lists the three primary hypotheses for this experiment.

Hyp1	While DirectX DirectSound® will perform better than Simple playback, the 3D algorithms from the research will produce more accurate sound reproduction than DirectX DirectSound®.
Hyp2	Over time, with training, a test subject's performance with the research algorithms will improve, while training with the other algorithms will not show improvement.
Hyp3	One of the algorithms developed in this research will produce superior results than the others. We anticipate the more complicated Bouncing Reflections algorithm to outperform all others.

Table 13: Experimental Hypotheses

Methods employed in the research algorithms fundamentally derive from procedures developed in off-line architectural analysis applications (e.g. EASE [http://www.auralisation.com/], CATT Acoustic [http://www.catt.se/], and Odeon [http://www.odeon.dk/]). These techniques applied in non-real-time applications provide acousticians somewhat accurate models and predictions of the sonic nature of conceptual or real buildings and spaces. This research takes the basic concepts of these applications and, in effect, extends the process. Instead of predicting the nature of the virtual space via these algorithms, the simulator recreates the acoustics based on the results. Most acousticians using the available modeling software attempt to eliminate or minimize the natural and undesirable properties of the environment. In a truly virtual world, the goal is exactly the opposite. We wish to reproduce these artifacts and deficiencies completely. With the reproduction of accurately modeled 3D sound, the simulator should generate superior localization to that of the industry standard techniques.

The first part of the first hypothesis, that DirectX DirectSound® should engender superior results over simple playback, we proved in the preliminary experiment and illustrate in Figure 17 and Figure 18. We take this conclusion as the basis for further hypothetical consideration. We will additionally establish that the 3D algorithms developed in this research (see “Current Work” on page 29) will produce improved sound reproduction over the current industry standard DirectX DirectSound®.

Secondly, we consider the impact of training, or experience of the test subject, in the experiment results. As the test subject proceeds with the experiment, accuracy should noticeably improve. Since we ask each subject only 20 questions and the experiment design breaks these into two groups, we may find this trend difficult to detect in the data. In order to prove this hypothesis conclusively we may require future experimentation. However, subtle analysis of the data might highlight meaningful inclinations.

Finally, and probably most obvious, we anticipate a conclusive ranking of the algorithms according to accuracy. One algorithm, presumably one of the research algorithms, should stand above the others. Because of the complexity compared to the others, we predict the Bouncing Reflections algorithm (see page 36) to outperform the Reflected Paths algorithm (see page 34) and DirectX DirectSound® algorithm, not to mention the Simple algorithm. Furthermore, we expect Reflected Paths algorithm to work better than or at least equivalent to the DirectX DirectSound® algorithm. Both of these algorithms use only basic 3D information from the environment and neither considers the virtual geometry in any substantial manner.

Experimental Environment

The simulator combines both the visual and aural reproduction of a virtual 3D environment. A subject can move avatars and sound sources throughout the virtual world represented on the display of a computer. On command, a sound source object will play from a set of pre-specified sounds, utilizing one of the various algorithms to enhance playback. Through the simulator, the user can select the playback method from Simple algorithm (i.e. normal playback or no 3D algorithm), the ubiquitous Microsoft DirectX® DirectSound® interface, or one of a number of 3D algorithms developed in this research. Most of the research algorithms can take into account the geometrical and acoustic properties of the virtual environment while off-the-shelf algorithms (i.e. Microsoft's DirectX DirectSound®) only concern themselves with distance and orientation between the source and the listener.

For the actual experiment, subjects sit in front of a standard computer that has conventional 3D graphics processing capabilities and no specialized 3D audio hardware. A traditional wireless mouse and laptop keyboard allow movement of the avatar and selection of the test question answers. Table 14 lists the computer setup used in the formal experiment. For the most part, this consisted of an off-the-shelf common system [http://store.sony.com/webapp/wcs/stores/servlet/ProductDisplay?&productId=8198552921666441731]. Additionally, we utilized the built-in soundboard on this computer, incorporating no specialized sound processing or reproduction abilities.

Model	Sony® VAIO™ T Series Ultrabook (SVT13118FXS)	
Processor	Intel® Core i7 1.90Ghz processor	
Storage	6GB memory, 128GB SSD hard drive	
Video	13.3" wide screen (1920 x 1200), Intel® HD Graphics 4000 with shared memory	
Audio	Realtek® High Definition Audio, Sennheiser® HD 280 Pro headphones	

Table 14: Computer Specifications Used in Experiments

Finally, the simulator generates sound feedback through a provided pair of circumaural (over the ear) Sennheiser® HD 280 Pro headphones [http://www.sennheiserusa.com/professional-dj-headphones-HD-280-PRO_004974]. This specific model and brand of headphones allows freedom of head motion while offering an uncorrupted flat response for uniform playback. They also, according to product specifications, provide "aggressive isolation from noise by design", meaning that the headphones comfortably encompass the entire ear and therefore effectively segregate noise from the external environment without the use of artificial filters. Anecdotal evidence supports this and even cursory Internet searches uniformly review these headphones as flat or neutral sound with excellent passive noise attenuation. Figure 19 shows a picture of the specific device used in this research.

The testing entirely took place in a secluded room in the College of Business at Southeastern Louisiana University [http://www.selu.edu/acad_research/colleges/bus/index.html]. The image in Figure 20 shows the room and configuration arrangement. Measurements from a variety of sensors confirmed the quiet nature of the space at an average of about 54dB, sufficient for use with this experiment.

Figure 19: Sennheiser® HD 280 Pro Headphones

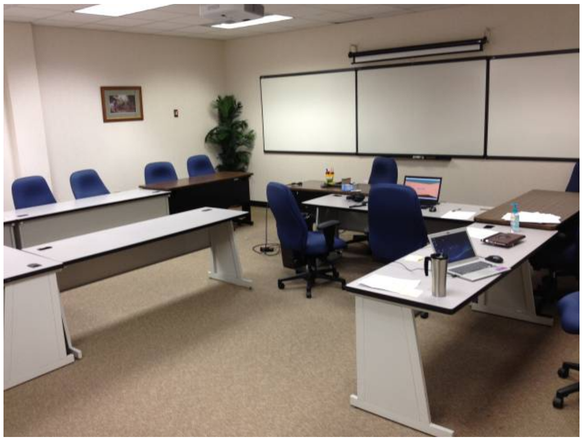


Figure 20: Picture of Experiment Environment

Experimental Procedure

Table 29 on page 159 exhibits a typical transcript of the procedure used to setup the experiment for each test subject. When the student enters the room, the researcher initially asks him to familiarize himself with the simulator by exploring the virtual world. After a brief introduction and basic training on moving through the environment, the simulator switches to the first question in the initial testing mode. The simulator places the avatar in a virtual room with two to seven, visually identical sound source objects, only one of which will produce sound. Of these randomly labeled, located, and oriented sound source objects, the TDS Simulator randomly selects one to playback sound through a randomly selected algorithm. The subject then must determine which object is producing the sound according to what he sees and hears. This process repeats for a series of twenty questions for each test subject. Upon conclusion of the experiment, the simulator presents the results and performance of the test subject and digitally records the generated data. The entire process then repeats for the next subject attending.

Testing Details

This study exposes the subjects to minimal/nominal risk. We ask test subjects to sit in front of a standard computer, wear a pair of off-the-shelf circumaural headphones (encompassing the entire ear), and move an avatar through the virtual environment with the keyboard and mouse. The simulator allows for two methods of questioning. We ask each test subject to go through ten questions with feedback and then ten questions without feedback. Each question should take less than thirty seconds, so we estimate the entire test period per subject to last no more than fifteen minutes.

Data and Data Collection

For each test subject, we assign a randomly generated, unique number and store this with the recorded data for each question. The simulator automatically outputs the data into a comma-separated value text file for later analysis. Data stored includes the time and date of the test, which algorithm generated the sound, the number of guesses, and the accuracy of the selection (correct or incorrect), and virtual geographical data of the simulation. The simulator also stores internal variables appropriate to the question. The geographical data consists of the location and orientation of the avatar and all sound sources present in the specific test. Since the subject can actually move the listener avatar during the test, we record only the final (i.e. when the specific question concludes) coordinates. Table 12 in the section "Experimental Testing Data Structure" (see page 90) lists the set of variables stored by the simulator and even provides a real example from the experiment.

With Feedback	• The simulator immediately informs the subject if their guess is correct.
	• If incorrect, the simulator prompts the subject to try again.
	• The simulator records each conjecture until the subject chooses correctly.
Without Feedback	• The simulator does not inform the subject of the accuracy of each choice.
	• After each selection, the simulator moves immediately to the next question.

Table 15: Testing Modes for the Experiment

Data Analysis

After we test a statistically appropriate number of subjects, we will analyze the data for algorithm effectiveness and possible future refinement of multiple aspects of the research and algorithms. Specifically, the study follows trends in accuracy across algorithm type, number and/or coordinates of sound sources, proximity of listener to correct sound source, and improvement of subject response accuracy over time. The analysis will explore, to the best that statistical accuracy will allow the three hypotheses listed in Table 13 on page 95. We will consider both a pragmatic and statistical (Spiegel, Schiller, & Srinivasan, 2009) approaches to analysis of this data.

Results

Introduction

In any experiment and associated study, one must consider and analyze multiple components of the topic. Below, we will separate these elements, evaluate their impact and effectiveness, and finally assemble the results into a cohesive conclusion. The following sections include an analysis of the assumptions made in the experiment, justification of the validity of the design and implementation, outline of the algorithms, a detailed investigation of the experiment data, and finally a summary of the application of the three hypotheses listed previously.

Assumptions

Algorithm Complexity

For reasons of efficiency and simplicity, we chose to limit the fidelity, or complexity of the Bouncing Reflections algorithm. Two variables control the different parts of the complexity of this algorithm. First, **breadth** determines the number of paths or directions that the algorithm should generate and traverse. As listed in Table 4 on page 39, the first 22 directions subdivide the spherical space around the sound source object based on its orientation. After this, the algorithm randomly generates any remaining needed paths. Both the breadth and depth variables range from integers 1 to 10 and the simulator randomly assigns these values for each question in the experiment. The **depth** variable controls the number of segments, or bounce reflections each path in the algorithm must navigate. Clearly, the combination of these two variables directly contributes to the complexity, and therefore the speed of the algorithm. Running the algorithm with a high depth and low breadth or vice versa will result in an almost instantaneous return.

For the purposes of the experiment, we limited not only the fidelity, but also the resolution of the algorithm in the extreme cases. Table 16 lists the values and results for the complexity variables used in this experiment and the speed test. Subsequent to the experiment, we pushed a large number of algorithm executions through the simulator to evaluate speed, not accuracy, of the algorithms at different complexity levels. In order to maintain acceptable processing times in the experiment, we primarily focused on the faster, low complexity level settings. The speed tests allow for more granulation and higher complexities levels. At higher breadth and depth in the experiment, the simulator clumps the resolution to the values listed in Table 16. Shaded areas in the table indicate values not employed in the formal experiment.

Paths or Directions										
Breadth:	1	2	3	4	5	6	7	8	9	10
For Experiment	4	6	14	22	1,000	1,000	1,000	1,000	1,000	1,000
Speed Test	4	6	14	22	50	100	300	1,000	5,000	10,000
Segments or Reflections										
Depth:	1	2	3	4	5	6	7	8	9	10
For Experiment	1	10	20	40	80	160	300	300	300	300
Speed Test	1	10	20	40	80	160	300	1,000	5,000	10,000

Table 16: Bouncing Reflections Complexity Increments

Sound Files

Keeping with the concept of design simplicity compelling analyzable results, we limited the sound files in the experiment to only two options. Both originating from the public domain and stored in monaural 16-bit PCM format, these sound files represent audio that one could typically find in any 3D environment. The file “dogbark.wav” lasts approximately 1.5 seconds with 32,240 samples and consists of a natural sounding dog barking twice. Sound file “bounce.wav” has a synthetic disposition with 6,712 samples in 0.6 seconds. Figure 21 and Figure 22 depict the waveforms for these sound files.

Virtual Environment

Though the TDS Simulator boasts the capability of processing complicated structures and non-rectangular spaces, we limited the experiment to a simple case. In order to provide control for this element of the experiment, all questions occurred in a building with a single room measuring about 8 x 14 x 3 meters (exactly 300 x 550 x 120 inches). This room has just one door and no windows. Furthermore, all of the walls have absorption and dispersion coefficients of 0.10 and 0.05, respectively. The ceiling and floor both have values of 0.05 and 0.01. During the experiment, we did not vary these factors.

Validity

In this section, we detail the various methods for judging the general validity of the algorithms designed for this experiment. For the most part, these analyses follow straightforward and objective measurements. For now, we do not consider formal experiment results.

Efficiency Validity

Without question, the simplest measurement of the effectiveness of an algorithm is the speed at which it performs. As these routines ideally should run in real-time, this

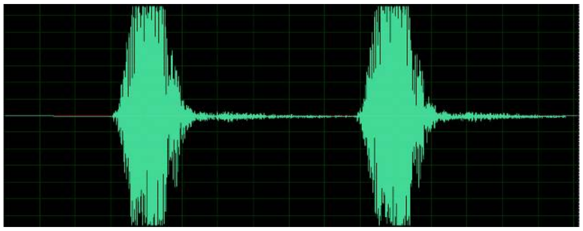


Figure 21: Waveform for 'dogbark.wav'

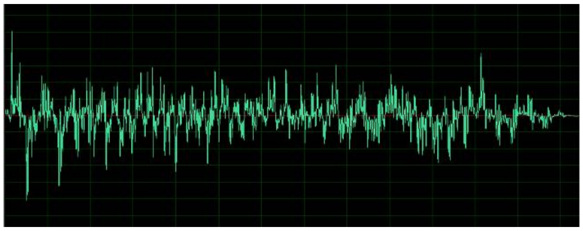


Figure 22: Waveform for 'bounce.wav'

threshold necessitates that it takes no more than 150ms (see “Latency” on page 47) from initialization of the sound to the actual auralization. Without proper specialized hardware (see “Software vs. Hardware” on page 156), the algorithms in this research must run completely in the software realm. Therefore, we must give some allowance to efficiency and still consider algorithms that fall outside this threshold. However, speed results should remain within a reasonable range (e.g. somewhere under or around two seconds). Table 17 lists some of the efficiency results for the Bouncing Reflections algorithm. After completion of the experiment, we designed a simple routine to run the simulator while repeatedly playing sounds through the Bouncing Reflections algorithm. This simulation randomly places the sound objects and the listener in the same space and in the same manner as the formal experiment. Each cell in the table represents the average from over twenty thousand random trial runs. The same computer system used in the experiment generated these results. We did not run this process for the other algorithms since the Simple, DirectX, and Direct Paths algorithms all ran almost instantly or at least undetectably for the time precision of the computer.

	Depth:	4 paths	6 paths	14 paths	22 paths	50 paths	100 paths	300 paths	1,000 paths	5,000 paths	10,000 paths
Breadth:		1	2	3	4	5	6	7	8	9	10
1 reflection	1	3.04	2.48	2.26	2.88	3.18	3.95	7.16	19.19	118	248
10 reflections	2	2.11	2.57	3.10	3.03	3.57	3.10	7.41	20.00	124	280
20 reflections	3	2.00	2.56	3.06	3.19	2.76	4.16	6.96	20.31	121	273
40 reflections	4	2.49	2.90	2.49	2.97	3.09	4.54	7.42	20.88	125	277
80 reflections	5	2.73	2.37	2.83	3.04	3.23	3.70	7.88	21.37	125	290
160 reflections	6	2.46	2.55	2.54	2.93	3.95	4.45	8.33	22.67	134	295
300 reflections	7	3.06	2.89	2.58	2.81	4.02	4.51	8.37	25.00	145	313
1,000 reflections	8	2.68	2.87	3.41	3.47	4.07	5.84	12.13	35.68	197	416
5,000 reflections	9	2.40	2.69	3.76	4.55	6.02	11.13	30.00	98.91	495	1018
10,000 reflections	10	2.83	3.01	4.35	5.67	10.25	18.66	52.27	170	883	1772

Table 17: Average Algorithm Runtime (ms) for the Bouncing Reflections Algorithm

Shaded cells again denote groups of data that we did not allow in the formal experiment (see Table 16). Close inspection of these speed results reveal that the increase of both the depth and breadth for the algorithm most significantly determines the impact on speed. However, even at extreme computations, where the algorithm must compute 1000 directions (breadth of 5 or more) and 300 reflections per path (depth of 7 or more), the algorithm still runs in far less time than the 150ms required by the latency threshold.

complexity = depth + breadth

	Sound File = bounce.wav				Sound File = dogbark.wav			
Complexity	Algorithm	Conversion	Update	Total	Algorithm	Conversion	Update	Total
2	3.39	15.09	0.71	68.84	2.65	3.92	0.20	66.08
3	2.63	15.26	0.72	64.91	2.02	4.26	0.22	60.37
4	2.27	15.31	0.78	66.17	2.27	3.94	0.14	62.06
5	2.87	15.12	0.62	58.99	2.59	3.97	0.16	57.89
6	2.81	15.11	0.68	68.92	3.04	3.83	0.15	51.41
7	2.76	15.50	0.59	62.20	2.52	3.73	0.15	56.43
8	2.85	15.18	0.62	68.61	2.84	3.92	0.07	45.92
9	6.83	15.20	0.50	68.29	6.54	3.73	0.22	59.31
10	7.84	15.58	0.72	69.11	9.46	3.34	0.22	42.87
11	11.31	15.46	0.63	73.47	11.54	3.23	0.00	62.14
12	20.65	16.13	0.97	83.98	21.07	3.61	0.16	69.84
13	20.22	15.59	0.22	83.55	22.47	4.12	0.21	78.04
14	21.94	15.20	0.31	74.91	23.44	4.19	0.11	83.23
15	24.14	16.31	0.00	79.46	25.88	4.50	0.09	74.78
13	3.90	14.96	0.86	62.10	4.27	3.48	0.11	60.13
14	5.72	14.50	0.81	79.82	6.15	4.33	0.10	65.53
15	9.78	15.26	0.58	77.91	12.75	3.50	0.08	62.08
16	20.56	14.86	0.55	85.70	27.24	3.52	0.05	76.80
17	113	14.82	0.60	163	137	3.89	0.10	182
18	397	15.26	0.87	448	507	4.14	0.24	565
19	794	14.58	0.30	849	1106	4.30	0.20	1158
20	1451	14.69	0.61	1508	2080	4.12	0.00	2134

Table 18: Average Runtime (ms) Breakdown for the Bouncing Reflections Algorithm

Table 18, originates from the same data as Table 17 and highlights the speed impact from the different portions of the Bouncing Reflections algorithm. To simplify analysis, we combined the depth and breadth variables by adding them into a single factor: complexity. Again, the shaded area underscores data corresponding to depth and/or breadth not allowed in the experiment. In addition, we note that a complexity of say 13 could consist of a number of combinations of the depth and breadth variables. Certainly, based on the results of the previous table, adding 10 and 3 (or 3 and 10) would produce notably different results than adding 6 and 7 (or 7 and 6). However, clear trends still appear in the data in this format. Finally, we also separated the data by sound file.

In conclusion, one can easily see, except in extreme fidelity cases, that the conversion segment of the routine requires the most time. We do not find this surprising, as this portion bundles the (linear) multiplication of the signals, the inverse FFT of the results, and finally scaling of the samples. The final section, updating, takes very little time, as it merely moves the data from the conversion portion into structures readable by the sound engine for output. Both conversion and update portions maintain an approximately static limit, regardless of simple or extreme fidelity or complexity. Most importantly though the algorithm section takes very little time to run. This supports the previous assertion that the computational differences between the Direct Paths algorithm and the Bouncing Reflections algorithm dwell in the process of calculating the reverberation paths and impulse responses and primarily the convolution aspect of the latter algorithm.

Logical Validity

On a somewhat more subjective level, logical validity analysis considers whether the resulting data seems appropriate given the environmental factors. A proper impulse response should contain hundreds and maybe even thousands of hits (see Figure 33 and Figure 34). Clearly, the results will depend proportionally on the depth and breadth, or complexity, of the algorithm. Additionally, we expect the impulse response to behave in what amounts to a commonly perceived manner, exponentially decaying with time.

Figure 23 through Figure 26 illustrate actual impulse responses from the simulator using the Bouncing Reflections algorithm. Datasets "A" and "B" show results from running the simulator at the maximum resolution (depth and breadth) used in the experiment. These two FIR's typically consisted of ten to twenty hits, sparsely distributed over the duration of the calculations. Both datasets "C" and "D" have far more dense dispersal of impulses due to the dramatically larger number of paths and reflections traversed. Running the simulator at this level generates an average of 150 impulse hits but requires well over one second of algorithm processing time. Clearly, all examples have the expected exponentially decaying nature and therefore pass logical validity.

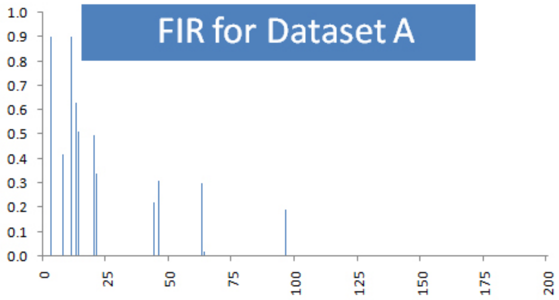


Figure 23: FIR A (300 reflections, 1,000 paths)

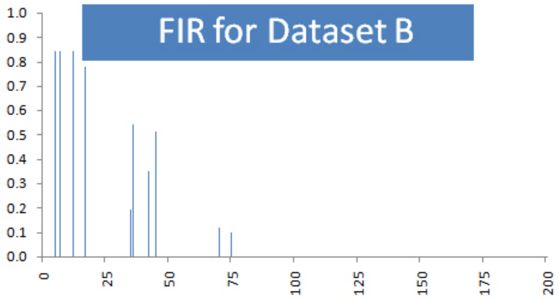


Figure 24: FIR B (300 reflections, 1,000 paths)

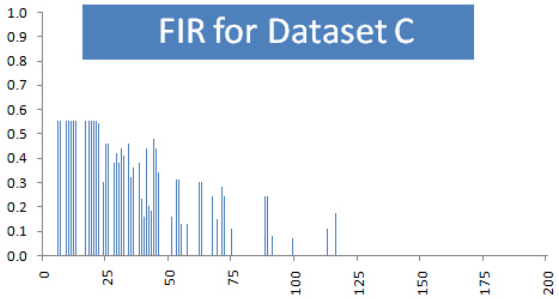


Figure 25: FIR C (10,000 reflections, 10,000 paths)

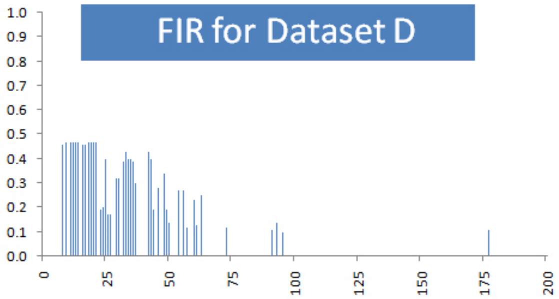
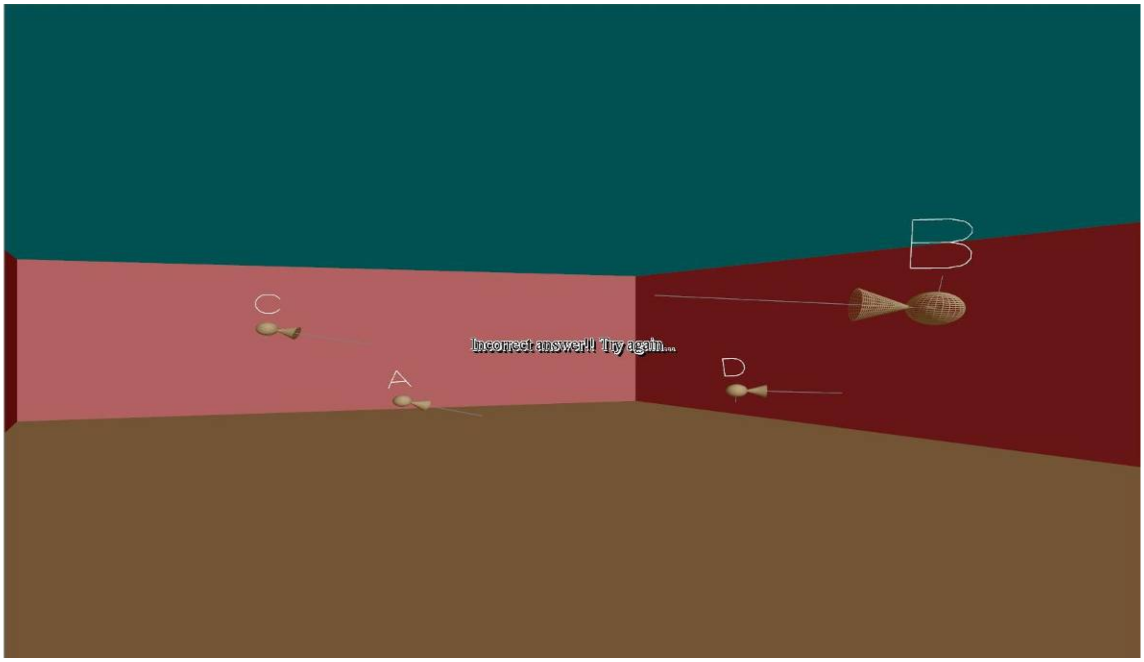


Figure 26: FIR D (10,000 reflections, 10,000 paths)



Incorrect answer!! Try again...

Figure 27: TDS Running in Testing Mode with Feedback

Experiment Analysis

Resolving the validity of the algorithms, though important, certainly does not constitute a sufficient analysis alone. We must also consider the objective effectiveness of each algorithm method compared to the others. In order to analyze this, the simulator contains a set of testing modes for subject-based experiments.

Regardless of the mode for the experiment, the simulator presents the test subject with between two and seven sound sources all located within a room. The scenario randomly places and orients the sound sources so that they all lie within a certain tolerance away from any walls and each other. Each sound source uses the same visual representation of a sphere and cone, which helps distinguish the placement and direction of the object. We also place an associated letter above each sound object. Figure 27 illustrates the experiment in action. The simulator randomly selects one of the objects to play the sound through one of the randomly selected algorithms. A test subject must to select which object is playing the sound, based on what he hears by pressing the appropriate key that corresponds to the object. The section "Research Methods" on page 93 details this procedure extensively and the appendix "Experiment Documents" on page 158 includes a typical transcript used in the experiment.

Testing with Feedback

As mentioned earlier, the simulator provides a testing mode with feedback (see Figure 27). In this mode, the test subject must eventually select the correct sound source from the available visible objects. The subject may guess as often as he wishes. After each guess, the simulator politely informs the subject if correct or not. If the student wrongly guessed, the simulator provides another attempt at the question.

Testing with Selection

Additionally, the simulator can perform testing on subjects without providing feedback. This is just a simplified version of the previous testing technique, except that the subject has only one attempt at each question and the simulator does not inform him of how he performed. Once answered, whether correct or not, the simulator moves directly on to the next question in the experiment.

Experiment Setup and Data

As explained previously the simulator provides two methods of testing: with feedback and without feedback. We ask each subject to sit through ten scenarios, or questions of each mode, first with and then without feedback (see "Testing Details" on page 99). The simulator then records the results and the stores the data for later analysis. We must emphasized that the testing environment should be considerable comfortable, quiet, and free of external distractions, with a base sound level of 70dB or less.

We tested 110 students in the course of eight days. All tests occurred in the same room, using the same computer system and setup described preciously (see Table 14 on page 97), and with consistent instructions and procedures. All care and effort ensued to maintain a consistent environment for the duration of the experiment.

Scoring Questions for Statistical Analysis

Analysis of the portion of the experiment with feedback proves a bit trickier than the data without feedback. This assertion follows from the fact that each question consists of more than just a correct or incorrect guess. Rather, the subject eventually must select the correct answer and can take many guesses to do so. The principal component variables are how many attempts he took to arrive at the correct result as well as the number of sound sources from which to select.

$$Score = \left(\frac{1 + [NumSources\ ] - [NumAttempt\ s]}{[MaxNumSour\ ces]} \right) * \left(\frac{[NumSources\ ]}{[MaxNumSour\ ces]} \right)$$

Equation 3: Accuracy Score for Testing WITH Feedback

For example, consider a test that has six sound sources and the subject selects correctly on the second try. He should score better compared to someone guessing correctly on the fifth attempt, with the same number of sources. Furthermore, if a subject succeeds on the second try with four sound sources, he should score lower than a subject that also answered correctly on the second guess but with only three sound sources.

Our experiment allowed for two to seven sound objects per question. Equation 3 shows how we calculated the accuracy score for Figure 17 and Figure 18 as well as the data from the final experiment. The left portion of the formula computes the relative difference between the number of sources and the number of attempts, while the right side simply scales the score based on the number of sources.

NumAttempts	NumSources	Score	NumAttempts	NumSources	Score
1	2	0.082	7	7	0.143
1	3	0.184	6	7	0.286
1	4	0.327	5	7	0.429
1	5	0.510	4	7	0.571
1	6	0.735	3	7	0.714
1	7	1.000	2	7	0.857
2	2	0.041	1	7	1.000
2	3	0.122	6	6	0.122
2	4	0.245	5	6	0.245
2	5	0.408	4	6	0.367
2	6	0.612	3	6	0.490
2	7	0.857	2	6	0.612
3	3	0.061	1	6	0.735
3	4	0.163	5	5	0.102
3	5	0.306	4	5	0.204
3	6	0.490	3	5	0.306
3	7	0.714	2	5	0.408
7	7	0.143	1	5	0.510
6	6	0.122	4	4	0.082
5	5	0.102	3	4	0.163
4	4	0.082	2	4	0.245
3	3	0.061	1	4	0.327
2	2	0.041	3	3	0.061
3	2	0.000	2	3	0.122
7	2	0.000	1	3	0.184
7	6	0.000	2	2	0.041
10	7	0.000	1	2	0.082

Table 19: Example Score Calculations

Table 19 illustrates some of the extreme example results using this same equation with a maximum number of sources set to seven. Note that if the subject takes the same number of attempts as the number of sources, he will score the lowest possible computed value, in this case 0.041 for two attempts with two sound sources. Not expressed in Equation 3, we reserve the option to allow only a minimum score of zero; thereby normalizing questions where the subject took more attempts than the number of sources. This accounts for questions where the subject was possibly confused or disoriented. Finally, we can easily generalize Equation 3 by multiplying the final score by one if the subject eventually answered the question correctly or by zero otherwise. This modification allows us to use the same formula for questions with or without feedback. The scores for questions without feedback will always be zero or come from the first group in Table 19 since that mode only allows one attempt per question.

Statistical Analysis

Due to the complicated nature of the experiment design, common statistical analysis methods fail to provide meaningful feedback and results. Therefore, we have chosen to provide an extensive examination utilizing multiple approaches in the statistical analysis of the data. We include justifications, assumptions, shortcomings, and detailed analyses for each methodology. Unless otherwise noted, we parsed the data through IBM® SPSS® Statistics 22 (64bit) on the same machine used in the experiment. We also divided the data into three groups for analysis separately: Data With Feedback, Data With Feedback (minzero), and Data Without Feedback.

Data Structure Analysis

In order to analyze data from any experiment, one must always carefully consider the data with respect to both meaning and conformity. Many statistical analysis procedures require restructuring, consolidation, grouping, weighting, and often the elimination of portions of the initial data. These requirements can dramatically vary depending on the statistical procedure and the types of variables involved.

In an attempt to keep some portion of this analysis simple and straightforward, we have limited the conversation to only the variables directly relevant to the three hypotheses for this experiment. Table 20 lists the primary variables recorded during subject testing. We use just the first three variables (ALGORITHM versus SCORE or ALGORITHM versus

SCOREMINZERO) in our analysis. Most of the remaining variables directly factor into SCORE or SCOREMINZERO or do not have meaning for the questions derived from the three hypotheses. Furthermore, as appropriate, we reserve the option to remove the scores for the SIMPLE ALGORITHM (or no 3D acoustic algorithm). These samples account for only about 6% of the data and generally lend nothing valuable to the discussion. We will expound on any further modifications to the original dataset during the analysis.

Variable	Type	Range of Values
ALGORITHM	Categorical	SIMPLE, DIRECTX, DIRECTPATHS, BOUNCINGREFLECTIONS (1 to 4)
SCORE	Interval	0.0 to 1.0 (possibly less than 0.0)
SCOREMINZERO	Interval	0.0 to 1.0
DEPTH	Ordinal	1 to 10 (only affects complexity of BOUNCINGREFLECTIONS)
BREADTH	Ordinal	1 to 10 (only affects complexity of BOUNCINGREFLECTIONS)
SOUNDFILE	Categorical	"bounce.wav" or "dogbark.wav" (1 or 2 respectively)
NUMBER_OF_SOURCES	Ordinal	1 to 7 (score directly takes this variable into account)
ATTEMPTS	Interval	1 to 7+ (score directly takes this variable into account)
GUESSEDCORRECTLY	Categorical	T/F (only affects the data with no feedback)
ORIENTATIONWEIGHT	Ordinal	0.0 to 1.0 (affects the proportional directionality)

Table 20: Breakdown of Experiment Data Variables

Repeated Measures

One of the two chief complications to properly analyzing the data of this experiment results directly from a dilemma of the experiment design. In order to optimize time and resources, we asked each subject, or student, to answer multiple versions of the same questions. In statistical terms, we can call this a “repeated measures” model. As we shall see, this can bias the analysis due to individual preference weighting. Consider an extraneous example for illustration purposes. One can ask ten subjects a single simple question, such as a list of their three favorite colors. Alternatively, one could also ask ten subjects to rank three specific colors with respect to their preference. In each case, we have thirty data points. We would have to analyze the data differently, however. In the second dataset, for each subject, the score becomes weighted or biased due to the comparative nature of the question. One subject might rate the three colors with a range from 7 to 10, while another might allow for a wider spread, say from 2 to 8. We cannot simply list all 30 answers and analyze them linearly as we would with the data for a single question per user (top three colors). Rather, we must group each subject's responses together and examine the relative rankings of data groups. Both the Friedman Test and ANOVA test handle repeated measures data.

This question now follows. Does the design of this experiment require a repeated measures test? Unfortunately, we cannot concretely answer this. Since we ask each student the same question ten times (and then ten times again for questions without feedback), we must at least contemplate the data in repeated measures terms. We can certainly see an argument for using a repeated measures analysis method. However, we must also consider, do we ask the same question repeatedly? Each question has many randomized variables that can affect the uniformity of the question. We widely vary the number of sources, locations and orientations of source, sound file played, complexity level of the algorithm, and especially algorithm used to produce the sound. Even the number of questions for each algorithm per subject varies dramatically. Thus, we must ponder if one student's score, on average, differs from another student's with respect to both range and performance. We will further explore this later, but it is difficult, if not impossible to verify this deterministically. Therefore, we reserve the option to analyze the data using both repeated measures and standard methods and compare the approaches.

Distribution

In addition to the repeated nature of the data, we must carefully consider distribution in our analysis. Statisticians consider data normally distributed when a normal bell curve bounds the majority of data points. Ideally, we would hope the data conforms to a normal, or Gaussian distribution model. Since the most common and numerous statistical analysis procedures assume a normal distribution of the data, evaluation would follow a much simpler and straightforward manner if our data does not break this rule. Unfortunately, as often is the case with real-life data, initial assessment and expanded analysis of normality shows a non-Gaussian tendency, across all datasets. If the data does not have a normal distribution, we must use non-parametric test models for analysis. Figure 28 visually illustrates the histograms of the scores in the three sets of data. We must keep in mind that these calculations do not factor in the repeated measures nature of the experiment. Therefore, the data might still have a normal distribution, contrary to the given analysis. We will expand on this later.

Friedman Test (Non-Parametric, Repeated Measures)

Perhaps the most obvious choice for analysis, we start with the Friedman test (Friedman, 1937). Developed by Milton Friedman in the 1930's, this test analyzes data trends by comparing the rankings across groups instead of the actual scores. In measuring rankings, this approach compensates for the natural bias encountered when presenting the same

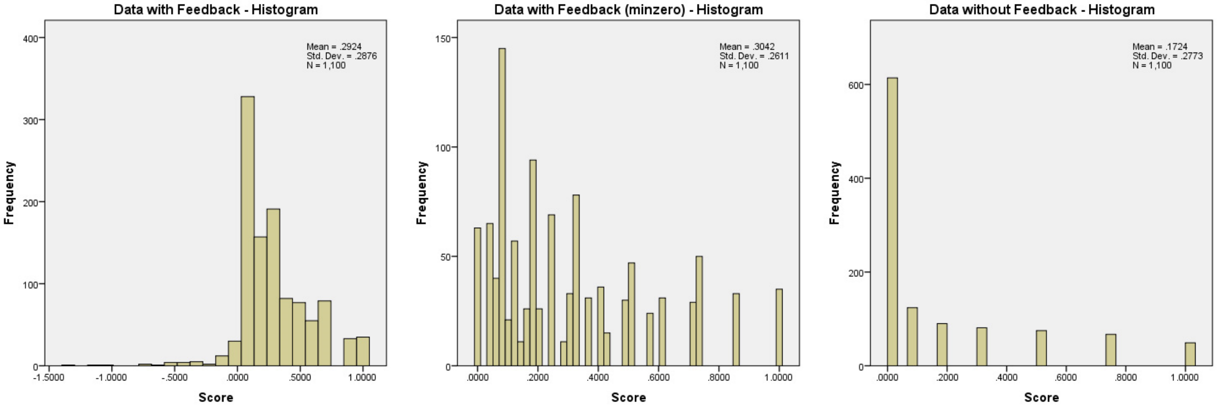


Figure 28: Linear Histograms for Datasets

question multiple times to each subject. At first glance, the Friedman test should manage all concerns with the experiment data. It allows for data that does not conform to a Gaussian distribution and specifically aims for repeated measures. However, in order to run the test, we must transform the data into symmetrical tables grouped by subject. To do this, we can average the scores per algorithm per subject. Table 21 illustrates this transformation for a portion of the data for questions without feedback.

Original Data		
Subject	**Score**	**Algorithm**
1	0.3265	4
1	0.1837	2
1	0.0000	2
1	0.0000	4
1	0.0816	1
1	0.0816	3
1	0.0000	4
1	0.0000	2
1	0.5102	2
1	0.0000	4
2	0.0000	4
2	0.0000	2
2	0.0000	2
2	0.3265	2
2	0.0000	4
2	0.0000	4
2	0.0816	4
2	0.3265	3
2	0.0000	1
2	0.0816	4
3	0.0000	2
3	0.0000	2
3	1.0000	3
3	0.3265	3
3	0.0816	2
3	0.0000	2
3	0.3265	3
3	0.0816	4
3	0.0816	2
3	0.0000	2

Transformed Data				
Subject	**Alg1**	**Alg2**	**Alg3**	**Alg4**
1	0.082	0.173	0.082	0.082
2	0.000	0.109	0.327	0.033
3		0.027	0.551	0.082
4	0.000	0.054	0.755	0.046
5		0.000	0.170	0.082
6	0.000	0.027	0.073	
8		0.000	0.433	0.510
9	0.000	0.510	0.118	0.245
10		0.000	0.066	0.000
11		0.046	0.054	0.109
12		0.337	0.109	0.361
13		0.041	0.306	0.212
14		0.000	0.000	0.066
15		0.041	0.314	0.082
16		0.111	0.000	
17		0.128	0.020	0.041
18		0.000	0.255	0.184
19	0.255	0.000	0.000	0.291
20	0.000	0.027	0.418	0.088

Table 21: Data Transformation for Friedman Test

The left side of Table 21 portrays the original data from the first three subjects. Using pivot tables in Excel, we transform the data into the right side by calculating the average score for each algorithm presented to each subject. This tactic essentially allows us to employ the Friedman test on the data, with some further refinements and assumptions, detailed subsequently.

Limitations

Table 22 lists the assumptions and limitations for each of the various approaches discussed below. With respect to the Friedman test, we must first address the maneuver of taking averages in the transformation. Most statisticians would consider this approach somewhat questionable for multiple reasons. By definition, an average naturally suffers from errors (i.e. the mean-squared error), thus the basis for these more advance statistical methods. Blindly taking averages before a statistical analysis does not conform to best practices. Therefore, we proceed with careful consideration. If each subject answered the same number of questions per algorithm, then we might justify using averages without much concern. Unfortunately, with the given dataset, in order to conform to this statistical test, we must group by averages of varying quantities. Fortunately, in terms of the number of tests per subject per algorithm, the variance of quantity does not contrast significantly relative to the number of algorithms.

Next, we must consider that the experiment did not force the same number of questions for each algorithm per subject. Before each question, the simulator randomly selects which of the four algorithms to employ. In fact, during this selection, if it picks SIMPLE, the simulator will try again, thus reducing the probability of SIMPLE from about 25% to 6.25% ($p = 0.25^2$). Even removing these samples from the data (which we do) it is still very common for subject to have different numbers of questions for the various algorithms. Since we only effectively ask ten questions per subject, obtaining meaningful commonality by number of algorithms per user within the data proves almost impossible.

Test	Assumptions/Limitations	Justification/Test	Effects/Results/Concerns
Non-Parametric, Repeated Measures Tests:			
Friedman			
	Grouping by averages	Small variations	Questionable method
	Removal of SIMPLE samples	SIMPLE has no meaning to hypotheses	6.25% of the questions
	Remove subjects with missing data	Data integrity	7.8% data lost
	Omnibus test (non-specific results)		
Wilcoxon Signed-Rank			
	Same as Friedman		
	Type I errors	Bonferroni adjustment	Possibly overly sensitive (too restrictive)
	Post-hoc for Friedman tests		
Parametric, Repeated Measures Test:			
ANOVA			
	Sphericity	Mauchly's Test	All data passed sphericity test
	Gausian distribution	Shapiro-Wilk Test	Residual data failed normality test
Parametric, Independent Tests:			
Kruskal-Wallis H			
	Non-repeated measures data	Dissimilar questions	Questionable data
	Only positive scores allowed	Inherent to test	Remove "Data With Feedback"
	Omnibus test (non-specific results)		
Mann-Whitney U			
	Same as Kruskal-Wallis H		
	Independence of observations	Distinct questions	Technically repeating participants
	Similar distribution shapes	Visual inspection	Compare mean ranks vs. medians
	Type I errors	Bonferroni adjustment	Possibly overly sensitive (too restrictive)
	Post-hoc for Kruskal-Wallis H tests		

Table 22: Assumptions and Justifications for Statistical Tests

Subsequently, with such a limited number of questions per topic relative to the number of possible algorithms, the simulator did not expose some subjects to one or more algorithms. This poses a similar problem to the previous discussion. The Friedman test requires data in each cell and we cannot simply replace null values with zeros and maintain data integrity. In Table 21, subjects 6 and 16 did not encounter the fourth algorithm; therefore, we remove them entirely from this analysis. Fortunately, these aberrations account for less than 8% of the data, so we feel reasonably justified in subtracting these records.

Analysis

The Friedman test allows us to determine if a significant difference exists in the mean ranks of scores between the various algorithms. It will not formally rank the performance of the algorithms. Rather, it falls in the category of an "omnibus test" and only tells us if groups differ significantly.

Hypothesis:	**Alg2=Alg3=Alg4**		**Alg2=Alg3**		**Alg3=Alg4**		**Alg2=Alg4**		**Conclusion:**
Dataset	p	χ2	p	χ2	p	χ2	p	χ2	
With Feedback	0.007	9.789	0.001	11.89	0.170	1.885	0.626	0.238	Alg2 does not equal Alg3
With Feedback (minzero)	0.026	7.303	0.003	8.824	0.239	1.385	0.626	0.238	Alg2 does not equal Alg3
Without Feedback	0.043	6.309	0.017	5.688	0.753	0.099	0.099	2.723	Alg2 does not equal Alg3

Table 23: Freidman Test P Value Summary

We will begin with the null hypotheses that the mean rank of ALGORITHM 2 equals the mean rank of ALGORITHM 3, which also equals the mean rank of ALGORITHM 4. We will use the level of significance of $p < 0.05$ to evaluate our results. If we end up with a test statistic of a P value less than 0.05, we will reject the null hypothesis and conclude that the data has significant differences of the mean scores between the tested groups. This will allow us to proceed with a post-hoc analysis for pairwise comparisons. Otherwise, we will accept the null hypothesis and say that we found no measurable difference over the tested algorithms.

For an initial pot hoc analysis, we will again run the Friedman test between the paired combinations of the algorithm groups. This, again, will allow us to find significant differences in the rank means of the scores by algorithms. Table 23 summarizes the results of the various tests. Cells shaded gray indicate P values of less than 0.05, thus allowing us to reject the null hypothesis with a 95% confidence interval. We can then preliminarily conclude that a statistical difference exists between ALGORITHM 2 and ALGORITHM 3.

Post-Hoc Analysis

The Friedman test showed us that a significant difference exists between the mean scores of the algorithms, and more specifically that ALGORITHM 2 and ALGORITHM 3 notably differ. We cannot conclude anything more about the other algorithm combinations and more importantly, we do not know which algorithm performed better. For this, we must run a post-hoc test. In order to perform a proper post-hoc analysis for non-normal, repeated measures data, we will use the Wilcoxon signed-rank test. This test is the non-parametric version of the dependent t-test for related groups.

We will run the Wilcoxon signed-rank test for all algorithm score combinations (ALGORITHM's 2 vs. 3, 3 vs. 4, and 2 vs. 4), for the moment ignoring the preliminary post-hoc Friedman tests above. To avoid Type I errors that would arise because we are making multiple comparisons between data groups, we use the Bonferroni adjustment. To do this, we simply find a new significance level by dividing the previous level ($p = 0.05$) by the number of combinations ($df = 3$). This gives us a stricter significance level of $p < 0.017$. This adjustment can lead to an overly sensitive test, proportional to the number of between-groups compared, but it remains sufficient for the small degrees of freedom in this analysis.

Hypothesis:	**Alg2=Alg3**		**Alg3=Alg4**		**Alg2=Alg4**		**Median Scores**		
Dataset	p	Z	p	Z	p	Z	Alg2	Alg3	Alg4
With Feedback	0.0003	-3.556	0.1132	-1.584	0.2465	-1.159	0.2313	0.2857	0.2755
With Feedback (minzero)	0.0010	-3.282	0.1894	-1.312	0.1830	-1.331	0.2449	0.2925	0.2755
Without Feedback	0.0026	-3.008	0.2156	-1.238	0.1176	-1.565	0.0816	0.2031	0.1327

Table 24: Wilcoxon Signed-Rank Test P Value Summary

Table 24 lists the statistical result values for the different datasets. We again find the same general results across the three datasets. ALGORITHM 3 scored significantly higher than ALGORITHM 2 globally. Specifically, we conclude the following.

We found a statistically significant difference between algorithm scores using the Friedman test: With Feedback $\chi^2(2) = 9.79$, $p = 0.007$; With Feedback (minzero) $\chi^2(2) = 7.30$, $p = 0.026$; Without Feedback $\chi^2(2) = 6.31$, $p = 0.043$. Table 24 lists the mean scores for the algorithms by dataset. We conducted a post-hoc analysis using Wilcoxon signed-rank tests with a Bonferroni correction applied. After modifying the significance level to $p < 0.017$, we found no significant differences between ALGORITHMS 3 and 4 (With Feedback, $Z = -1.58$, $p = 0.113$; With Feedback (minzero), $Z = -1.31$, $p = 0.189$; Without Feedback, $Z = -1.24$, $p = 0.216$). We also found no significant differences between ALGORITHMS 2 and 4 (With Feedback, $Z = -1.16$, $p = 0.247$; With Feedback (minzero), $Z = -1.33$, $p = 0.183$; Without Feedback, $Z = -1.57$, $p = 0.118$). However, we did find that ALGORITHM 3 scored significantly higher than ALGORITHM 2 for the three datasets (With Feedback, $Z = -3.56$, $p = 0.000$; With Feedback (minzero), $Z = -3.28$, $p = 0.001$; Without Feedback, $Z = -3.01$, $p = 0.003$).

ANOVA Test (Parametric, Repeated Measures)

Similar to the previously described Freidman test, we can also consider one-way repeated measures ANOVA tests on the datasets. For this test, we must again evaluate the data in the condensed model describe in Table 21. We perform this test as well as the non-parametric repeated measures test above because we did not conclusively prove that the data in the condensed form does not violate the normality assumption required for ANOVA. In order to use this test, though, we must first analyze the datasets for violations of three assumptions: normal distribution, no significant outliers, and sphericity.

Sphericity

As one of the primary assumptions for running one-way repeated measures ANOVA tests, we must check for sphericity in the data. Sphericity is defined as the "the condition where the variances of the differences between all combinations of related groups are equal" (Lund & Lund, 2014). Fortunately, the test for sphericity is far less complicated than the definition. We simply run Mauchly's Test of Sphericity while probing for the ANOVA results. The null hypothesis in this case holds that the variances of the differences are equal, or the data does not violate the sphericity rule. Therefore, *P* values greater than or equal to 0.05 allow us to assume sphericity.

Mauchly's Test of Sphericity shows the data did not violate the assumption of sphericity for all three datasets: With Feedback $\chi^2(2) = 4.60$, $p = 0.100$; With Feedback (minzero) $\chi^2(2) = 2.31$, $p = 0.315$; Without Feedback $\chi^2(2) = 0.167$, $p = 0.920$. Therefore, we may proceed with the ANOVA test with respect to the sphericity assumption.

Normal Distribution

In an attempt at thoroughness, we again consider the distribution of the data. Figure 11 shows histograms of the original, unmodified datasets. It does not describe the condensed, combined dataset used in the repeated measures analyses. In order to determine normality, we run the ANOVA tests in SPSS and examine the standardized residuals of each factor for normality. In all three datasets, for each of the three algorithm groups, the residual data failed the Shapiro-Wilk test for normality. Given this, and our previous analysis, we conclude that we cannot reliably employ the one-way repeated measures ANOVA test for the given datasets.

Kruskal-Wallis H Test (Non-Parametric, Independent)

Previously, we alluded to the idea that the experiment design and subsequent data might not maintain an entirely repeated measures nature. The simulator presented each subject with two sets of ten questions. These tasks each had a notable random and inconsistent nature that forces us to consider the data through a standard, linear model. Therefore, in addition to the preceding repeated measures tests we performed, we also include this short analysis using simple, non-grouping statistical measures. The final analysis must include all considerations, shortcomings, and limitations addressed in this document. Table 22 summarizes the different tests and issues encountered with each model.

We have discussed at length the non-Gaussian nature of the data, both in repeated measures and linear (Figure 11) terms. In no form does the data portray any hint of a Gaussian, or normal distribution natures, so we will not evaluate the data using simple parametric statistical tests. Rather, we will employ the Kruskal-Wallis H test in a similar fashion as the Friedman test above, then, if results permit, utilize the post-hoc Mann-Whitney U test for further refinement.

As it turns out, the Kruskal-Wallis H test in SPSS does not allow negative values of scores. In the three datasets, only the With Feedback set allows for negative scores. This aberration occurs when a subject guesses more times than the number of sound sources in the environment. We generally attribute this to extreme confusion of the subject and consider the question having a failing answer. To better account for this, we created the third dataset, With Feedback (minzero), which simply sets the negative values to zero. We could certainly transform or normalize the data of all sets (for all tests) to remove the negative values, however this could unnecessarily weaken the intention of the scoring formula. Instead, we simply remove the offending dataset from this part of our analysis and allow the dataset With Feedback (minzero) to stand alone.

Analysis

Like the Friedman test, the Kruskal-Wallis H test allows us to determine if a significant difference exists in the mean ranks of scores between the various algorithms. It will not formally rank the performance of the algorithms. Rather, it is an "omnibus test" and only tells us if groups differ significantly.

We will begin with the null hypotheses that the mean rank of ALGORITHM 2 equals the mean rank of ALGORITHM 3, which also equals the mean rank of ALGORITHM 4. We will use the level of significance of $p < 0.05$ to evaluate our results. If we end up with a test statistic of a P value less than 0.05, we will reject the null hypothesis and conclude that the data has significant differences of the mean scores between the tested groups. This will allow us to proceed with a post-hoc analysis for pairwise comparisons. Otherwise, we will accept the null hypothesis and say that we found no measurable difference over the tested algorithms.

Hypothesis:	Alg2=Alg3=Alg4		Alg2=Alg3		Alg3=Alg4		Alg2=Alg4		Conclusion:
Dataset	p	χ^2	p	χ^2	p	χ^2	p	χ^2	
With Feedback (minzero)	0.019	7.959	0.006	7.640	0.297	1.090	0.078	3.098	Alg2 ≠ Alg3
Without Feedback	0.000	24.808	0.000	24.123	0.049	3.864	0.002	9.568	Alg2 ≠ Alg3, Alg2 ≠ Alg4

Table 25: Kruskal-Wallis H Test P Value Summary

For an initial pot hoc analysis, we will again run the Kruskal-Wallis H test between the paired combinations of the algorithm groups. This, again, will allow us to find significant differences in the rank means of the scores by algorithms. Table 25 summarizes the results of the various tests. Cells shaded gray indicate P values of less than 0.05, thus allowing us to reject the null hypothesis with a 95% confidence interval. We can then preliminarily conclude that a statistical difference exists between ALGORITHM 2 and ALGORITHM 3 in both datasets. We will further evaluate the comparisons in the post-hoc analysis, paying careful attention to the results for the Without Feedback dataset.

Post-Hoc Analysis

The Kruskal-Wallis H test showed us that a significant difference exists between the mean scores of the algorithms, and more specifically that ALGORITHM 2 and ALGORITHM 3 notably differ. For the Without Feedback dataset we noted significant differences in each algorithm combination. We cannot conclude anything more about the other algorithm combinations and more importantly, we do not know which algorithm performed better. For this, we must run a post-hoc test. In order to perform a proper post-hoc analysis for non-normal, repeated measures data, we will use the Mann-Whitney U test. This test is the non-parametric version of the independent-samples t-test.

We will run the Mann-Whitney U test for all algorithm score combinations (ALGORITHM's 2 vs. 3, 3 vs. 4, and 2 vs. 4), for the moment ignoring the preliminary post-hoc Kruskal-Wallis H tests above. To avoid Type I errors that would arise because we are making multiple comparisons between data groups, we use the Bonferroni adjustment. To do this, we simply find a new significance level by dividing the previous level (*p* = 0.05) by the number of combinations (*df* = 3). This gives us a stricter significance level of *p* < 0.017. This adjustment can lead to an overly sensitive test, proportional to the number of between-groups compared, but it remains sufficient for the small degrees of freedom in this analysis.

Hypothesis:	**Alg2=Alg3**		**Alg3=Alg4**		**Alg2=Alg4**		**Median Ranks**		
Dataset	p	Z	p	Z	p	Z	Alg2=Alg3	Alg3=Alg4	Alg2=Alg4
With Feedback (minzero)	0.006	-2.76	0.297	-1.04	0.078	-1.76	331 < 374	333 ≈ 341	370 ≈ 343
Without Feedback	0.000	-4.91	0.049	-1.97	0.002	-3.09	312 < 380	349 ≈ 322	327 < 369

Table 26: Mann-Whitney U Test P Value Summary

Table 26 lists the statistical result values for the two datasets. ALGORITHM 3 scored significantly higher than ALGORITHM 2 globally. In the Without Feedback dataset, ALGORITHM 4 scored notably better than ALGORITHM 2. Specifically, we conclude the following.

We found a statistically significant difference between algorithm scores using the Kruskal-Wallis H test: With Feedback (minzero) $\chi^2(2) = 7.96$, *p* = 0.019; Without Feedback $\chi^2(2) = 24.81$, *p* = 0.000. Table 26 lists the comparable mean ranks for the algorithm combinations by dataset. We conducted a post-hoc analysis using Mann-Whitney U tests with a Bonferroni correction applied. After modifying the significance level to *p* < 0.017, we found no significant differences between ALGORITHMS 3 and 4 (With Feedback (minzero), *Z* = -1.04, *p* = 0.297; Without Feedback, *Z* = -1.97, *p* = 0.059). We also found no significant differences between ALGORITHM 2 and ALGORITHM 4 for the With Feedback (minzero)

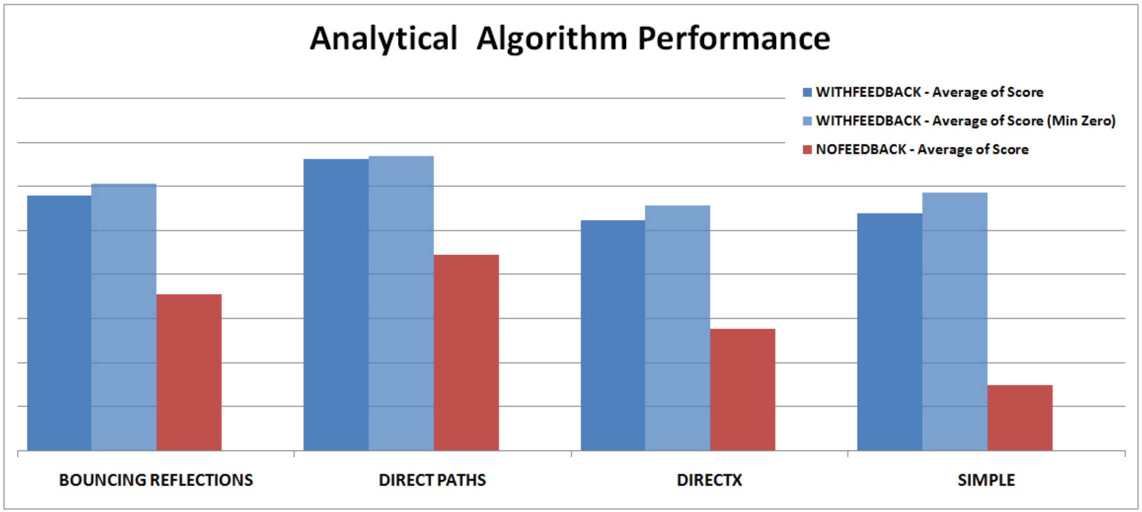


Figure 29: Comparable Analytical Analysis of Algorithms

dataset, *Z* = -1.76, *p* = 0.078. Conversely, the dataset Without Feedback showed a notable difference between ALGORITHM 2 and ALGORITHM 4, *Z* = -3.09, *p* = 0.002. Furthermore, we found that ALGORITHM 3 scored significantly higher than ALGORITHM 2 for both datasets (With Feedback (minzero), Z = -2.76, p = 0.006; Without Feedback, Z = -4.91, p = 0.000).

Analytical Analysis

Though not as meaningful technically as the statistical analyses in the previous sections, we can also consider a more mundane and aesthetically analytical approach to understanding the data. Figure 29 illustrates this tactic. Here, we simply graph the average of the scores for each algorithm using Equation 3, breaking them down by question type. This chart incorporates no statistical refinement except for general averages. Clearly, we cannot directly compare questions with feedback to those without feedback as they have different proportions and population numbers. However, previously predicated trends become quickly apparent and unambiguous. We will further consider the analysis of this breakdown in the next section.

Additional evaluation in this manner can also characterize the temporal nature of the experiment. According to our second hypotheses (see Table 13 on page 95), we anticipate that over time as a student becomes more familiar with the simulator and the algorithms, performance should correspondingly improve. There is nothing revolutionary about this assertion, but the real question remains, how much time would this process take? Figure 30 illustrates a basic and initial analysis of this question using the approach just described. Without performing overly complex and extensive statistical calculations that would certainly require a third-party analysis program and possibly innovative statistical techniques therein, we have generated a snapshot of the algorithm performance over time for the experiment. This chart dissects the average score for each algorithm during the course of the experiments.

We expect to see increasing trends in some or all of the six groups in Figure 30. Unfortunately, arguably, this is not the case and therefore we cannot conclude that training improves performance for the limited number of questions per subject in this experiment. Considering we only ask 20 questions per student and each of these can be any of the four

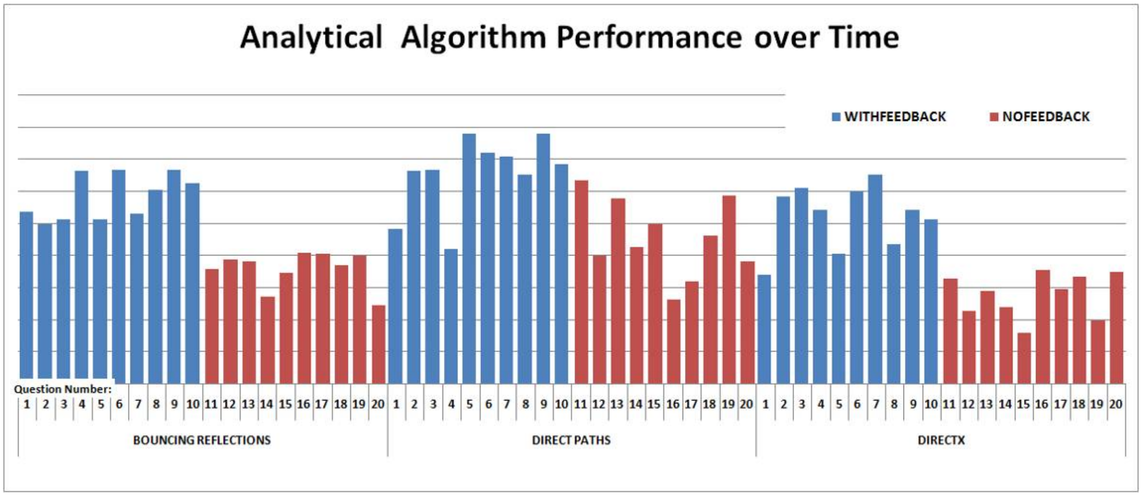


Figure 30: Comparable Analytical Analysis of Algorithms over Time

algorithm possibilities the inconclusiveness of this result does seem reasonable. We will discuss this further in the following section. However, we tentatively assert that we will need to design future experiments in which we ask more questions of each student with fewer variables at play.

Evaluation

To a great extent, from the explanations in the previous sections, we can rank the four algorithms in order of effectiveness for this experiment. In every instance, whether with feedback or without, allowing negative scores, or employing a scoring function or not, we can unquestionably say that the Direct Paths algorithm algorithms developed for this research outperformed the DirectX® (and Simple) algorithm. Notably, the Direct Paths achieved significantly better results than the Bouncing Reflections algorithm in many facets of the earlier analyses. In the following subsections, we will look at each algorithm and then consider the three hypotheses for this experiment listed in Table 13 on page 95.

Direct Paths Algorithm Evaluation

Unquestionably, and even somewhat surprisingly, the Direct Paths algorithm came out the clear winner in this experiment. Insofar as our analysis provided, against the control, or the Simple Algorithm, the Direct Paths algorithm maintained superior results, as one would expect. Not surprisingly, we also see that this algorithm as well as the other research algorithm performed much better than the industry standard DirectX algorithm. We further consider this when we evaluate the DirectX algorithm.

Notably, though, the Direct Paths algorithm scored significantly better than the other research algorithm, Bouncing Reflections. As the Direct Paths algorithm mainly consists of a dramatically simpler subset of the more robust Bouncing Reflections algorithm, we expected the opposite result. The Direct Paths algorithm, as exemplified in Table 1 on page 32, simply computes the zero-order reflection paths, or the two straight paths from the sound source to the ears. Expanding this, the Bouncing Reflections algorithm listed in Table 5 on page 40 initially uses this same code for the first paths in the list of impulses it retains. Therefore, we logically expected the more completed Bouncing Reflections algorithm to outperform the basic Direct Paths algorithm. We further consider this and the implications in the section “Hypothesis Three Evaluation” on page 136.

Bouncing Reflections Algorithm Evaluation

For the same reasons as the Direct Paths algorithm, the Bouncing Reflections algorithm arguably surpassed both DirectX and Simple. With the exception of the repeated measures statistical analysis we noted significant performance distinctions. Though we did not find the numbers as dramatically supportive in for this case, they still left little room for question about the relative execution.

With a somewhat loose level of confidence, we can statistically conclude that the Direct Paths algorithm performed better in this experiment than the Bouncing Reflections algorithm. We must however, emphasize that this level of confidence falls in the 90% certainty range and will require further analysis and experimentation to evaluate properly. However, the simple fact that the results seem so one-sided, regardless of the question type or analysis method, gives us pause to consider why. We will further consider these implications in the section “Hypothesis Three Evaluation” on page 136.

DirectX Algorithm Evaluation

Generally, we should find that including no 3D algorithm as we did with the Simple algorithm should return statistically inferior results than any algorithm with some 3D virtualization nature. The only exception to this would be when the 3D sound algorithm in question generates misleading or blatantly incorrect acoustic reproductions. Clearly, considering the data from the experiment, we have cause to have some trepidation about the efficacy of the DirectX algorithm. As we discuss in sections “SoundPlayDirectX Algorithm Object” on page 61 and “DirectSound® Library” on page 83, this library from Microsoft® has numerous flaws and deficiencies. Our system attempted to overcome these issues, but it could only compensate so much. We have no statistical basis to say that DirectX is much better than no algorithm at all. To be sure, based on this experiment we cannot state the inverse, as the numbers do not lead to any certainty. However, we find this ambiguity concerning to say the least.

Clearly, though, the experimental algorithms both overwhelmed the DirectX algorithm with respect to accuracy. We can say this in both cases with an extremely high level of statistical certainty. As programmers, we appreciate the low-level access to the sound card that this library provides. Yet does this merit the promotion and institution of this algorithm as the present industry standard? Even the most comparable of our algorithms, Direct Paths, spectacularly surpassed this library’s capabilities. Both the Direct Paths algorithm and the DirectX algorithm only calculate zero-order reflection paths and ignore all room and other environmental geometry. So how could the Direct Paths algorithm perform so much better than the DirectX algorithm? Arguably, the difference resides in the poor implementation of DirectX by Microsoft® and the industry’s lack of focus on development of any proper 3D sound virtualization algorithm.

Hypothesis One Evaluation

The first hypothesis of this research states (from Table 13 on page 95): “While DirectX DirectSound® will perform better than Simple playback, the 3D algorithms from the research will produce more accurate sound reproduction than DirectX DirectSound®.”

In the previous section, we demonstrated the second part of this statement, while rejecting the first half. Based on the statistical analyses and even anecdotal evidence, the DirectX DirectSound® library has major flaws. Simply stated, we have no evidence to support that the DirectX algorithm conclusively performs better than the Simple algorithm (which uses no 3D acoustic virtualization). In some cases, though not with any statistical significance, the Simple algorithm actually did better than the DirectX algorithm!

Undeniably, however, the Direct Paths algorithm and the Bouncing Reflections algorithm readily outstripped the DirectX algorithm in terms of acoustic reproduction performance. This, again, we do not find surprising given the notable flaws inherent in the DirectSound DirectX® library (see “DirectSound® Library” on page 83).

As with any good hypothesis, this evaluation brings up more questions than answers. While DirectX DirectSound® did not perform better than Simple playback, the 3D algorithms from the research did produce more accurate sound reproduction. So, is the next step to improve DirectX DirectSound® (now called X3DAudio®) or further develop one of our algorithms? Should we analyze what components of the algorithm led to this substantial failure or leave this to the industry to self-correct? Does the Microsoft® algorithm truly fall so short than no 3D algorithm, even considering that anecdotally it seems to improve virtualization to some degree in simple 3D spaces? How would performance for the DirectX

DirectSound® library change in more complicate or even open-space environments? Unfortunately, we must leave these considerations for future experiments and analysis.

Hypothesis Two Evaluation

The second experimental hypothesis states: "Over time, with training, a test subject's performance with the research algorithms will improve, while training with the other algorithms will not show improvement."

In section "Analytical Analysis" on page 128, we loosely considered this concept via Figure 30. This chart, though not purely statistical in nature effectively allows us to visualize the impact of question about accuracy over time. In fairness, the temporal nature of the chart's data derives only from the fact that each subject answers the twenty questions in a numerical order. The data used to create this graph does not consider the actual time the subject spent determining each answer. During the experiment, we did record the time of each correct answer and, if we ignore the first question, we can easily calculate the differential time per question. Since we did not record the experiment start time, we cannot factor in the first question for analysis of time taken. Furthermore, we must keep separate the analysis of the times for question with feedback versus without feedback, since the two question types have such different characteristics.

Given any method of determining the temporal nature of the questions with respect to scores, we quickly realized two significant flaws in analyzing this aspect of the data. First, and paramount, we simply do not have enough data to analyze properly. With only twenty questions per subject, each of which can be one of the four algorithm scenarios, we have far too many variables to consider in a limited dataset. Clearly, we need more than ten questions for each type to see trends or we must limit the algorithm choices. To be sure, the simulator only allows about 6% of the questions to use the Simple algorithm. However, this limitation still does not give us anywhere close to a large enough population data for statistical analysis. Additionally, the experiment randomized many other variables, including depth and breadth, number of sources, and source locations and orientations. Correlating this many variables in a temporal analysis requires a much larger dataset. Secondly, we would need to use a formal (and expensive) statistical analysis package, assuming we had enough data. Presently, this lies outside the scope of this document, but we propose future refinement and analysis of this concept.

Though we hesitate to directly compare P values across separate datasets, generally we note a differential trend between the two types of questions. Questions without feedback gave us typically much lower P value and therefore superior confidence levels in our assertions than the data from questions with feedback. We ostensibly could attribute this to the fact that the questions without feedback followed the questions with feedback. This could illustrate a trend of improvement over time, between the two groups. However, we could just as easily rationalize this based off the definitive and unambiguous nature of the later questions. We have intentionally kept analysis of the two question types separate throughout this document simply because they differ so completely in disposition. Alternatively, this differential trend could simply stem from lack of a comparable statistical nature of the two data sets. Therefore, we still cannot determine the validity of hypothesis two, with any degree of certainty.

Hypothesis Three Evaluation

According to Table 13 on page 95, the third hypothesis states: “One of the algorithms developed in this research will produce superior results than the others. We anticipate the more complicated Bouncing Reflections algorithm to outperform all others.”

Across the board, both Direct Paths and Bouncing Reflections algorithms beat out the DirectX algorithm and, not surprisingly the Simple algorithm. Yet, now we must consider how they compared to each other. We can see from the repeated measures tests and to a lesser extent the non-parametric independent tests that the two algorithms do not significantly differ in mean scores. Most likely, this proximity is only a coincidence since, do to convoluted nature of the analysis and the many assumptions take. However, we can say with at least a loose degree of certainty (90%) that the Direct Paths algorithm outperformed the Bouncing Reflections algorithm in the independent tests model. Obviously, the first part of hypothesis three holds true, regardless of which of the two scored better. For the latter half, though, we find, at best, exactly the opposite results than we expected.

So we must ask the obvious question: why would the Direct Paths algorithm score higher than the Bouncing Reflections algorithm? Another way to view this same issue: why did we assume the Bouncing Reflections algorithm would perform better in the first place? Finally, we should consider, since our hypothesis turned out exactly opposite of what we expected, was our analysis or experiment flawed in some way? We will address these three questions in reverse order.

First, though we cannot absolutely prove the validity of the statistical analysis covered in the previous section, we can say with relative certainty that the techniques used employed processes are common to any statistical research. Furthermore, the non-statistical analysis provided strongly supports the same conclusions. Even cursory parsing of the data supported this. Finally, anecdotal evidence supports the conclusion that the Direct Paths algorithm would surpass the Bouncing Reflections algorithm. After parsing the data, we resurrected the simulator and reevaluated our initial assessment of the algorithms. Upon careful aural consideration, aesthetically and subjectively we found that we could agree with the ranking conclusion. The Direct Paths algorithm simply sounds better than the Bouncing Reflections algorithm. We will consider why later in this section.

Now we must consider why we predicted that the Bouncing Reflections algorithm would perform better in the first place. As indicated in the evaluations of the algorithms earlier, the Bouncing Reflections algorithm actually expands directly on the Direct Paths algorithm. One of the first things the Bouncing Reflections algorithm does is to run the simpler algorithm and store these results for the zero-order reflection paths. It then enhances this data by bouncing some to many reflections off the environment walls to generate a finite impulse response for the room. This clearly should give us not only the direct line-of-sight information, but also more extensive audio cues. In the appendix, we elucidate that audio cues such as interaural delay time, head shadow, and head motion contribute primarily to the lower order reflections. Other audio cues like pinna and shoulder response more heavily depend on higher frequencies and upper order reflections or even reverberation. Clearly, though, providing more audio information to the listener should logically enhance perception. Therefore, we must finally question why the Direct Paths algorithm scored higher than the Bouncing Reflections algorithm.

We have two theories about this conundrum. First, we must remember that early echo response and reverberation provide audio hints as to the nature of the sound source, its location, and properties of the room. We designed this experiment to test strictly localization and not spatialization. The difference is key. If we designed an experiment to test, say, the reflective properties of the walls and the room, arguably, the Bouncing Reflections algorithm should prove far superior. Considering this, we do not find it surprising that the Direct Paths algorithm scored higher than the Bouncing Reflections algorithm. This is not to say that the Direct Paths algorithm is better than the Bouncing Reflections algorithm. Rather, it simply performed better on a localization scale. We can further say this about all of the algorithm comparison conclusions. Considering DirectX DirectSound® promotes no spatialization and only localization like our Direct Paths algorithm, our conclusions for general superiority still hold.

Finally, we can also theorize that the Bouncing Reflections algorithm could have performed better given much higher complexity. On the flip side, and probably could go without mention, the Bouncing Reflections algorithm reduces almost to the Direct Paths algorithm at the lowest complexity level and therefore should perform better for this experiment at low levels. In fact, the data in Table 27 superficially supports this as we see a spike for the score for the Bouncing Reflections algorithm at a complexity of two. However, we cannot conclusively analyze this since, statistically speaking, we do not have enough data points due to probability distribution of combining the breadth and complexity variables versus the number of algorithms and question types. The same holds for the spikes at the upper end of the average values for the Bouncing Reflections algorithm in Table 27. Furthermore, considering the clumping of resolution listed in Table 16 and the example finite impulse responses in Figure 23 through Figure 26, we can make the argument that the experiment did not allow a high enough level of fidelity for the Bouncing Reflections algorithm. Clearly, given faster processor times or dramatically better-optimized algorithms, we could have allowed this algorithm enhanced performance. However, the limits of this experiment constrained this algorithm. We believe that the limited fidelity of this algorithm actually distracted from the localization. The subject had just enough spatialization information to distract him from the localization component.

Complexity	**Without Feedback**								**With Feedback**							
	BOUNCING REFLECTIONS		DIRECT PATHS		DIRECTX		SIMPLE		BOUNCING REFLECTIONS		DIRECT PATHS		DIRECTX		SIMPLE	
	avg score	num	avg score	num	avg score	num	avg score	num	avg score	num	avg score	num	avg score	num	avg score	num
2	0.50	2	0.04	5	0.13	4	0.00	1	0.52	4	0.41	6	0.48	3		
3	0.28	9	0.40	5	0.05	7	0.00	1	0.32	6	0.45	7	0.26	10	0.24	4
4	0.12	5	0.38	5	0.03	11			0.35	16	0.39	6	0.28	9	0.80	4
5	0.25	13	0.29	10	0.16	13	0.03	6	0.38	17	0.24	17	0.42	16	0.11	2
6	0.06	19	0.23	18	0.20	19	0.12	7	0.21	15	0.29	13	0.35	17	0.09	3
7	0.31	21	0.20	30	0.22	24	0.06	3	0.29	14	0.30	21	0.35	16	0.20	3
8	0.24	25	0.22	19	0.14	28	0.04	7	0.39	20	0.39	30	0.23	26	0.34	4
9	0.12	38	0.16	23	0.06	31	0.18	6	0.30	24	0.31	27	0.24	19	0.27	5
10	0.19	39	0.27	29	0.16	33	0.09	6	0.26	29	0.33	35	0.32	39	0.21	5
11	0.13	37	0.27	38	0.12	32	0.07	9	0.27	40	0.38	37	0.26	40	0.36	7
12	0.20	21	0.17	32	0.10	38	0.02	4	0.31	36	0.30	26	0.25	39	0.47	4
13	0.19	13	0.18	31	0.14	27	0.05	11	0.24	20	0.33	27	0.16	28	0.18	2
14	0.14	25	0.22	20	0.15	19	0.05	6	0.21	21	0.33	17	0.22	27	0.21	2
15	0.16	13	0.20	18	0.16	18	0.09	4	0.28	15	0.32	17	0.14	14	0.19	5
16	0.12	25	0.20	19	0.12	13	0.51	1	0.22	14	0.38	19	0.27	21	0.20	2
17	0.14	14	0.30	11	0.20	16			0.20	18	0.25	15	0.17	15	0.18	2
18	0.33	10	0.23	11	0.19	16	0.00	1	0.38	18	0.19	8	0.09	8	-0.52	2
19	0.26	8	0.30	4	0.20	4	0.00	1	0.41	7	0.61	3	0.31	18		
20			0.23	5	0.00	3			0.24	7	0.34	2	0.43	4	0.41	1

Table 27: Average Scores for Algorithms by Complexity Level

Future Work

Acoustic Assumptions

For this experiment, we have intentionally limited the simulator to constrain certain global acoustic properties. This allowed us to evaluate other variables without distraction. However, future experiments should include these as enhancements and promote the testing of these assumptions. The easiest of these assumptions to next implement are those made while calculating reverberation paths. Specifically, we dramatically simplified the formulas for attenuation and absorption/reflection for acoustic paths. We would also like future iterations to include frequency distribution for attenuation. Another easy improvement would refine the method of material absorption and dispersion (the `calculatedCoefficient`).

Algorithm Optimizations

Within the Bouncing Reflections algorithm, we blindly test each wall for an intersection with each reflected path. Clearly, we could easily optimize this algorithm by simply attempting to predict the first face to test in the next segment when we find a reflection. This is only one of many optimizations we could consider to speed up this algorithm to run in extreme cases.

Direct Paths Algorithm Expansion

In previous developments of the Direct Paths algorithm, we designed an algorithm called Reflected Paths that could quickly calculate the first-order and even second-order reflection paths for a basic rectangular room. This used optimized trigonometric formulas that ran practically in real-time. Figure 9 on page 36 illustrates this approach. However, we excluded this functionality because it only worked in limited, geometrically simple spaces.

This algorithm, though, if we could generalize it, has potential to provide exceptional localization, if not spatialization.

Alternative Algorithm Development

In the section "Current Work" on page 29, we described a number of alternate algorithms for future development. Expansion on this level could provide further insight about how we consume 3D sound and methods to exploit acoustic properties in virtual environments. We would especially like to develop the Matrix algorithm (see "Matrices of Impulses Algorithm" on page 42), as this has potential to make slower, powerful algorithms run in real-time.

Orientation and Location Thresholds

Currently, every movement of the avatar or sound source automatically causes the simulator to recalculate the current algorithm. However, we could possibly employ thresholds to minimize this burden on the processor. It is unclear to what degree a listener will tolerate the resulting lack of fidelity, but we can easily design an experiment to test and evaluate threshold updates.

Complicated Rooms and Structures

Though the simulator allows for complicated, even non-rectangular rooms and structures, we did not account for this in the current experiment. The experiment limited questions to a simple, unified space clear of objects and obstructions. We advise further research and consideration of complex rooms, multi-room structures, spaces and with objects, and/or areas of varying sizes and shapes. We also would like to pursue implementation of the algorithms in large-scaled spaces or outside environments. The

Bouncing Reflections algorithm, with significant optimizations, has the potential to manage some or all of these enhancements.

Experiment Redesign

During the analysis of this experiment, we realized design shortcomings that did not allow for certain considerations. Due to the large number of variables, small set of questions per student, and limited number of student subjects we could not properly analyze hypothesis two. Future experiments could easily focus on providing the ability to analyze improvement over time due to learning. We also intend to submit the data from the current experiment into a third-party statistical package and process the results.

Conclusion

We began this research well aware that the current industry standard for 3D sound, Microsoft's DirectX DirectSound®, contained flaws and significant shortcomings. Over the past few decades, many users have patiently waited for the market to correct itself and provide serviceable virtual acoustics, to no avail. With that in mind, we strove to develop comparable and ideally superior 3D acoustic algorithms that might eventually replace or enhance what is currently available. To accomplish this task, we built a robust virtual environment simulator that not only presents dynamic and sophisticated 3D visuals, but also has the potential to reproduce acoustics from any number of 3D sound algorithms. We enhanced the Three Dimensional Sound (or TDS) Simulator with the flexibility to run experiments that compare and evaluate these acoustic algorithms and eventually incorporated two of our own 3D sound algorithms for comparison with commercial approaches.

Evaluation of the experiment data strongly supports the assertion that the two algorithms designed for this research dramatically outperform Microsoft's DirectX DirectSound®. In fact, we discovered clear ranking with respect to acoustical localization for the algorithms in this experiment. The Direct Paths algorithm outstripped all others, followed by the Bouncing Paths algorithm. The undisputable loser, the DirectX algorithm performed only marginally better than sound reproduction with no 3D algorithm at all!

Since the algorithms developed in this research purport independence of platform, programming language, and even application, the potential practical applications of this research are limitless. These algorithms could easily enhance common virtual environments from video games to dynamic movie story telling. Even the government or military could deploy these algorithms to enhance virtual simulators, significantly reducing expense and risks from training. With further development and refinement, this research has the profound potential to influence any 3D virtual environment application.

Appendix

Concept Overview

In order to better comprehend the concepts in this research, in this section we provide material for a basic understanding of acoustics and related topics. The following includes a simple primer on acoustics and the physical aspects of sound and perception. We do not assume knowledge of physics and psychology. Other sections follow that further support the research presented in the main part of this document. The reader may skip the following sections.

3D Sound Propagation

Reflection

In typical atmospheric conditions, sound uniformly spherically propagates out from a source at a rate of approximately 344 m/s. As it gets further from the source, it loses power, or volume proportional to the inverse square of the distance traversed. Furthermore, sound does not simply travel in a straight line from the source to the listener. Instead, it emanates out in all directions, bouncing off some surfaces, bending around others, and even partially absorbs into boundaries such as carpets or rugs. Within any given space, a listener receives audio cues about the surroundings through these reflections, refractions, absorptions, and attenuations. Depending on the environment and circumstances, certain aural interactions become more important than others do. When outside or in a large space, attenuation, or volume decay prevails most, since few surfaces interact with the sound. For the purposes of enclosed environments such as rooms and buildings, which primarily concerns this research, initial reflections give the listener the most information about the sound source. These cues can include the relative location, condition, and physical properties about the space occupied by the sound source. We can group reflection paths into categories of order (Figure 2, Figure 3, and Figure 4 on page 17) based on the complexity or number of bounces off the walls. Sound that travels directly from the source to the listener we define as the zero-order reflections group. Paths that bounce off only one wall, we call first-order reflections, while those making two bounces are second-order, and so on. See “Reflected Paths Algorithm” on page 34 for a brief analysis of some equations involved in generating these reflections.

Refraction

Sound does not always travel in a straight line from the source. Typically, it bends around or through objects and surfaces. For instance, one can hear an approaching siren around a corner from far away or somebody speaking in a completely different room. Sometimes refraction occurs due to the sound waves actually bending around objects. Other times, the sound travels through the medium and will distort perceivably. For example, when a person puts his head underwater and hears somebody speaking the voice seems unclear. These effects depend highly on frequency. Unfortunately, our algorithms do not currently take into account refraction for calculations. We hope to employ this in future models.

Absorption & Dispersion

When sound wave hits a surface, not all of its energy reflects off the boundary. Depending on the medium and surface properties of the material, some energy might gets absorbed into the object, while some will disperse in multiple directions. For more details about this aspect of sound propagation and how our algorithms can manage it, see “Wall Dispersion Algorithm” on page 41.

Frequency	**Relative Humidity %**								
(Hz)	10%	20%	30%	40%	50%	60%	70%	80%	90%
1,000	14.0	6.5	5.0	4.7	4.7	4.8	5.0	5.1	5.3
1,250	21.0	9.4	6.7	5.9	5.7	5.7	5.9	6.1	6.3
1,600	32.0	14.0	9.8	8.1	7.5	7.2	7.2	7.4	7.5
2,000	45.0	22.0	14.0	11.0	9.9	9.3	9.0	9.0	9.1
2,500	63.0	32.0	21.0	16.0	14.0	12.0	12.0	11.0	11.0
3,150	85.0	49.0	32.0	24.0	20.0	17.0	16.0	15.0	15.0
4,000	110.0	75.0	49.0	36.0	30.0	26.0	23.0	21.0	20.0
5,000	130.0	110.0	74.0	55.0	44.0	38.0	33.0	31.0	28.0
6,300	160.0	160.0	110.0	84.0	68.0	57.0	50.0	45.0	42.0
8,000	180.0	220.0	170.0	130.0	110.0	89.0	78.0	69.0	63.0
10,000	190.0	280.0	240.0	190.0	160.0	130.0	120.0	100.0	95.0
12,500	210.0	360.0	340.0	280.0	240.0	200.0	180.0	160.0	140.0
16,000	230.0	430.0	470.0	420.0	360.0	320.0	280.0	250.0	230.0
20,000	260.0	510.0	600.0	580.0	520.0	470.0	420.0	380.0	350.0
25,000	300.0	580.0	740.0	770.0	730.0	680.0	620.0	570.0	520.0
31,500	360.0	670.0	890.0	990.0	1000.0	960.0	900.0	840.0	790.0
40,000	460.0	780.0	1100.0	1200.0	1300.0	1300.0	1300.0	1200.0	1200.0
50,000	600.0	940.0	1300.0	1500.0	1700.0	1700.0	1700.0	1700.0	1700.0
63,000	840.0	1200.0	1500.0	1800.0	2100.0	2200.0	2300.0	2300.0	2300.0
80,000	1200.0	1600.0	2000.0	2300.0	2600.0	2800.0	3000.0	3100.0	3100.0
100,000	1800.0	2200.0	2500.0	2900.0	3300.0	3600.0	3800.0	4000.0	4100.0

Table 28: Sound Attenuation (dB/km) (Bacon & Jarvis, 2007) (ISO 9613-1, 1993)

Attenuation

As sound travels, it loses power, or intensity, and thus decreases in volume. In air, under typical circumstances, perceived volume decreases approximately according to the inverse square of the distance traveled. See Equation 4 and Equation 5 for the basic formulas to calculate the decreased decibel value of mono-frequency spherical waves emanating out from a source with a power of *P* watts (Resnick, Halliday, & Krane, 1992) (Kinsler, Frey, Coppens, & Sanders, 2000). Generally, this loss depends on environmental conditions such as ambient temperature, relative humidity, atmospheric pressure, and even the frequency of the signal (ISO 9613-1, 1993). Thus, when a reflection path reaches an observer from a sound source, we must take into account the distance traveled to calculate the resulting volume of that reflection. Table 28 summarizes the attenuation of sound source due to humidity.

Functionally, employing attenuation breaks down to simply applying a filter spectrum envelope to the original signal. This envelope will accentuate certain frequencies, while moderating others, according to the table and the distance the reflection has traveled. Initially, however, the simulation will treat all frequencies the same, attenuating according to the formulas below. We consider the volume of original sound file as the reference level after traveling a direct path from the sound source to the listener. See "Attenuation Algorithm" on page 67 for how the TDS Simulator currently factors and manages attenuation.

$$I = \frac{P}{4\pi r^2}$$

Equation 4: The Intensity of a Spherical Wave at Radius [r]

$$SL = 10\log\frac{I}{I_0} = 10\log\frac{P}{4I_0\pi r^2}$$

Equation 5: The Decibel (dB) Level of a Spherical Wave

3D Sound Perception

After we have an understanding of the physical properties and the equations involved in calculating sound paths, it is also necessary to consider how human physiology and psychology come into play. In order to appreciate this, we will further clarify the distinction between sound spatialization and sound localization. We define sound localization as the listener's ability to determine, via acoustic information, the physical position of a sound source. For instance, one can find a sound source in a space without actually seeing it. On the other hand, sound spatialization encompasses the overall 3D sound effect, including sound localization, perception of the environment, condition of the sound source, and any other information given by auditory assessment. Specifically, we consider least eight cues (Blauert,

1997) (Burgress, 1992) of sound localization and spatialization, each of which we describe below.

Interaural Delay Time

Interaural Delay Time (IDT) is the delay between when a sound signal arrives at each ear. This can range from no delay if the source sits directly in front, behind, above, or below the source to up to approximately 0.63 ms (Blauert, 1997) (Burgress, 1992) if the sound comes from either side of the listener. In other words, IDT is simply the difference in time it takes sound to travel to the ears. Variable λ in Figure 31 illustrates this measurement. Primarily, this cue gives the listener a sense of horizontal direction to the sound source, and it is only important to the very low order, initial reflection paths. However, this audio cue does have a significant impact on complexity of the sound path calculations, as it requires twice as many reflection computations (one for each ear).

As a possible method for avoiding this doubling of computations, we could determine the monaural sound path to the center of the head and estimate the offset for each ear, factoring in the relative orientation and frequency of the signal (Figure 31). We can even

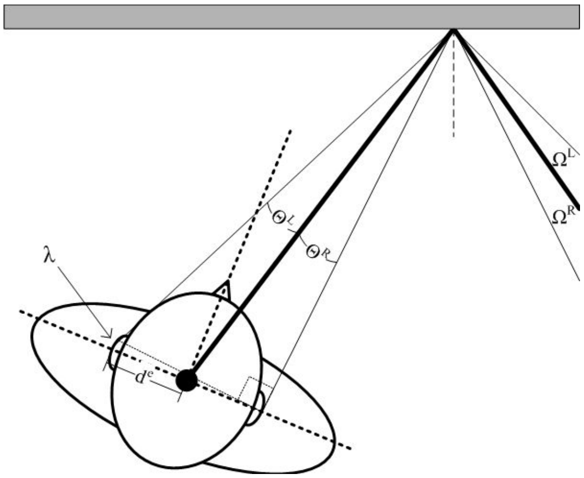


Figure 31: Interaural Delay Time due to Avatar Orientation

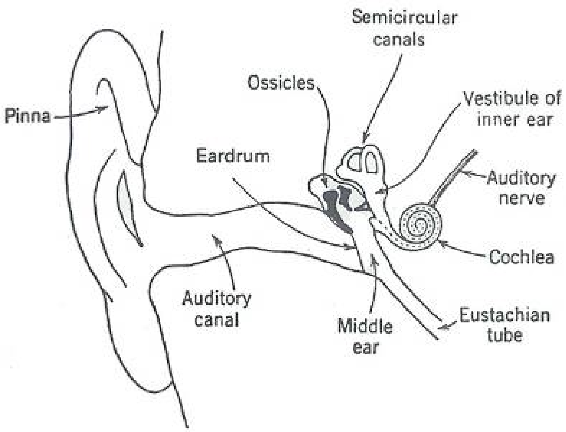


Figure 32: The Anatomy of the Ear

further simplify this calculation by determining the stereo effect on just the last segment of the sound path. This approximation should intuitively hold for cases where the avatar does not sit in extreme proximity to the wall. As the distance from the wall increases relative to d^e, the angles Ω^R and Ω^L approach zero, while the distance difference λ between the final stereo segments remains acoustically significant.

Head Shadow

Next, we define *Head Shadow* as the difference in volume between one ear and the other caused by sound traveling the added distance. Similar to the interaural delay time, we find this audio cue at its apex when the sound source sits to the side of the listener and nothing when in front, behind, above, or below. At most, head shadow accounts for about 9 dB of sound loss (Blauert, 1997) (Burgress, 1992), and it only truly affects zero-order reflection paths. More complex reflection paths will effectively cancel each other out with respect to this cue because they reach either ear at approximately the same time and volume. Head shadow influences both direction and distance localization cues. If calculating two paths for interaural delay time and compensating for attenuation in general, we incorporate this factor by default.

Pinna Response

Pinna Response takes into account the ability of the outer ear to filter sound. The actual shape of the human ear and the ability of the brain to compare the filtered signals sent by the two ears both cause this audio cue. Due to the physiology of the pinna, this cue helps the listener determine both direction and elevation of the sound source. For the purpose of sound path calculations and our algorithms specifically, we deal with this cue within the speaker abstraction layer that the system utilizes (see “Computational Issues” on page 153).

Shoulder Response

Sound reflections off the upper body of the listener provide a spatialization cue called *Shoulder Response*. These typically limit to frequencies in the signal of around 1-3 kHz (Blauert, 1997) (Burgress, 1992) and can enhance elevation and directional localization. Our algorithm does not currently account for this cue, as it does not have a significant impact on spatialization and localization in small, enclosed environments. However, further expansions might include calculations for shoulder response.

Head Motion

A more abstract audio cue, *Head Motion*, occurs when the listener moves his head to re-evaluate the other filters. Since the head of the virtual avatar represents the head of the listener (or the area surrounding the virtual camera), the physical head motion of the listener, or end user, does not affect this cue. Rather, the avatar must turn its head in the virtual world. An ideal virtual environment would rotate the avatar's orientation when the user physically turns his head in the real world. Turning the avatar's orientation simply forces the reevaluation of the algorithm's calculations, so by nature of the simulator we have included implementation of this audio cue.

Vision

Vision can significantly influence any received audio cues, causing the listener to ignore what he hears if it differs from what he sees. Human tendency emphasizes visual input over audio information. However, vision could also reinforce the effectiveness of sound spatialization (or vice versa) if the two correspond. For this reason, we have developed a sophisticated 3D virtual environment through which users can navigate (see "The TDS Simulator" on page 49).

Early Echo Response & Reverberation

Finally, the two most significant audio cues to this project, *Early Echo Response* and *Reverberation*, give the user dramatic hints as to the nature of the sound source, its location, and properties of the room. These cues do not help localization in large open environments, but have substantial influence in enclosed spaces. Effectively, they are both part of the same physical concept; however, they have distinct effects on the listener's perception. Early echo response (EER) includes the reflections that reach the listener up to 50 or 100 ms after the sound begins. Depending on the room dimensions and properties, these usually consist of the zero, first, and sometimes second-order reflections (Blauert, 1997) (Burgress, 1992). Reverberation includes the group of dense reflections that follow the EER. Figure 33 shows a standard impulse response for a room. In the real world, we can measure this by playing a high-powered impulse from a speaker and recording the results. There exists a point in time when the echoes begin to function as reverberation. We define this variable as approximately the time that the sound pressure takes to decay 60 dB after the source has ceased transmitting (Resnick, Halliday, & Krane, 1992) (Kinsler, Frey, Coppens, & Sanders, 2000). Arguably, reverberations do not directly affect sound localization; however, they do significantly influence sound spatialization.

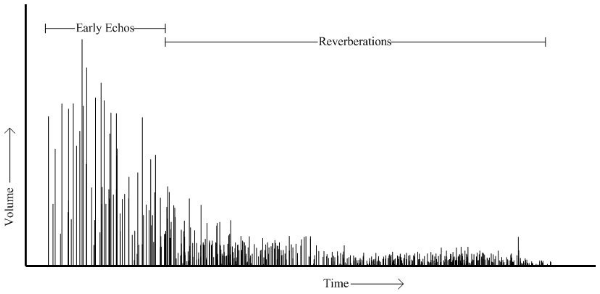


Figure 33: Echo vs. Reverberation

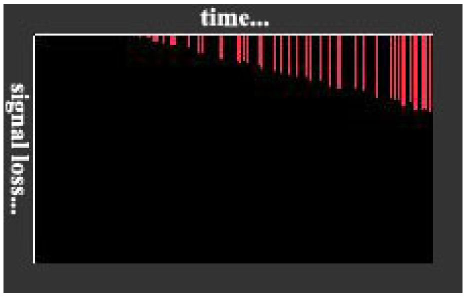


Figure 34: Generated Impulse Response

Reverberation and especially EER computations must take into account other audio cues, such as attenuation, absorption, and IDT. The depth, or order, to calculate is somewhat subjective within the reverberation range, as the ear at some point fails to notice the separation between the impulses.

We note that we can diagram and store an impulse response in a number of ways. Figure 34 shows a typical rendering of the strength of the echo in decibels versus time. Since the sound API (DirectSound DirectX®) which we use describes (Microsoft, 2007) volume for playback as an integer from 0 to -10,000, or hundredths of a decibel loss from the original sound volume (it cannot play a sound louder than the original source), we can store the impulse response in the format of Figure 34. This diagram presents the conceptual inverse of the data in Figure 33. It is also possible, and sometimes useful, to store the impulse as an array of the path lengths from the original source (see “Reverberations Data Structure” on page 64 and “Impulse Response Data Structure” on page 66).

Computational Issues

Sound API’s

Most of the computers that end-users find available today come with sound cards and methods of accessing these cards. The industry calls these methods, which generally reside deep in the operating system, application programming interfaces (or APIs). A computer program such as a 3D environment, 3D game, or CAD program must go through these types of interfaces, or abstraction layers, to access physical devices. An API can allow an application to process and then represent graphics, interpret network data, or even play and possibly manipulate sounds. We refer to an API as an abstraction layer because it provides a common standardized software interface to any number of different hardware configurations, so that the programmer does not need to focus on low-level, system dependent aspects of programming.

Discussed in the section “DirectSound® Library” (see page 83), some common examples of the API’s available for sound include Microsoft’s DirectX DirectSound® (now called XAudio2 and X3DAudio), OpenAl, fmod, EAX, and a few other more antiquated interfaces. Typically, the choice of which API to employ resides with designer of the application, though frequently he will allow the end user the choice between a select few. Many of these API’s tout the ability to generate 3D sound for virtual environments. However, they all fall dramatically short on the algorithmic side and use the shortcuts listed in the sections below such as HRTF or incorporating limited or no geometrical data.

In addition to analyzing the relationship between the sound API and the sound card, we must also look at how the speaker configuration factors into the product. Sometimes called the speaker abstraction level, or also the auditory display, the speaker types and locations can dramatically affect the overall experience of the end user. On most systems, the user tells sound API, via the operating system, which auditory display configuration connects to the system. The API then applies this information to modify slightly the algorithm or filters it will use to generate the 3D sound. In order to simplify the analysis of our algorithms, we assume the use of only simple headphones. This allows for a uniform and standard speaker abstraction model, regardless of user, user-physiology, speaker design and location, or room acoustics.

3D Sound Cards

Presently, on most computer systems, the sound cards frequently advertise as having the capability of generating 3D sound. Manufactures such as Creative Labs, Crystal Sound, EAS, TurtleBeach, have offered many 3D sound cards over the past years. The basic, low-end PC's available from Gateway, DELL, Compaq, most game consoles, and even some Macintosh computers typically include one of these cards, often integrated into the system.

In fairness, some of the more formidable sound cards do perform basic hardware accelerated 3D sound calculations. They provide a faster method of performing the algorithms requested by the API via specialized processing hardware. However, like the API designers, the manufacturers of each of these devices make major, unjustified assumptions about how best to handle 3D sound. The industry standard tends to exploit the common surround sound algorithms such as those found in most home audio systems in combination with abstract and unfounded algorithms to make sound seem like it comes from a 3D location. This essentially fools the user into thinking he is experiencing a realistically calculated 3D sound. Without the visual 3D environment to reinforce the sound virtualization, the average listener might not even be able to guess the actual sound location.

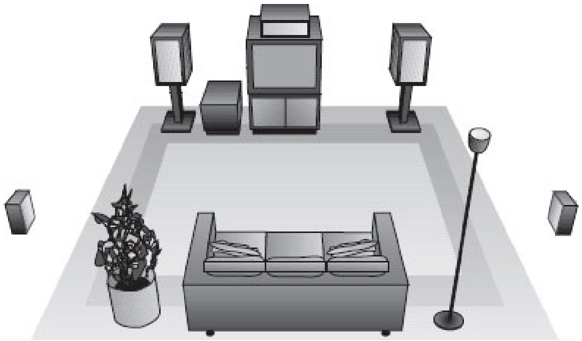


Figure 35: Dolby 5.1 Speaker Placement (Hull, 1999)

Figure 36: The Neumann KU 100 Dummy Head (Neumann, 2007)

Software vs. Hardware

Ideally, in order to produce a proper 3D effect, an application would send basic 3D information to the sound card through the installed API and the API has the responsibility to use the processing hardware on the sound card to generate effective 3D sound. The API would take the environmental information, run it through a set of algorithms, and then use the hardware available to it to mix and filter the sounds to get the desired effect. The 3D component of many of the currently available sound cards amounts to little more than hardware accelerated mixing and the application of built-in (HTRF) filters made available to the API. On systems without this hardware or when specified by the application, the API should have the option of performing its calculations at the software level which would obviously run much slower.

Surround Sound

Surround sound defines as the specific placement of speakers around a listener to achieve the illusion of sound coming from any or all directions (Hull, 1999). It also typically describes the method of dispersing the sound signal to these speakers and any filter effects placed on the original source to achieve the desired 3D spatialization results. Some examples of surround sound include a "wide stereo" effect using two speakers or headphones, DTS (Digital Theater Systems), THX, or most commonly Dolby 5.1, as well as 6.1 and 7.1, (Figure 35) which the average home stereo system typically includes. Some even more sophisticated auditory display systems [http://paw.princeton.edu/issues/2012/02/08/pages/7041/index.xml] have potential widespread commercial applications. In this paper, we do not propose to replace or modify these systems. They are ubiquitous and even standardized. Rather, we wish to send better or more accurate information to the surround sound speakers via the API layer of the computer system. The computer's sound API will still need to know the speaker placement configuration. However, our algorithms should always generate the same output for a given virtual situation regardless of the speaker setup, and then let the sound API express the results through the speaker system as it sees fit.

Head Related Transfer Function (HRTF)

An intuitive approach to designing sound synthesis systems derives from the consideration of how one would perceive the sound given a particular environment. The Head Related Transfer Function (HRTF) converts the signal a listener might receive through his aural system to all assorted external acoustic factors. In practice, it is little more than an organically designed set of filters performed on a sound to make it seem to come from a specific direction (Burgress, 1992). The filters typically have careful design and engineers base them on good acoustic and physiological assumptions. Yet, they have many limitations.

The concept of HRTF follows rather simply. An engineer uses an accurate model (Figure 36) of a typical ear, head, and sometimes even upper body to "listen" to a variety of sounds from a range of locations and orientations around the avatar. The model can even be a living person. Deep in the ear, near the eardrums, the designer places microphones to capture the signals, representing how the human ear receives sounds. These signals go through intense analysis to engineer, in reverse, a series of filters that a system can applied to sounds in order to replicate perceived spatialization.

Though the results seem efficient and even somewhat convincing, HRTF implementations have several well-known inadequacies. First, the generated filters only match accurately to the model used in development. Since every human differs anatomically, we cannot inherently generalize this approach. Furthermore, the generated filters are expensive to create and typically tightly tuned to a specific playback environment, such as an auditorium or a padded room. Though some assumptions can generalize the surroundings, HRTF fundamentally only produces effective results in the specific controlled circumstances.

Experiment Documents

Below we have included documents relevant to the experiment explained in this research. Table 29 shows an example transcript of the instructions provided by the researcher to start the experiment for each subject. Table 30 exhibits the IRB approved consent form that each subject must sign.

Data File Examples

We have included below some examples of the data files used by the simulator. See the section "Scene3D Data Structure" on page 52 for a detailed explanation of design and implementation.

Please have a seat. If you wouldn't mind, please take a moment and read through this consent form and sign it at the bottom. Also, put your name and student number on this sheet to get extra credit.

[Subject reads the consent form and signs and fills out extra credit sheet. Researcher resets the simulator and cleans the keyboard, mouse, and headphones.]

You are going to be moving the avatar, or you, through this virtual environment. I'll explain the experiment in a moment, but first let me show you how to move around and use the controls. [While the researcher demonstrates.] You can use the cursor keys to look left or right and move forward or backward. You can also use the mouse. Left-click and drag looks left, right, up, or down, but it does not actually move the avatar. Right-click and drag also lets you look left and right, but it will allow you to move forward and backward. As you can see, the mouse can be a little sensitive, so it is best to use slow, short movements. It is more efficient then the keyboard, but can be erratic if you are not careful. The keyboard is slower to use, but you have more control. You are welcome to use either, both, switch back and forth, or use them in combination. You can also use the mouse pad on the keyboard. Use whatever is comfortable for you.

In order to familiarize yourself with the controls, I am going to have your go through a short training exercise. [Researcher initiates training mode of the simulator.] Here you are asked to go up to each object, "A", "B", "C", and "D" in turn and get right in front it. Then just step through it as if it were a door. You can use the keyboard or mouse or switch between them. [Researcher observes and verbally assists while subject moves though the training. Researcher will almost certainly have to instruct the subject to move through the first object and then point out that "D" is to the right (off screen initially). Subject completes the training.]

Now the actual experiment will consist of twenty questions, each of which will have two to ten of those sound objects randomly scattered throughout the room. Your job is to pick which one is playing the sound based on what you are hearing and seeing. The first ten questions will be with feedback. This means that if you see, say objects "A", "B", "C", "D", and "E" and you think it is "C", you type "C" to select it. You do not have to move through the object at this point. Just type it to select. But moving around might help you decide. If you are correct, you will be told "Correct Answer" and it will automatically move you to the next question. If you are wrong, it will say "Incorrect Answer, Please Try Again" and will keep asking until you get it right. Obviously, you can get it right by process of elimination, if nothing else.

Questions eleven through twenty are without feedback. If you think it is "D", you type the letter "D" and it will move you on to the next question without letting you know one way or the other if you are correct until the experiment is done.

When each question starts, it will automatically play the sound for you. You will almost certainly want to replay the sound as you are moving around to hear it from different perspectives. Just hit the "Space Bar" to replay anytime and as often as you would like. Occasionally, if you replay the sound to soon after it just finished playing, the simulator is not ready to play. Just wait half a moment and replay the sound again. And remember to select the sound you think is playing, just hit the key. If you think "C" is playing, just hit "C". If you think "D" is playing, just hit "D". You do not have to move through the sound like before. However, it might help to move around to hear it from different locations.

One last thing. Sometimes, no matter how much you move around you will not be able to tell. That is okay. It is okay to guess. Do not get frustrated, as this is part of the experiment. Any questions?

[Researcher hands the subject the headphones and makes sure they are on properly. Researcher starts the experiment and moves away.]

Table 29: Experiment Instructions Transcript

Consent form

You are being invited to participate in a research project by Dr. Chee-Hung Henry Chu and Student Researcher Scott McDermott from the University of Louisiana at Lafayette. This study will be conducted to determine the accuracy and effectiveness of virtual 3D acoustic algorithms and sound reproduction.

You will be asked to sit in front of a standard computer, wearing a pair of off-the-shelf circumaural headphones (encompassing the entire ear), and initially explore the simulator to get a feel for manipulation and movement within the virtual environment. Actual testing will pursue with your selecting from a possible list of candidate objects, one of which is actually playing the sound. It will take you about 30 minutes.

You are under no obligation to participate in this research, it is your choice whether to be a part of the study or not. You may decide not to be a part of the study at any point before, during, or after the study. There will be no bias or penalty from this agency, the State of Louisiana or the University of Louisiana at Lafayette or Southeastern Louisiana University if you decide not to participate or if you decide to stop participating in the research.

There is no particular benefit to you if you participate, but the project may allow research to be done to further explore the benefits and implementation of acoustics in simulations and real-world settings. The major risk to you is of taking up your time.

The results of this research will be published in a professional journal after it has been completed but no personal information about any of the people who were included will be part of any of the reports. The form you are filling out today will be destroyed after all the data has been entered into analysis. There will be a unique number given to each test subject that the simulator will assign. These numbers will be used to keep the data sets separate from each other. No association with your personal information will be stored in this test. If you have any questions about this research or your participation in the study you are welcome to call Dr. Chu at 337-482-6309 and/or Mr. McDermott at 337-482-6338 at UL at Lafayette. You may also contact the Chair of the ULL IRB, Dr. Mueller, 337-482-6489, for general concerns. You may also contact the Management department head, Dr. Toni Phillips (985) 549-2051. We will make every effort to answer your questions.

CONSENT

I understand that I am participating in research and that the research has been explained to me so that I understand what I am doing. I understand that I may stop participating at any time. I understand that minors are not included in this research.

Signed ____ Date ____________________________

Witness ______________________________ Relationship if any ____________________________

Reason for witnessing the form (ex: unable to read, signs with Ax@)

Table 30: Experiment Consent Form Example

```
Avatar 1.0
// avatar1.txt, Created for TDS

//Measurements are in meters, feet,
Units  inches

Location     150.0 150.0 48.0
Rotation     90 0 1.0
```

```
SoundSource 1.2
// ss speaker.txt, Created for TDS

Units  meters
Location     3.0 4.0 2.0
Rotation     10 10 0

// The loss (in dB) for the source
DirLoss      6.0
File   bounce.wav
```

Table 31: Avatar 1.0 and SoundSource 1.2 Data File Examples

```
Scene 1.2
// scene.txt, Created for TDS Simulator
Units  inches
///////////////////
// Structures... //
///////////////////
NumStructures  7
Structure      house0a
{
   File        simplehouse.txt
   Location    10.0 1210.0 10.0
}
Structure      house0b
{
   File        simplehouse.txt
   Location    600.0 1210.0 10.0
}
Structure      house1
{
   File        house.txt
   Location    0.0 0.0 10.0
}
Structure      house2
{
   File        house.txt
   Location    500.0 0.0 10.0
}
Structure      warehouse1
{
   File        warehouse.txt
   Location    1250.0 0.0 0.0
}
Structure      warehouse2
{
   File        warehouse.txt
   Location    1250.0 480.0 0.0
}
Structure      examplehouse
{
   File        examplehouse.txt
   Location    1250.0 1300.0 0.0
}
////////////////
// Avatars... //
////////////////
NumAvatars     2
Avatar         her
{
   File        avatar2.txt
   Structure   house0a
}
Avatar         me
{
   File        avatar1.txt
   Structure   house1
}

//////////////////////
// Sound Sources... //
//////////////////////
NumSoundSources       2
SoundSource           Speaker
{
   File        ss speaker.txt
   Structure   house0a
}
SoundSource    Dog
{
   File        ss dog.txt
   Structure   house1
}
///////////////////
// Test Sounds... //
///////////////////
NumFeedbackTests      10
NumSelectionTests     10
NumTestSounds         8
SoundSource           A
{
   File        ss dog.txt
}
SoundSource    B
{
   File        ss speaker.txt
}
SoundSource    C
{
   File        ss dog.txt
}
SoundSource    D
{
   File        ss speaker.txt
}
SoundSource    E
{
   File        ss dog.txt
}
SoundSource    F
{
   File        ss speaker.txt
}
SoundSource    G
{
   File        ss dog.txt
}
SoundSource    H
{
   File        ss speaker.txt
}
////////////////
// Grounds... //
////////////////
NumGrounds     16
Ground  Pavement
{
   File               Textures\Ground1.tga
   RepresentedSize    300.0 300.0 1.0
   Thickness          0.5
   Dimensions         0.0 4000 0.0 4000
   Offset             -3.0
}
Ground  AheadRoad 1
{
   File               Textures\Road3.tga
   RepresentedSize    100.0 20.0 1.0
   Thickness          0.5
   Dimensions         1000 1200 0.0 4000
   Offset             -2.0
}
Ground  AheadRoad 2
{
   File               Textures\Road3.tga
   RepresentedSize    100.0 20.0 1.0
   Thickness          0.5
   Dimensions         2000 2200 0.0 4000
   Offset             -2.0
}
Ground  AheadRoad 3
```

```
{
   File              Textures\Road3.tga
   RepresentedSize   100.0 20.0 1.0
   Thickness         0.5
   Dimensions        3000 3200 0.0 4000
   Offset            -2.0
}
Ground  SideRoad 1
{
   File              Textures\Road4.tga
   RepresentedSize   20.0 100.0 1.0
   Thickness         0.5
   Dimensions        0.0 4000 1000 1200
   Offset            -2.0
}
Ground  SideRoad 2
{
   File              Textures\Road4.tga
   RepresentedSize   20.0 100.0 1.0
   Thickness         0.5
   Dimensions        0.0 4000 2000 2200
   Offset            -2.0
}
Ground  SideRoad 3
{
   File              Textures\Road4.tga
   RepresentedSize   20.0 100.0 1.0
   Thickness         0.5
   Dimensions        0.0 4000 3000 3200
   Offset            -2.0
}
Ground  IntersectionRoad 1 1
{
   File              Textures\Road5.tga
   RepresentedSize   100.0 100.0 1.0
   Thickness         0.5
   Dimensions        1000 1200 1000 1200
   Offset            -1.9
}
Ground  IntersectionRoad 1 2
{
   File              Textures\Road5.tga
   RepresentedSize   100.0 100.0 1.0
   Thickness         0.5
   Dimensions        1000 1200 2000 2200
   Offset            -1.9
}
Ground  IntersectionRoad 1 3
{
   File              Textures\Road5.tga
   RepresentedSize   100.0 100.0 1.0
   Thickness         0.5
   Dimensions        1000 1200 3000 3200
   Offset            -1.9
}
Ground  IntersectionRoad 2 1
{
   File              Textures\Road5.tga
   RepresentedSize   100.0 100.0 1.0
   Thickness         0.5
   Dimensions        2000 2200 1000 1200
   Offset            -1.9
}
Ground  IntersectionRoad 2 2
{
   File              Textures\Road5.tga
   RepresentedSize   100.0 100.0 1.0
   Thickness         0.5
   Dimensions        2000 2200 2000 2200
   Offset            -1.9
}
Ground  IntersectionRoad 2 3
{
   File              Textures\Road5.tga
   RepresentedSize   100.0 100.0 1.0
   Thickness         0.5
   Dimensions        2000 2200 3000 3200
   Offset            -1.9
}
Ground  IntersectionRoad 3 1
{
   File              Textures\Road5.tga
   RepresentedSize   100.0 100.0 1.0
   Thickness         0.5
   Dimensions        3000 3200 1000 1200
   Offset            -1.9
}
Ground  IntersectionRoad 3 2
{
   File              Textures\Road5.tga
   RepresentedSize   100.0 100.0 1.0
   Thickness         0.5
   Dimensions        3000 3200 2000 2200
   Offset            -1.9
}
Ground  IntersectionRoad 3 3
{
   File              Textures\Road5.tga
   RepresentedSize   100.0 100.0 1.0
   Thickness         0.5
   Dimensions        3000 3200 3000 3200
   Offset            -1.9
}
////////////
// Sky... //
////////////
Sky clouds
{
   File              Textures\Sky1.tga
}
```

Table 32: Scene 1.2 Data File Example: A Large-Scale Environment Configuration

```
Structure 1.3
// simplehouse.txt, Created for TDS Simulator
//Measurements are in meters, feet, etc...
Units          inches
// The thickness of faces are counted or not
Measurements   outside
// Move the entire structure from (0,0,0)
Offset         0.0 0.0 0.0
//Rotation around axis's (after translation)
Rotation       0 0 0
NumMaterials   7
Material   Floor
{
   Color   BROWN
// Lower value makes a perfect reflection.
   Absorption 0.0005
// Lower value makes a perfect reflection.
   Dispersion 0.0001
}
Material   Wall
{
   Color   RED
   Absorption 0.0010
   Dispersion 0.0005
}
Material   Ceiling
{
   Color   CYAN
   Absorption 0.0005
   Dispersion 0.0020
}
Material   Grass
{
   File               \Textures\Ground4.tga
   RepresentedSize    1.0 1.0 0.1
   Absorption         0.50
   Dispersion         0.40
}
Material   WallTextured
{
   File               Textures\WallPnt2b.tga
   RepresentedSize    25.0 25.0 0.1
   Absorption         0.10    // Arbitrary
   Dispersion         0.05    // Arbitrary
}
Material   WoodFloor
{
   File               Textures\Wood4.tga
   RepresentedSize    10.0 50.0 1.0
   Absorption         0.05
   Dispersion         0.01
}
Material   CeilingTextured
{
   ColorF             0.8 1.0 0.8
   Absorption         0.05
   Dispersion         0.01
}
//Ground faces in the structure
NumGrounds 1
Ground  Grass
{
   Thickness          0.5
// Floor coverage (xmin xmax ymin ymax)
   Dimensions         -10.0 310 -10.0 560
   Offset             -1.0
   Material           Grass
}
//Rooms in the structure
NumRooms   1
Room       MainRoom
{
   Offset     0.0 0.0 0.0
   Dimensions 300.0 550.0 120.0
   Rotation   0 0 0
   NumFaces   6
   Face       StandardWall
   {
       Name                  PocketDoors
       Thickness             3.0
       Side                  ahead
       Material              WallTextured
       NumWallOpenings       0
   }
   Face       StandardWall
   {
       Name                  Outside
       Thickness             3.0
       Side                  side
       Material              WallTextured
       NumWallOpenings       0
   }
   Face       StandardWall
   {
       Name                  Inside
       Thickness             3.0
       Side                  negside
       Material              WallTextured
       NumWallOpenings       0
   }
   Face       StandardWall
   {
       Name                  Entrance
       Thickness             3.0
       Side                  negahead
       Material              WallTextured
       NumWallOpenings       1
       WallOpening    FramedFrenchDoor
       {
          Type       FramedNormalDoor
          Offset             8.5 0.0
          Dimensions         36.0 80.0
          Material           Ceiling
          TrimMaterial       Floor
       }
   }
   Face       ceiling
   {
       Name                  theCeiling
       Thickness             3.0
       Material              CeilingTextured
       NumWallOpenings       0
   }
   Face    floor
   {
       Name                  theFloor
       Thickness             3.0
       Material              WoodFloor
       NumWallOpenings       0
   }
}
```

Table 33: Structure 1.3 Data File Example: A Simple House Structure

```
Structure 1.3
// warehouse.txt, Created for TDS Simulator
Units  feet
Measurements  outside
Offset 0.0 0.5 1.0
Rotation   0 0 0
NumMaterials  8
Material   MainRoomWalls
{
   File               Textures\Metal4.tga
   RepresentedSize    5.0 0.1 5.0
   Absorption         0.10
   Dispersion         0.05
}
Material   ConcreteFloor
{
   File               Textures\Concrete1.tga
   RepresentedSize    5.0 5.0 0.1
   Absorption         0.10
   Dispersion         0.05
}
Material   Ceiling
{
   File               Textures\Metal1.tga
   RepresentedSize    5.0 5.0 0.1
   Absorption         0.10
   Dispersion         0.05
}
Material   RoomWalls
{
   File               Textures\WallPnt1.tga
   RepresentedSize    3.0 3.0 3.0
   Absorption         0.10
   Dispersion         0.05
}
Material   Doors
{
   File               Textures\Wood2.tga
   RepresentedSize    3.0 3.0 3.0
   Absorption         0.10
   Dispersion         0.05
}
Material   Trim
{
   File               Textures\Metal2.tga
   RepresentedSize    3.0 3.0 0.1
   Absorption         0.10
   Dispersion         0.05
}
Material   RoomTrim
{
   ColorF             0.9 0.65 0.3
   Absorption         0.10
   Dispersion         0.05
}
Material   Grass
{
   File               Textures\Ground4.tga
   RepresentedSize    1.0 1.0 0.1
   Absorption         0.10
   Dispersion         0.05
}
NumGrounds 1
Ground  Grass
{
   Thickness          0.01
   Dimensions         -3.0 63.0 0.0 40.0
   Offset             -0.8
   Material           Grass
}
//Rooms in the structure
NumRooms   1
Room       MainRoom
{
   Offset             0.0 0.0 0.0
   Dimensions         60.0 40.0 30.0
   Rotation           0 0 0
   NumFaces           9
   Face    ceiling
   {
       Name                   theCeiling
       Thickness              0.5
       Material               Ceiling
       NumWallOpenings        0
   }
   Face    floor
   {
       Name                   theFloor
       Thickness              0.5
       Material               ConcreteFloor
       NumWallOpenings        0
   }
   Face    StandardWall
   {
       Name                   southwall
       Thickness              0.5
       Side                   negahead
       Material               MainRoomWalls
       NumWallOpenings        3
       WallOpening    FramedEmptyWindow
       {
           Type       FramedEmptyWindow
           Offset             46.0 15.0
           Dimensions         5.0 5.0
           Material           Doors
           TrimMaterial       Trim
       }
       WallOpening    FramedEmptyWindow
       {
           Type       FramedEmptyWindow
           Offset             11.0 15.0
           Dimensions         5.0 5.0
           Material           Doors
           TrimMaterial       Trim
       }
       WallOpening    FramedNormalDoor
       {
           Type       FramedNormalDoor
           Offset             5.0 0.0
           Dimensions         5.0 10.0
           Material           Doors
           TrimMaterial       Trim
       }
   }
   Face    StandardWall
   {
       Name                   sidewall
       Thickness              0.5
       Side                   side
       Material               MainRoomWalls
       NumWallOpenings        1
       WallOpening    FramedEmptyDoor
       {
           Type       FramedEmptyDoor
           Offset             15.0 0.0
           Dimensions         10.0 15.0
```

```
            Material             Doors
            TrimMaterial         Trim
        }
    }
    Face       StandardWall
    {
        Name                     aheadwall
        Thickness                0.5
        Side                     ahead
        Material                 MainRoomWalls
        NumWallOpenings          3
        WallOpening   FramedEmptyWindow
        {
            Type      FramedEmptyWindow
            Offset               10.0 15.0
            Dimensions           5.0 5.0
            Material             Doors
            TrimMaterial         Trim
        }
        WallOpening   FramedEmptyWindow
        {
            Type      FramedEmptyWindow
            Offset               45.0 15.0
            Dimensions           5.0 5.0
            Material             Doors
            TrimMaterial         Trim
        }
        WallOpening   FramedEmptyDoor
        {
            Type      FramedEmptyDoor
            Offset               51.0 0.0
            Dimensions           5.0 10.0
            Material             Doors
            TrimMaterial         Trim
        }
    }
    Face       StandardWall
    {
        Name                     negsidewall
        Thickness                0.5
        Side                     negside
        Material                 MainRoomWalls
        NumWallOpenings          0
    }
    Face       Wall
    {
        Name   RoomEntrance
        Thickness                0.5
        Height                   0.0 11.0
        Length                   0.0 15.0
        Rotation                 0
// Location of the wall, after rotation
        Offset                   0.0 10.0 0.0
```

```
        Material                 RoomWalls
// Add Doors and Windows to the Wall...
        NumWallOpenings          2
        WallOpening   FramedNormalDoor
        {
            Type      FramedNormalDoor
            Offset               1.0 0.0
            Dimensions           3.0 6.5
            Material             Doors
            TrimMaterial         RoomTrim
        }
        WallOpening   FramedEmptyWindow
        {
            Type      FramedEmptyWindow
            Offset               8.0 3.0
            Dimensions           4.0 4.5
            Material             Doors
            TrimMaterial         RoomTrim
        }
    }
    Face   Wall
    {
        Name                     RoomWall
        Thickness                0.5
        Height                   0.0 11.0
        Length                   0.0 10.0
        Rotation                 270.0
        Offset                   15.0 0.0 0.0
        Material                 RoomWalls
        NumWallOpenings          1
        WallOpening   FramedEmptyWindow
        {
            Type      FramedEmptyWindow
            Offset               4.0 3.0
            Dimensions           4.0 4.5
            Material             Doors
            TrimMaterial         RoomTrim
        }
    }
    Face   ceiling
    {
        Name                     roomCeiling
        Thickness                0.5
// Floor coverage (xmin xmax ymin ymax,
// Non-standard
        Dimensions               0.0 15 0.0 10
// Height of ceiling relative to typical
// Non-standard!!!
        Offset                   19.0
        Material                 RoomWalls
        NumWallOpenings          0
    }
}
```

Table 34: Structure 1.3 Data File Example: A Complicated Warehouse Structure

References

Antani, L., Chandak, A., Savioja, L., & Manocha, D. (2012, January). Interactive sound propagation using compact acoustic transfer operators. *ACM Transactions on Graphics,* 31(1), 7:1-7:12. doi:10.1145/2077341.2077348

Bacon, D., & Jarvis, D. (2007). *The speed and attenuation of sound.* Retrieved October 2007, from http://www.kayelaby.npl.co.uk/general_physics/2_4/2_4_1.html

Begault, D. R. (1994). *3-D sound for virtual reality and multimedia.* San Diego, CA: Academic Press Professional, Inc.

Blauert, J. (1997). *Spatial Hearing: The Psychophysics of Human Sound Localization.* MIT Press. doi:http://dx.doi.org/10.1121/1.392109

Burgress, D. (1992). Techniques for Low Cost Spatial Audio. *UIST '92 Proceedings of the 5th annual ACM symposium on User interface software and technology* (pp. 53-59). New York, NY: ACM UIST. doi:10.1145/142621.142628

Courchesne, L. (2007). Panoscope 360. Retrieved October 2007, from http://www.panoscope360.com/

Cowan, B., & Kapralos, B. (2011, November). A GPU-Based Method to Approximate Acoustical Reflectivity. *Journal of Graphics, GPU, and Game Tools,* 15(4), 210-215. doi:10.1080/2151237X.2011.619888

Cowan, B., Rojas, D., Kapralos, B., Collins, K., & Dubrowski, A. (2013, June). Spatial sound and its effect on visual quality perception and task performance within a virtual environment. *POMA,* 19, 050126. doi:10.1121/1.4798377

Flaherty, N. (1998). 3D audio: new directions in rendering realistic sound. *Electronic Engineering,* 49, 52, 55, 56.

Friedman, M. (1937, December). The use of ranks to avoid the assumption of normality implicit in the analysis of variance. *Journal of the American Statistical Association,* 32, 675–701. doi:10.2307/2279372

Funkhouser, T., Carlbom, I., Elko, G., Pingali, G., & Sondhi, M. (1998, July). A Beam Tracing Approach to Acoustic Modeling for Interactive Virtual Environments. *Computer Graphics (SIGGRAPH 98),* 21-32.

Funkhouser, T., Carlbom, I., Elko, G., Pingali, G., & Sondhi, M. (1999). Interactive Acoustic Modeling of Complex Environments. *The Joint Meeting of the 137th Regular Meeting of the Acoustical Society of America and the 2nd Convention of the European Acoustics Association: Forum Acusticum `99, Journal of the Acoustical Society of America,* 105(2).

Funkhouser, T., Jot, J.-M., & Tsingos, N. (2002, July). Sounds Good to Me! Computational Sound for Graphics, Virtual Reality, and Interactive Systems. *SIGGRAPH 2002,* Course Notes #45.

Funkhouser, T., Min, P., & Carlbom, I. (1999). Real-time Acoustic Modeling for Distributed Virtual Environments. *Computer Graphics (SIGGRAPH 99),* 365-374.

Funkhouser, T., Tsingos, N., & Jot, J.-M. (2004). Survey of Methods for Modeling Sound Propagation in Interactive Virtual Environment Systems. *Presence*.

Funkhouser, T., Tsingos, N., Carlbom, I., Elko, G., & Sondhi, M. (2002, September). Modeling Sound Reflection and Diffraction in Architectural Environments with Beam Tracing. *Forum Acusticum.*

Funkhouser, T., Tsingos, N., Carlbom, I., Elko, G., & Sondhi, M. (2004, February). A Beam Tracing Method for Interactive Architectural Acoustics. *Journal of the Acoustical Society of America,* 115(2), 739-756.

Gamper, H., & Lokki, T. (2011). Spatialisation in audio augmented reality using finger snaps. *Principles and Applications of Spatial Hearing,* 383–392.

Hammershøi, D. (2009). Localization Capacity of Human Listeners. Principles and *Applications of Spatial Hearing,* 3-13.

Hiipakka, J., Ilmonen, T., Lokki, T., Gröhn, M., & Savioja, L. (2001). Implementation issues of 3D audio in a virtual room. *Stereoscopic Displays and Virtual Reality Systems VIII,* 4297, 486-495. doi:0277-786X/01

Hull, J. (1999). *Surround Sound Past, Present, and Future.* Dolby Laboratories Inc.

ISO 9613-1. (1993). *Acoustics - Attenuation of sound during propagation outdoors - Part 1- Calculation of the absorption of sound by the atmosphere.* International Organization for Standardization. Retrieved from http://www.iso.org/iso/catalogue_detail.htm?csnumber=20649

Jin, C., Best, V., Lin, G., & Carlile, S. (2011). Spatial Unmasking of Speech Based on Near-Field Distance Cues. In *Biomedical Engineering* (pp. 3-20). Rijeka, Croatia: InTech Open Access Publishers.

Kahrs, M., & Brandenburg, K. (2002). Applications of *Digital Signal Processing to Audio and Acoustics.* (M. Kahrs, & K. Brandenburg, Eds.) New York, NY: Kluwer Academic Publishers.

Kinsler, L., Frey, A., Coppens, A., & Sanders, J. (2000). *Fundamentals of Acoustics* (4th ed.). New York, NY: John Wiley and Sons, Inc.

Laine, S., Siltanen, S., Lokki, T., & Savioja, L. (2009). Accelerated Beam Tracing Algorithm. *Applied Acoustics,* 70(1), 172–181. Retrieved from https://mediatech.aalto.fi/~samuli/

Lund, A., & Lund, M. (2014). Retrieved February 2014, from Laerd Statistics: https://statistics.laerd.com/

Martin, A., Jin, C., & Schaik, A. V. (2009, December). Psychoacoustic Evaluation of Systems for Delivering Spatialized Augmented-Reality Audio. *Journal of the Audio Engineering Society,* 57(12), 1016-1027. Retrieved from http://www.aes.org/e-lib/browse.cfm?elib=15234

Mehra, R., Raghuvanshi, N., Antani, L., Chandak, A., Curtis, S., & Manocha, D. (2013, April). Wave-Based Sound Propagation in Large Open Scenes using an Equivalent Source Formulation. *ACM Transactions on Graphics,* 32(2), 19:1-19:13. doi:10.1145/2451236.2451245

Microsoft. (2007). *Microsoft DirectSound - SetVolume.* Retrieved October 2007, from http://msdn.microsoft.com/archive/default.asp?url=/archive/en-us/directx9_c/directx/htm/idirectsoundbuffer8setvolume.asp

Neumann, G. (2007). *Georg Neumann GmbH Products.* Retrieved October 2007, from http://www.neumann.com/

Raghuvanshi, N., Snyder, J., Mehra, R., Lin, M., & Govindaraju, N. (2010, July). Precomputed Wave Simulation for Real-Time Sound Propagation of Dynamic Sources in Complex Scenes. *ACM Transactions on Graphics,* 29(4), 68:1-68:11. doi:10.1145/1778765.1778805

Resnick, R., Halliday, D., & Krane, K. (1992). *Physics* (4th ed.). John Wiley and Sons, Inc.

Schroder, D., & Pohl, A. (2013). Modeling (Non-)uniform scattering distributions in geometrical acoustics. *POMA,* 19, 015112. doi:10.1121/1.4800288

Sikora, M., Mateljan, I., & Bogunović, N. (2012). Beam Tracing with Refraction. *Archives of Acoustics,* 37(3), 301–316. doi:10.2478/v10168-012-0039-y

Siltanen, S. (2010). *Efficient Physics-Based Room-Acoustics Modeling and Auralization.* PhD thesis, Aalto University, Department of Media Technology. Retrieved from https://mediatech.aalto.fi/~saasilta/

Siltanen, S., Lokk, T., & Savioja, L. (2010, August). Room Acoustics Modeling with Acoustic Radiance Transfer. *International Symposium on Room Acoustics,* 376-381.

Siltanen, S., Lokki, T., & Savioja, L. (2006). Geometry Reduction in Room Acoustics Modeling. *Sixth International Conference on Auditorium Acoustics.* Retrieved from https://mediatech.aalto.fi/~ktlokki/Publs/59geom.pdf

Siltanen, S., Lokki, T., & Savioja, L. (2010, August). Rays or Waves - Understanding the Strengths and Weaknesses of Computational Room Acoustics Modeling Techniques. *Proceedings of the International Symposium on Room Acoustics,* 382-387.

Siltanen, S., Lokki, T., Kiminki, S., & Savioja, L. (2007). The Room Acoustic Rendering Equation. *Journal of the Acoustical Society of America, 122.* Retrieved from http://lib.tkk.fi/Diss/2010/isbn9789522482655/article3.pdf

Smith, S. W. (1997). *The Scientist and Engineer's Guide to Digital Signal Processing.* San Diego: California Technical Publishing. Retrieved from http://www.dspguide.com/

Southern, A., & Siltanen, S. (2013). A hybrid acoustic model for room impulse response synthesis. *POMA,* 19, 015113. doi:10.1121/1.4800212

Spiegel, M. R., Schiller, J. J., & Srinivasan, R. A. (2009). *Probability and Statistics* (3rd ed.). New York: McGraw-Hill.

Stephenson, U. M. (2013, June). The differences and though the equivalence in the detection methods of particle, ray and beam tracing. *POMA,* 16, 015111. doi:10.1121/1.4799952

Vorländer, M. (2011, November). Models and Algorithms for Computer Simulations in Room Acoustics. *International Seminar on Virtual Acoustics,* 72-82. Retrieved from http://www.upv.es/contenidos/ACUSVIRT/info/U0568398.pdf

Weinberger, S. (2007). Opening Soon: *Military Holodeck.* Retrieved August 2007, from http://blog.wired.com/defense/2007/08/building-a-mili.html

Wu, J.-R., Duh, C.-D., Ouhyoung, M., & Wu, J.-T. (1997). Head Motion and Latency Compensation on Localization of 3D Sound in Virtual Reality. *VRST 97 Proceedings of the ACM symposium on Virtual reality software and technology,* 15-20. doi:10.1145/261135.261140

McDermott, Scott D. Bachelor of Arts, University of Washington, Spring 1997; Master of Science, University of Louisiana at Lafayette, Fall 2001; Doctor of Philosophy, University of Louisiana at Lafayette, Fall 2014

Major: Computer Science
Title of Dissertation: An Analysis of Accurate, Real-Time Reproduction of 3D Acoustics in Virtual Environments
Dissertation Director: C.-H. Henry Chu
Pages in Dissertation: 175; Words in Abstract: 210

ABSTRACT

Many of the applications, virtual environments, and video games available to average computer users integrate stunning three-dimensional (3D) graphics and real-world visualizations. Developers spend an extraordinary amount of time and effort creating these immersive, realistic virtual environments, primarily focusing on the graphics components. Within these virtual realities, the user should easily perceive the locations of sound sources accurately, as well as the acoustic nature of the environment. However, for reasons of economy and simplicity, most developers apply readily available industry standards for generating pseudo-3D sounds in their applications. This research explores the shortcomings of these standards, proposes an effective alternative, and provides a detailed analysis of the various possible approaches.

This project includes a number of computationally efficient, physics-based 3D acoustics simulations, each of which will produce realistic aural reproductions. The primary goal is to evaluate and compare these algorithms against each other, non-3D sound reproduction, and the current industry standards (e.g. Microsoft's DirectX® pseudo-3D algorithm). We will test three hypotheses. First, users will find that physics-based 3D algorithms will render improved auralization reproductions compared against industry standards like DirectX® and/or OpenAL. Second, localization and spatialization will

improve with user training when using these algorithms. Finally, we should discover an unambiguous ranking system for the quality of each tested algorithm.

Keywords

Acoustics, real-time, 3D sound, virtual environments, spatialization, virtualization, auralization, localization.

Biographical Sketch

Scott McDermott grew up in the Pacific Northwest and attended University of Washington, earning a Bachelor of Arts in Music Technology. He relocated to Louisiana and earned a Master of Science in Computer Science at the University of Louisiana at Lafayette. While pursuing his Doctor of Philosophy in Computer Science at UL Lafayette, he has been an Instructor in the College of Business, Southeastern Louisiana University.

Mr. McDermott received the Louisiana Board of Regents Fellowship and a Research Assistantship during his PhD studies. He published in journals such as the Acoustical Society of America and the Association for Computer Machinery.