A Graph-Based Real-Time Spatial Sound Framework

by

Brent B. D. Cowan

A thesis submitted to the
School of Graduate and Postdoctoral Studies in partial
fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

Faculty of Science
Ontario Tech University
Oshawa, Ontario, Canada
January 2020

THESIS EXAMINATION INFORMATION

Submitted by: **Brent Cowan**

Doctor of Philosophy in **Computer Science**

Thesis title: A Graph-Based Real-Time Spatial Sound Framework

An oral defense of this thesis took place on December 9, 2019 in front of the following examining committee:

Examining Committee:

Chair of Examining Committee	Dr. Faisal Qureshi
Research Supervisor	Dr. Bill Kapralos
Examining Committee Member	Dr. Alvaro Quevedo
Examining Committee Member	Dr. Karen Collins
University Examiner	Dr. Richard Pazzi
External Examiner	Dr. George Tzanetakis, University of Victoria

The above committee determined that the thesis is acceptable in form and content and that a satisfactory knowledge of the field covered by the thesis was demonstrated by the candidate during an oral examination. A signed copy of the Certificate of Approval is available from the School of Graduate and Postdoctoral Studies.

Author's Declaration

I hereby declare that this thesis consists of original work of which I have authored. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize the Ontario Tech University to lend this thesis to other institutions or individuals for the purpose of scholarly research. I further authorize the Ontario Tech University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research. I understand that my thesis will be made electronically available to the public.

The research work in this thesis that was performed in compliance with the regulations of Ontario Tech University's Research Ethics Board Committee under **REB file number 14875**.

BRENT B. D. COWAN

Statement of Contributions

The spatial sound framework that is the focus of this thesis, contains components that have previously been published in peer reviewed journals. I was responsible for the development and testing of those components. The convolution, occlusion, and reverberation work is detailed in Chapters 3, 4, and 5 respectively. Research studies were conducted using an early version of the framework. Those publications are briefly discussed in Chapter 8. In addition, this thesis contains new unpublished work included in Chapters 6 and 7 (the graph-based sound propagation algorithm, and the user study conducted to evaluate it respectively). For the publications where I am first author, I took a leading role in the work (completed all the development), and contributed to the writing of the paper.

The following peer reviewed publications have resulted from this thesis work and are discussed within its pages.

Journal Publications

1. **B. Cowan**, D. Rojas, B. Kapralos, F. Moussa, A. Dubrowski Effects of sound on visual realism perception and task performance. *Visual Computer*, 31(9):1207-1216, 2015. (**2018 IF: 1.415**).

2. **B. Cowan**, and B. Kapralos. A real-time, GPU-based method to approximate acoustical reverberation effects. *Journal of Graphics, GPU, and Game Tools,* 15(4):210-215, 2011.

3. **B. Cowan**, and B. Kapralos. GPU-based real-time acoustical occlusion modeling. *Virtual Reality*, 14(3):183-196, 2010. (**2018 IF: 2.906**)

4. **B. Cowan**, and B. Kapralos. GPU-based one-dimensional convolution for real-time spatial sound generation. *Loading… Feature Issue: FuturePlay* 2008 Edition. 3(5):1-14, 2009.

Refereed Conference Proceedings

5. **B. Cowan** and B. Kapralos. Interactive Rate Acoustical Occlusion/Diffraction Modeling for 2D Virtual Environments & Games. In *Proceedings of the 6th International Conference on Information, Intelligence, Systems and Applications (IISA2015)*, Ionian University, Corfu, Greece, July 6-8, 2015, pp. 1-6.

6. D. Rojas, **B. Cowan**, B. Kapralos, K. Collins, and A. Dubrowski. The effect of sound on visual quality perception and task completion time in a cel-shaded serious gaming virtual environment. In *Proceedings of the 7th IEEE International Workshop on Quality of Multimedia Experience*, Messinia, Greece, May 26-29 2015, pp. 1-6.

7. D. Rojas, **B. Cowan**, B. Kapralos, K. Collins, and A. Dubrowski. The effect of contextual sound cues on visual fidelity perception. *Medicine Meets Virtual Reality* 2014, Manhattan Beach, CA, USA, February 20-22, 2014. Appears in *Studies in Health Technology and Informatics*,196:346-352, 2014.

8. **B. Cowan**, and B. Kapralos. Interactive rate virtual sound rendering engine. In *Proceedings of the 18th IEEE International Conference on Digital Signal Processing (DSP 2013)*. Santorini, Greece, July 1-3, 2013, pp. 1-6.

9. **B. Cowan**, D. Rojas, B. Kapralos, K. Collins, and A. Dubrowski. Spatial sound and its effect on visual quality perception and task performance within a virtual environment. *Proceedings of the 21st International Congress on Acoustics*, Montreal, Canada, June 2-7, 2013, pp. 1-7.

10. **B. Cowan**, and B. Kapralos. GPU-based acoustical occlusion modeling with acoustical texture maps. In *Proceedings of ACM Audio Mostly* 2011. Coimbra, Portugal, September 7-9, 2011.

11. **B. Cowan**, and B. Kapralos. GPU-based acoustical diffraction modeling for complex virtual reality and gaming environments. In *Proceedings of the Audio Engineering Society 41st International Conference: Audio for Games*. London, UK, February 2-4, 2011.

12. **B. Cowan**, and B. Kapralos. Real-time acoustical diffraction and first order specular reflection modeling using the GPU. In *Proceedings of the 10th Western Pacific Acoustics Conference*. Beijing, China, September 21-23, 2009.

13. **B. Cowan**, and B. Kapralos. Spatial sound for video games and virtual environments utilizing real-time GPU-based convolution. In *Proceedings of the ACM FuturePlay 2008 International Conference on the Future of Game Design and Technology*. Toronto, Ontario, Canada, November 3-5, 2008, pp 166-172.

Extended Abstracts

14. **B. Cowan**, B. Kapralos, and K. Collins. Spatial audio modelling used to provide artificially intelligent characters with realistic sound perception. *GameSoundCon 2016*, Los Angeles, CA, September 27-28, 2016.

15. **B. Cowan** and B. Kapralos. GPU-based acoustical occlusion modeling for virtual environments and games. In *Proceedings of the IEEE Games Innovation Conference (IGIC)* 2013, Vancouver, Canada, September 23-25, 2013, pp. 48-49.B.

16. **B. Cowan**, and B. Kapralos. Real-time GPU-based convolution: A follow-up. In *Proceedings of the ACM FuturePlay @ GDC Canada 2009 International Conference on the Future of Game Design and Technology*. Vancouver, British Columbia, Canada, May 12-13, 2009, pp. 25-26.

Additional Publications

The following peer reviewed publications are indirectly related to the thesis topic. For the publications where I am first author, I took a leading role in the work (completed all the development), and contributed to the writing of the paper.

Journal Publications

17. M. Nguyen, M. Melaisi, B. **Cowan, B**, A. Uribe Quevedo, and B. Kapralos). Low-end haptic devices for knee bone drilling in a serious game. *World Journal of Science, Technology and Sustainable Development*, 14(2/3), 241-253, 2017.

18. **B. Cowan**, H. Sabri, B. Kapralos, M. Porte, D. Backstein, S. Cristancho, and A. Dubrowski. A serious game for total knee arthroplasty procedure education and training. *Journal of Cybertherapy and Rehabilitation*, 3(3):285-298, 2010.

Refereed Book Chapters

19. **B. Cowan**, and B. Kapralos (2017). An overview of serious game engines and frameworks. In *Recent Advances in Technologies for Inclusive WellBeing: Wearables, Virtual Interactive Spaces (VIS) / Virtual Reality, Emotional Robots, Authoring tools, and Games* (pp. 15-38). Springer, Cham.

Refereed Conference Proceedings

20. **B. Cowan**, S. Khattak, B. Kapralos, and A. Hogue. Screen Space Point Sampled Shadows. In *Proceedings of the IEEE Games Entertainment and Media (GEM 2015)*, Toronto, Canada, October 14-16, 2015, pp. 198-204.

21. **B. Cowan,** B. Kapralos, F. Moussa, and A. Dubrowski. The serious gaming surgical cognitive education and training framework and SKY script scripting language. In *Proceedings of the 8th International Conference on Simulation Tools and Techniques,* Athens, Greece, August 24-26, 2015, pp. 308-310.

22. S. Khattak, **B. Cowan**, I. Chepurna, and A. Hogue. A real-time reconstructed 3D environment augmented with virtual objects. In P*roceedings of the IEEE Games Entertainment and Media (GEM) 2014*, Toronto, Ontario, Canada, October 22-24, 2014, pp. 1-8.

23. D. Rojas, **B. Cowan**, B. Kapralos, and A. Dubrowski. Gamification and health professions education. *In Proceedings of the IEEE Games Entertainment and Media (GEM) 2014*, Toronto, Ontario, Canada, October 22-24, 2014, pp. 1-2.

24. **B. Cowan**, and B. Kapralos. A survey of engines for serious games development. *The 14th IEEE International Conference on Advanced Learning Technologies (ICALT2014)*, Athens, Greece, July 7-10, 2014, pp. 662-664.

25. **B. Cowan**, H. Sabri, B. Kapralos, S. Cristancho, F. Moussa, and A. Dubrowski. SCETF: Serious game surgical cognitive education and training framework. In *Proceedings of the Third IEEE International Games Innovation Conference*, City of Orange, CA, USA, November 2-4, 2011, pp. 130-133.

26. **B. Cowan**, H. Sabri, B. Kapralos, F. Moussa, S. Cristancho, and A. Dubrowski. A serious game for off-pump coronary artery bypass surgery procedure training. *Medicine Meets Virtual Reality 18*, Newport Beach, CA, USA, February 8-12, 2011. Appears in *Studies in Health Technology and Informatics*, 163:147-149, 2011.

27. H. Sabi, **B. Cowan**, B. Kapralos, F. Moussa, S. Cristancho, and A. Dubrowski. Off-pump coronary artery bypass surgery procedure training meets serious games. In *Proceedings of the International Symposium on Haptic Audio-Visual Environments and Games*. Phoenix, AZ. USA, October 16-17, 2010, pp. 123-127.

28. H. Sabi, **B. Cowan**, B. Kapralos, M. Porte, D. Backstein, and A. Dubrowski. Serious games for knee replacement surgery procedure education and training. *World Conference on Educational Sciences (WCES) 2010*. Istanbul, Turkey, February 4-8, 2010. Appears in *Procedia - Social and Behavioral Sciences,* 2(2):3483-3488.

Extended Abstracts

29. **B. Cowan**, B. Kapralos, S. Cristancho, F. Moussa, and A. Dubrowski. The serious game surgical cognitive education and training framework. *Graphics Interface* 2012 Poster Presentation, Toronto, Ontario, Canada, May 28-30, 2012, pp 1-2.

30. **B. Brent**, B. Kapralos, A. Hogue, S. Khattak. Screen Space Point Sampled Shadows. Poster presentation - *Graphics Interface 2012*, Toronto, Ontario, Canada, July 16, 2012, pp. 1-2. (**Awarded Best Poster**)

31. D. Rojas, J. J. H. Cheung, **B. Cowan**, B. Kapralos, and A. Dubrowski. Serious games and virtual simulations debriefing using a social networking tool. In *Proceedings of the Computer Games Multimedia and Allied Technology (CGAT 2012)* Conference, Bali, Indonesia, May 7-8, 2012, pp. 1-2.

32. **B. Cowan**, and B. Kapralos. A simplified level editor. In *Proceedings of the Third IEEE International Games Innovation Conference*, November 2-4, 2011, City of Orange, CA., USA, pp. 53-54.

33. **B. Cowan**, M. Shelley, H. Sabri, B. Kapralos, A. Hogue, M. Hogan, M. Jenkin, S. Goldsworthy, L. Rose and A. Dubrowski. Interprofessional care simulator for critical care education. In *Proceedings of the ACM FuturePlay 2008 International Conference on the Future of Game Design and Technology*. Toronto, Ontario, Canada, November 3-5, 2008, pp 260-261.

Refereed Abstracts

34. **B. Cowan**, H. Sabri, B. Kapralos, M. Porte, D. Backstein, and A. Dubrowski. A serious game for total knee arthroplasty (replacement) surgery procedure education and training. In Extended Proceedings of *ACM Fun and Games* 2012, September 4-6, 2012, Toulouse, France.

35. **B. Cowan**, H. Sabri, B. Kapralos, M. Porte, D. Backstein, and A. Dubrowski. A serious game for knee replacement surgery procedure training. *GRAND 2012*, Montreal, Quebec, Canada, May 2, 2012.

36. **B. Cowan**, H. Sabri, B. Kapralos, F. Moussa, S. Cristancho, and A. Dubrowski. Serious Games: 2D vs 3D? *11th Annual International Meeting on Simulation in Healthcare*. New Orleans, LA., USA, January 21-26, 2011.

37. H. Sabri, **B. Cowan**, B. Kapralos, M. Porte, D. Backstein, S. Cristancho, and A. Dubrowski. A serious game for total knee arthroplasty procedure education and training. *The Richard K. Reznick Wilson Centre Research Day 2010*. Toronto, Canada, October 22, 2010.

38. **B. Cowan**, H. Sabri, B. Kapralos, F. Moussa, S. Cristancho, and A. Dubrowski. A serious game for off-pump coronary artery bypass surgery procedure training. Poster presentation - *The Richard K. Reznick Wilson Centre Research Day 2010*. Toronto, Canada, October 22, 2010.

Abstract

Given the importance of sound, and our ability to localize sound sources in three-dimensions in the real world, incorporating spatial sound in virtual environments can help increase realism, improve the sense of “presence”, or “immersion”, improve task performance, and improve navigation speed and accuracy. However, despite advancements in the real-time simulation of spatial sound, current solutions are computationally expensive and often rely on specialised hardware, and as a result, spatial sound cues are often overlooked in virtual environments and games notwithstanding their importance.

Here, a novel spatial sound rendering framework that approximates spatial sound for virtual environments, yet conforms to physical sound propagation rules/laws, is introduced. The framework employs graphs in order to reduce computation time and each node in the graph is processed in parallel using the graphics processing unit (GPU) making this method suitable for real-time virtual immersive applications such as video games and virtual simulations. Results of a user study that was conducted with human participants (the intended users of any spatial sound method), to test the effectiveness of the spatial sound framework introduced here, indicates it does lead to improved player performance over traditional panning (binaural sound cues only) and ray-cast occlusion in a 3D first-person video game.

Keywords: Spatial sound; real-time; acoustic modeling; sound propagation; GPU

Acknowledgments

I would like to thank my supervisor, Dr. Bill Kapralos, for all of his help and encouragement throughout this project, and the many smaller research projects that led to this point. The support and advice that he has provided over the years has been greatly appreciated.

I would also like to thank Dr. Alvaro Joffre Uribe Quevedo, Dr. Richard Pazzi, Dr. Faisal Qureshi, Dr. Karen Collins, and Dr. George Tzanetakis. Dr. Quevedo (Faculty of Business and Information Technology, Ontario Tech University) and Dr. Collins (Department of Communication Arts and the Department of English Language and Literature, University of Waterloo) served on my thesis defense committee and provided useful feedback regarding my thesis. Dr. Pazzi (Faculty of Business and Information Technology, Ontario Tech University) served as the university examiner, and Dr. Tzanetakis (Faculty of Science, University of Victoria) served as the external examiner, both provided me with useful feedback. Dr. Qureshi (Faculty of Science, Ontario Tech University) served as Chair during my thesis defense.

The financial support of the Natural Sciences and Engineering Research Council of Canada (NSERC) in the form of the Alexander Graham Bell Canada Graduate Scholarship is greatly appreciated. In addition, the financial support of the Ontario Graduate Scholarship (OGS) Program in the form of a doctoral OGS scholarship is gratefully acknowledged.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

2D	Two Dimensional
2.5D	Graphically rendered in 3D, game-play is restricted to a 2D plane
3D	Three Dimensional
AI	Artificial Intelligence
API	Application Programming Interface
BEM	Boundary Element Method
BCC	Body-Centered Cubic
CAD	Computer Aided Design
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DFT	discrete Fourier Transform
DSP	Digital Signal Processing
EAX	Environmental Audio Extensions
EFX	Environment Effects Extension
FFT	Fast Fourier Transform
fps	Frames Per Second
FPS	First-Person Shooter
GLSL	OpenGL shading language
GPU	Graphics Processing Unit
G-SpAR	GPU-Spatial Audio Renderer
HLSL	High Level Shading Language
HRTF	Head-Related Transfer Function
HUD	Heads-Up Display
ILD	Interaural Level Difference
ITD	Interaural Time Difference
MIDI	Musical Instrument Digital Interface
NPC	Non-Player Character
PBSL	Propagation based sound localisation
PC	Personal Computer
RIR	Room Impulse Response
RGB	Red, Green, and Blue
UTD	Uniform Theory of Diffraction
WYSIWYG	What You See Is What You Get
VR	Virtual Reality

Chapter 1 Introduction

Given the importance of sound, and our ability to localize sound sources in three-dimensions in the real world, virtual environments (including virtual simulations, serious games, and video games in general), where often times the goal is to replicate the real world, should include realistic spatial sound cues. Spatial (three-dimensional or 3D) sound systems allow a listener to perceive the relative position of sound sources. Spatial sound systems add the effect of the interaction of sound sources with the acoustic structure of the environment. Spatial sound has three main components: i) one or more sound sources, ii) the environment, and iii) one or more listeners [Kapralos, Jenkin, & Milios, 2003]. Incorporating realistic spatial sound in virtual environments can help increase realism [Antani et al. 2012], improve the sense of "presence", or "immersion" experienced by the user over visual cues alone [Nordahl, Turchet, & Serafin 2011], improve task performance [Zhou et al. 2007] (although with respect to video games, there are still various questions related to how sound can contribute to immersion [Chandak 2011]). Presence or immersion in the context of virtual environments, refers to a sense of "being there," in the user's mind [Pausch, Proffitt, & Williams 1997; Bowman & McMahan 2007]. Realistic spatial sound has also been shown to improve task performance [Zhou et al. 2007], convey information that is difficult to convey using other modalities including vision [Zhou et al. 2007], and improve navigation speed and accuracy [Mehra et al. 2015; Makino, Ishii, & Nakashizuka 1996]. Finally, realistic spatial sound can provide a very realistic sense of being remotely immersed in the presence of people, musical instruments, and environmental sounds [Zhou et al. 2007]. At the time of this writing, spatial sound including

propagation based sound localisation (PBSL) is not a common feature in current 3D virtual environments and video games in particular, where graphical realism has traditionally been the priority [Eriksson 2017].

Spatial sound technologies attempt to simulate the acoustics of a virtual environment for a human listener [Zhang et al. 2017]. Some Computer Aided Design (CAD) tools provide acoustical modeling in order to simulate sound propagation for 3D architectural environments. Acousticians can use CAD software to evaluate the acoustical properties of a theatre before it is built, in order to ensure that the design will provide an adequate listening experience for the entire audience [Shtrepi et al. 2015]. Realistic acoustical modeling is a computationally intensive task for anything other than the most trivial of environments. Virtual environments (and video games in particular), are often very complex and dynamic (constantly changing in response to user interaction), and may contain both indoor and outdoor areas, with many sounds located throughout the 3D world; realistic spatial sound if thus often ignored.

Of course, not all sounds in a virtual environment/video game need to be spatialized. Interface sounds such as beeps or clicks activated in response to the user interacting with the menus may not require a 3D location in the world. It may also be desirable for background sounds such as music or ambience to be void of 3D positioning. Instead, non-spatialized sounds can have a directional ambiguity which may give the user the impression that the sound is coming from every direction.

A sound source in a virtual environment is a sound file (a one dimensional array of samples used to recreate the waveform) that is played at a location in 3D space [Fmod 2019]. The wave form is stored as a sound file such as a WAV, Ogg, or MP3. Stereo (stereophonic) sounds have an independent right and left channel which can be used to embed the sound file with spatial attributes such as a location relative to the listener [Davies 2015]. Music is typically played in stereo so that each instrument and voice can be panned to the left or right. Offsetting the instruments and voices that make up a piece of music is referred to as "separation". Separation helps the listener to be able to focus on a single instrument or voice within the mix. When recording music, it is common to place the vocals and low frequency instruments such as drums and bass guitars near the centre and subtly pan other instruments left or right, with the goal of reproducing the spatial layout of a band performing on a stage [Mansbridge, Finn, & Reiss 2012]. Monaural (mono) sound files store only one channel, and therefore do not have any spatial properties embedded in the file format [Davies 2015]. Sound application programming interfaces (APIs) such as Fmod, split the mono sound into the right and left channels that the player hears. The sound is processed using digital signal processing (DSP) techniques in real-time to provide it with a distance and direction relative to the listener, and to imbue it with characteristics of the environment such as reverberation [Pulkki et al. 2018].

1.1 Human Auditory Localisation

An understanding of human auditory perception is fundamental to developing effective spatial sound for virtual environments [Kapralos et al. 2008]. As a result, in this section, a brief overview of human auditory localisation will be provided.

1.1.1 Attenuation

Attenuation is a fundamental characteristic of sound, one that is commonly understood by most people. Sounds become quieter or less intense with increased distance. The attenuation, or the reduction of sound intensity with increased distance between the sound source and a listener, is fundamental to our perception of distance [Plack 2018]. Sound is created when a vibrating object (such as our vocal cords) vibrates the propagation medium (air). The medium we are most familiar with is air, however, sound can also propagate to some degree through many different types of materials including gasses, liquids, and solids. The air surrounding vibrating objects is compressed and expanded causing rapid fluctuations in the air pressure. The fluctuations cause the surrounding air to fluctuate as well. If unobstructed, a transverse wave of sound energy would radiate outward at the speed of sound in a roughly spherical shape. As the surface area of the sound increases, the intensity of the sound must decrease due to the initial energy being spread over larger area [Plack 2018]. This loss of sound energy due to attenuation can be approximated using the inverse square law:

$$I = r / (r + D^2), \qquad (1.1)$$

where **I** is the Intensity predicted at Distance **D**, and ***r*** is the rate of attenuation or range. The value assigned to ***r*** depends on the intensity of the sound. For example, a gunshot can be heard from a greater distance than a whisper. Increasing the value of ***r*** extends the distance that a sound can be heard from. Distance is measured in meters by default in most sound engines (audio APIs used for game development), including Fmod. Ideally, all of the sound effects used in a game or simulation should be recorded at the same distance in an anechoic chamber (a room lined with materials that absorb sound preventing reverberation) so that the reverberation can be rendered by the game's audio system in response to the changing game environment. However, this is not always feasible because some sounds need to be recorded from a safe distance such as explosions, other sounds can only be recorded while they are in motion (vehicles), and some sound effects are generated in software and not recorded at all. It is the responsibility of the sound designer to then adjust the volume and attenuation of each sound effect in proportion to the other sounds in the environment, a process that is more art than science. For example, a sound designer might apply an unrealistic attenuation model to an important sound in order to improve its audibility.

The linear attenuation model outlined in Equation 1.2 below, is often used in video games despite being less realistic. The attenuation for each sound is controlled by two parameters set by the sound designer, a minimum and maximum distance. If the distance between the listener and the sound source is less than the minimum, the sound is played at full volume. If the distance is greater than the maximum, the sound is given a volume of zero. When the distance lies between the minimum and

maximum value set by the designer, the volume is attenuated linearly as shown in Equation 1.2 [Fmod 2019].

$$I = 1 - (D-M_1) / (M_2-M_1), \quad (1.2)$$

where intensity **I** is clamped to a range between zero and one, **D** is the distance between the listener and the sound source, $\mathbf{M_1}$ represents the minimum distance, or the distance at which attenuation begins, and $\mathbf{M_2}$ represents the maximum distance, or the distance at which the intensity decreases to zero. Figure 1.1 graphically illustrates the difference between linear and inverse attenuation.

Playback is halted for sound effects with zero volume, making the channel available for other sounds, while potentially reducing the processing requirements of the system. Sound cards have a limited number of channels which limits the number of sounds that can be played concurrently. With an inverse distance attenuation model (attenuation based on the inverse square law), the intensity (and thus, the corresponding volume) will never reach zero at any distance. Figure 1.1 shows the difference between the linear and inverse attenuation models (the values used to construct the graph were arbitrarily selected). Linear sounds (sound effects utilizing the linear attenuation model), have a 3D spherical area of effect with a radius equal to the maximum ($\mathbf{M_2}$) specified.

Sounds only need to be played, and therefore require resources when the listener is inside their area of effect. A large-scale virtual environment may contain hundreds of objects generating sounds, although only a small subset of them need

to be played at any given time. The linear attenuation model simplifies the process of determining which sounds to play on the limited number of sound channels. The linear attenuation model also provides the sound designer with greater control over the audibility of individual sounds.

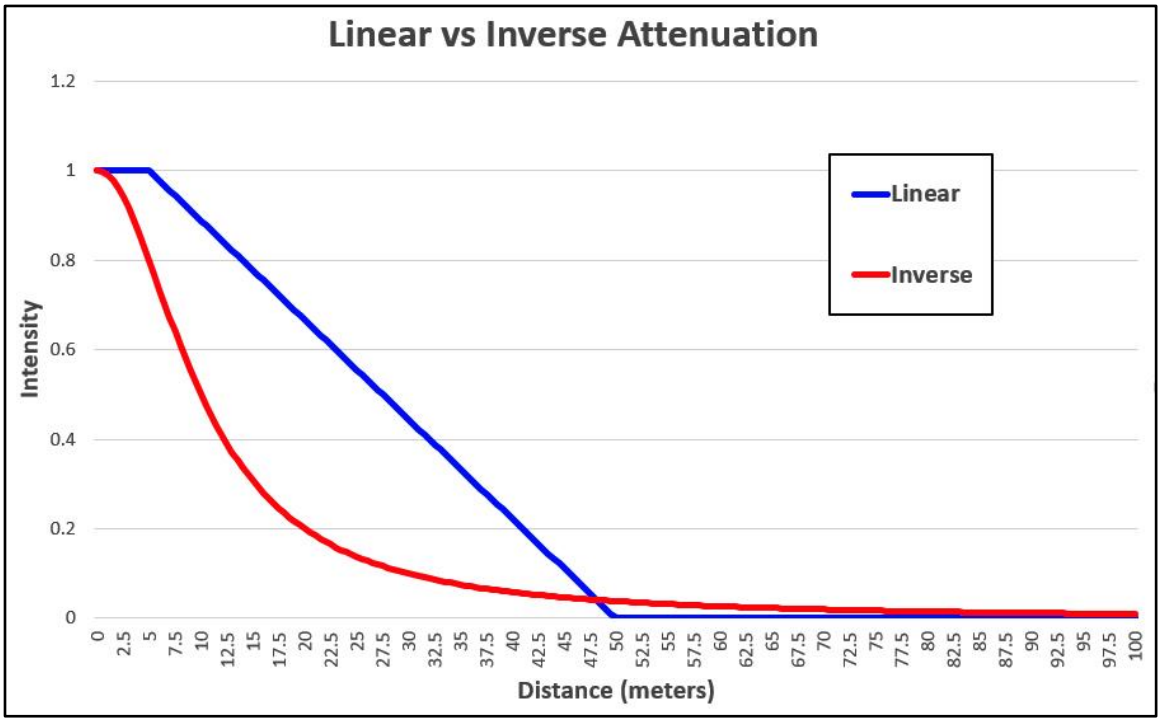


Figure 1.1: Sound intensity by distance with inverse and linear attenuation models. The inverse attenuation curve was created by setting parameter r equal to 100. The linear attenuation was graphed using a minimum (M_1) of 5 and a maximum (M_2) of 50.

1.1.2 Binaural Sound Cues

When a sound source is located to the left or right of a listener, the listener's head blocks or occludes some of the sound from reaching the far ear (see Figure 1.2). The listener's head is essentially casting an acoustic shadow. The result is an interaural sound pressure level difference (ILD). The ILD is less noticeable for lower frequencies because low frequency sounds diffract more, bending around the listener's head rather than being blocked [Middlebrooks & Green 1991]. In the late

19th century, Lord Rayleigh determined that the ILD for frequencies lower than 1,000 Hz are negligible and therefore ILD alone was not sufficient to localise low frequency sounds. Lord Rayleigh proposed that the interaural phase difference must play a role in sound localisation [Rayleigh 1907]. Sound will reach the closer ear first causing the phase at the near and far ears to be out of sync. Unlike the ILD, this slight difference in the interaural time (ITD) cannot be perceived consciously, but it is used by our auditory system to estimate the direction of a sound source. ILD and ITD are binaural sound cues, meaning that the information they provide is based on the difference between the signals received by the right and left ear.

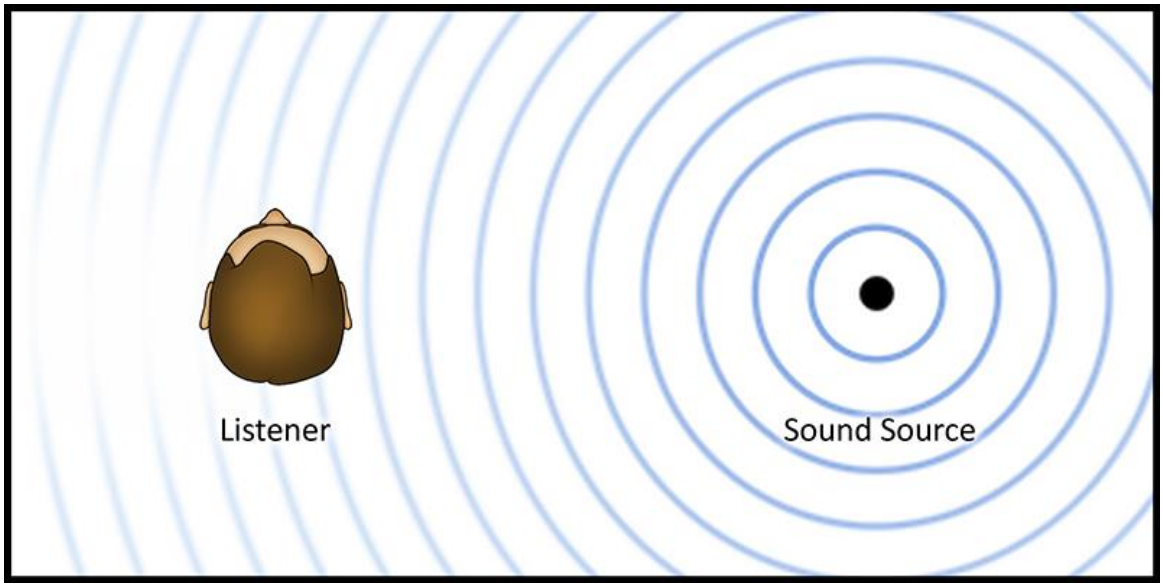


Figure 1.2: ILD and ITD caused by the orientation of the listener's head relative to a sound source.

The duplex theory, which Lord Raleigh published in 1907, [Raleigh 1907] assumes that human sound localization is based on the binaural sound cues only (ILD and ITD). Furthermore, it assumes that the shape of the listener's head is a perfect sphere with no external ears (pinnae). The duplex theory alone cannot explain how

a listener is able to differentiate sounds that are located in front, behind, above, or below them. In addition, it also does not explain how a person who is deaf in one ear retains the ability to localise sound [Slattery & Middlebrooks 1994].

1.1.3 Occlusion and Diffraction

Focusing on the situation when the direct line of straight propagation between the sound source and the listener is occluded, through the phenomenon of diffraction, sound is able to "bend" around an obstacle that lies directly in the line of straight propagation. This allows us to hear sounds around corners and around barriers and is thus an important aspect of sound propagation, given that often times in the real world the sound source and listener will be occluded by some object/obstacle [Cremer & Müller 1978]. Diffraction is dependent on both wavelength and obstacle/surface size, increasing as the ratio between wavelength and obstacle size is increased. In other words, diffraction will typically be greater for lower frequency sounds and when the obstacles are small. The frequency spectrum of audible sound ranges from approximately 20 Hz – 20 kHz, corresponding to wavelengths ranging from 17 m down to 0.017 m (with a velocity of $v_c = 343\ ms^{-1}$ for sound in air and a frequency of f Hz, wavelength $\lambda = v_c/f$ [Cremer & Müller 1978]). Since, the dimensions of many of the objects/surfaces encountered in our daily lives are of the same order of magnitude as the wavelength of audible sounds, diffraction is an elementary means of sound propagation especially when there is no direct path between the sound source and the listener, such as in buildings [Tsingos et al. 2001].

1.1.4 Reflection and Reverberation

When a sound wave comes into contact with a surface it is reflected similar to the way that light reflects. If the reflecting surface is smooth and flat, the angle of incidence equals the angle of reflection (following the law of reflection). The resulting specular reflection is both coherent and predictable. However, where the surface is rough (contains bumps, groves, or imperfections) relative to the wavelength, the sound energy is scattered and the reflection is diffuse. Accurately modelling the reflection of sound is complex due to the large variation of possible wavelengths. A single sound source can simultaneously emit sound waves containing a variety of wave lengths ranging between [Doelle 1972].

Based on the material properties of the surface, some of the sound energy will be absorbed with each reflection. In addition, some of the sound energy may be allowed pass through the object (transmitted) (see Equation 1.3) [Doelle 1972]. Reflection, absorption, and transmittance are all frequency dependent however, frequency dependence is often ignored for real-time models which trade accuracy for processing efficiency.

$$I_{incident} = I_{reflected} + I_{absorbed} + I_{transmitted}, \qquad (1.3)$$

Reverberation is the combination of many reflected sound waves over time. The direct sound will reach the listener first provided that the line-of-sight between the sound source and listener is not obstructed (occluded). The direct sound wave is followed by early reflections which are sound waves that have been reflected by surfaces in the environment. These early reflections (also known as primary or

first-order reflections) arrive within the first 80 – 100 milliseconds following the direct sound and describe the environment around the listener [Parfit 2005]. For example, a lack of early reflections indicates that the listener is either in an open outdoor area or that they located in an environment made up of materials that absorb sound (anechoic). Loud reflections denote the presence of large reflective surfaces such as walls. The number of early reflections coincides with the number of reflecting surfaces, providing the listener with information regarding the shape or uniformity of the environment. The delay between the direct and the reverberant energy provides a sense of the scale to the listener [Hakala 2019].

The early reflections are followed by late reflections, that is, reflection of reflections. Each time a wave is reflected, it loses some of its energy due to absorption and transmittance. There are exponentially more late reflections which blend together and can no longer be heard individually [Savioja & Svensson 2015]. Late reflections (also referred to as the reverberation tail) sound like faint background noise that quickly fades away. Figure 1.3 illustrates the concept of acoustical reflection. The lines show the path of direct and reverberant sound waves traveling from a sound source to a listener in a small rectangular room.

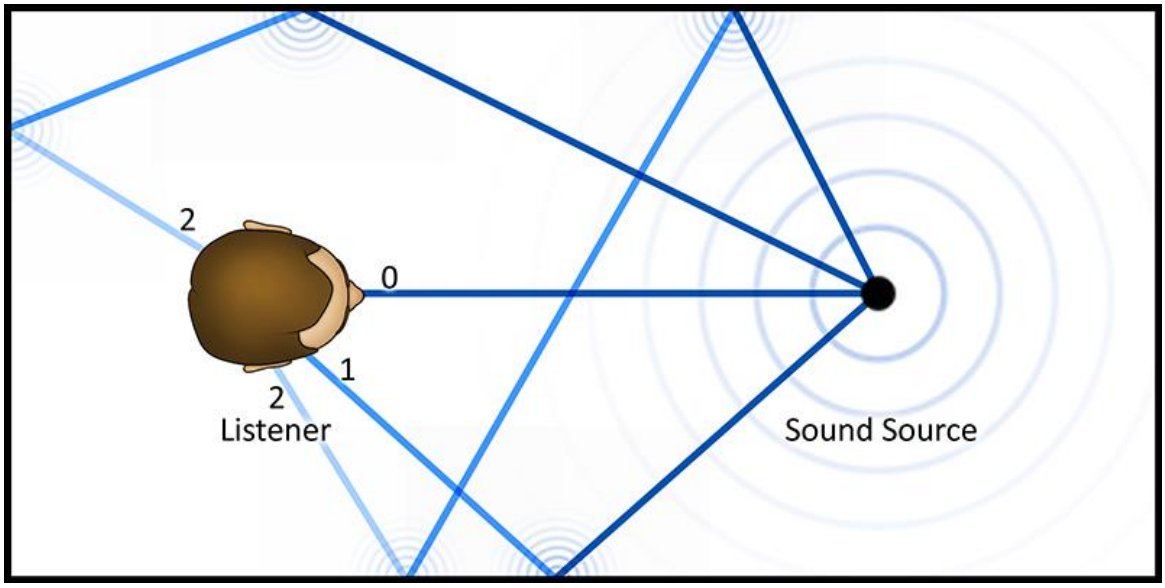


Figure 1.3: Direct and reverberant sound reaching the listener. The number indicates the reflection "order", that is, the number of times the sound wave is reflected before reaching the listener. The lines become lighter with each reflection which is used to represent a decrease of energy due to absorption of sound energy by the medium (e.g., air), and any surface it may encounter.

The Fmod sound engine refers to early reflections as the "dry mix" and simulates them by playing a number of delayed and attenuated versions of the direct sound. Fmod refers to late reflections as the "wet mix" and simulates them by adding noise with a volume and frequency based on the direct sound. Fmod's reverberation is parametric, meaning that it is applied algorithmically based on a few input parameters. It is still common for reverberation settings to be manually applied by a designer rather than simulating sound propagation based on the in-game geometry [Kastbauer 2010]. Parametric reverberation is the most common method used to apply reverberation to sounds in real-time because it is very efficient and does not require specialised hardware (hardware containing features that are uncommon at the consumer level). Alternatively, filters may be computed or

recorded in real-world locations however, applying these filters to every sample of a sound file requires more processing time than traditional parametric methods.

1.1.5 Head Related Transfer Function

Collectively, "the process of rendering audible, by physical or mathematical modeling, the sound field of a sound source in space, in such a way as to simulate the binaural listening experience at a given position in the modeled space" is known as auralization [Kleiner et al. 1993]. The goal of auralization is to recreate a particular listening environment, taking into account the acoustics of the environment and the characteristics of the listener (see Kapralos et al. [2008] for a thorough review of spatial sound and auralization). Auralization is typically accomplished by determining the binaural room impulse response (BRIR). The BRIR represents the response of a particular acoustical environment to sound energy and captures the room acoustics for a particular sound source and listener configuration. Once obtained, the BRIR can be used to filter a monaural sound through a convolution process described by Equation 1.4

$$s[n] \times r[n] = \sum_{k=-\frac{N}{2}+1}^{N/2} s[n-k]t[k] \leftrightarrow S(f)R(f), \qquad (1.4)$$

where $\boldsymbol{s}$ is the input signal, $\boldsymbol{r}$ is the filter, $\boldsymbol{n}$ denotes the sample index, $\boldsymbol{S}$ and $\boldsymbol{R}$ denote the discrete Fourier transform (DFT) of $\boldsymbol{s}$ and $\boldsymbol{r}$, respectively, $\boldsymbol{f}$ denotes the DFT index and $\boldsymbol{N}$ denotes the filter sample size. "Convolution is a mathematical way of combining two signals to form a third signal" [Smith 1997]. When this filtered sound is presented to a listener the original sound environment is recreated. The BRIR can be considered as the signature of the room response for a

particular sound source and human receiver. Although interlinked, for simplicity and reasons of practicality, the room response and the response of the human receiver are commonly determined separately and combined via a post-processing operation to provide an approximation to the actual BRIR [Kleiner et al. 1993]. The response of the room is known as the Room Impulse Response (RIR) and captures the reflection properties (reverberation), diffraction, refraction, sound attenuation and absorption properties of a particular room configuration (i.e., the environmental context of a listening room or the "room acoustics"). The response of the human receiver captures the direction dependent effects introduced by the listener due to the listener's physical make-up (i.e., pinnae, head, shoulders, neck, and torso) and is known as the Head Related Transfer Function (HRTF). HRTFs encompass various sound localization cues including ITD, ILDs, and the changes in the spectral shape of the sound reaching a listener [Kapralos, Jenkin, & Milios 2003]. The HRTF modifies the spectrum and timing of sound signals reaching each ear in a location-dependent manner. Various techniques are available for determining (measuring, calculating) both the HRTF and the RIR however, a detailed discussion of these techniques is beyond the scope of this thesis (see Kapralos et al. [2008] for greater details). The output of the techniques used to determine the HRTF and the RIR is typically a transfer function which forms the basis of a filter that can be used to modulate source sound material (e.g., anechoic or synthesized sound) via a convolution operation. When the filtered sounds are presented to the listener, in the case of HRTFs, they create the impression of a sound source located at the corresponding HRTF measurement position and when considering the RIR, recreate a particular acoustic environment. However, as

previously described, convolution is a computationally expensive operation especially when considering long filters associated with HRTFs and RIRs (filters with 512 coefficients are not uncommon) thus limiting their in real-time applications (the operation described in Equation 1.4 must be performed for each input signal sample).

1.1.6 Propagation Based Sound Localisation

When there are no occluding objects located between the sound source and listener, the listener may perceive the relative distance and direction of the sound source fairly accurately (sound source localization error ranges between 2.75° for sounds that are in front of the listener, and up to 20° for sounds that are above or below the median plane and behind the listener under ideal conditions (young listeners seated in an anechoic chamber) [Makous & Middlebrooks 1990]). The direction vector can be calculated simply by subtracting the listener's position from the position of the sound source. The direct distance between the sound source and listener is equal to the length of the direction vector. Imagine instead that the listener is located in a room with soundproof walls and that there is only one opening. Any sound outside of this room must reach the listener by passing through the opening. Regardless of the actual direction vector, the listener will perceive the sound to be coming from the direction of the opening. The sound will also be perceived as being farther away because the attenuation is based on the distance that the sound wave has travelled. The listener may perceive the sound as originating from a different location as shown in Figure 1.4.

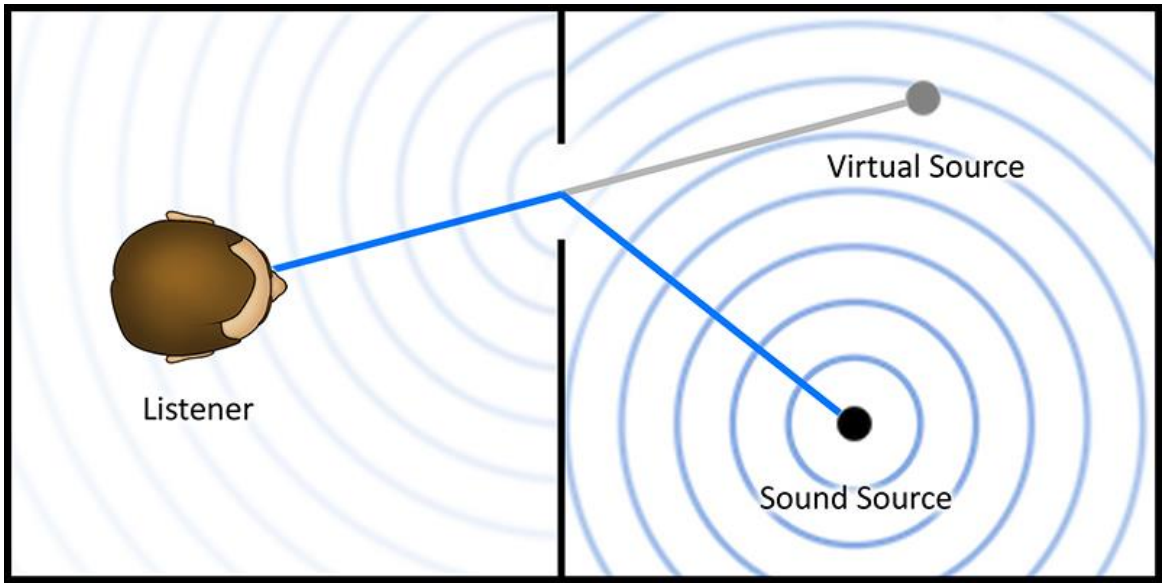


Figure 1.4: The sound source and listener are located in separate rooms connected by an opening. The listener will perceive the sound as coming from the direction of the opening. The sound will also be perceived as being farther away because attenuation is based on the distance that the sound wave has travelled.

By default, most sound engines base their ILD, ITD, and attenuation on the actual location of the sound source and ignore the localisation error caused by sound traveling around occluding structures. In order to incorporate propagation based sound localisation (PBSL) into current sound engines, the concept of virtual sound sources can be used. To do this, the path that sound travelled in order to reach the listener must be calculated. The distance of the virtual sound source is equal to the total length of the path taken (or a weighted average in the case of multiple paths), and the direction points to the last-leg of the journey (the location of the last change in direction before reaching the listener). By placing the sound at the location of the virtual sound source, the sound engine will be "tricked" into rendering the ILD, ITD, and attenuation properly. The problem with using virtual sound sources is that the path (or paths), taken by the sound wave must first be calculated; modelling the flow of sound energy through complex 3D structures is not a trivial

task. Some of the current methods for modeling sound propagation in real-time will be discussed in Chapter 2.

1.2 Motivation

Given the importance of sound, and our ability to localize sound sources in three-dimensions in the real world, virtual environments (including virtual simulations, serious games, and video games in general), where often times the goal is to replicate the real world, should include realistic spatial sound cues (including PBSL). “The goal of spatial sound rendering is to create a virtual auditory environment that is indistinguishable from a real auditory environment” [Li et al. 2006]. However, current real-time spatial sound systems are overly simplified in order to reduce their computational complexity, resulting in spatial sound cues that are unrealistic [Li et al. 2006]. The framework introduced here is capable of efficiently modelling the flow of sound energy in virtual environments with the goal of increasing the auditory realism of virtual environments at interactive rates.

1.2.1 Spatial Sound in Virtual Reality (VR) Applications

The recent resurgence of virtual reality and augmented reality brought about by new consumer level devices, has pushed developers to create more immersive virtual worlds with high fidelity graphics and sound [Serafin et al. 2018]. The Merriam-Webster (2018) dictionary defines virtual reality (VR) as “an artificial environment which is experienced through sensory stimuli (such as sights and sounds) provided by a computer and in which one's actions partially determine what happens in the environment; also: the technology used to create or access a

virtual reality." VR refers to a range of devices including those that provide haptic feedback (sense of touch) or tracking (measuring the position and orientation of the user's body). However, in this thesis, the term VR will refer to the visual and auditory experience provided by VR head mounted displays (headsets).

Virtual reality headsets (head mounted displays) such as the Oculus Rift (and more recently the Oculus Quest), PlayStation VR or HTC Vive, enhance the sense of presence and immersion by presenting the virtual environment with stereoscopic 3D visuals, and by tracking the orientation of the user's head [Candusso 2017]. With head tracking technology, the user's head becomes a very natural and intuitive input device [Qian & Teather 2017]. When playing video games on a TV or monitor without the use of headphones, it is understood by the player that the amplitude panning from left to right is relative to the orientation of the screen, and not the player's head. Wearing headphones can provide better sound fidelity and additional spatial sound cues to the player, however the sound localisation is generally relative to the view being rendered graphically and not relative to the orientation of the player's head. While wearing a VR headset, the player's physical orientation dictates the orientation of the virtual camera, and by extension the rendered scene. With the virtual world aligned to the user's head, sound localisation in the form of HRTF filtering and a simulated RIR generated based on the virtual environment becomes more intuitive. If the player hears a sound coming from their right, they can simply turn their head to discover the source of the sound. While VR headsets encourage users to make use of spatial sound cues, they also expose any inaccuracies in the rendering of spatial sound. For example, when

sound localisation is not based on the path (or paths) taken by the sound in order to reach the listener, the user may perceive sounds to be passing through solid structures such as walls. Calculating the flow of sound waves through complex environments built from potentially millions of individual surfaces (polygons) requires far too much processing time. In the past, it was common to simply ignore occluding objects such as walls, instead, the direction of the sound source was used. This implied that sound could pass through walls and other solid objects unhindered, which can be disorientating for the player given that sound does not behave this way in the real world. Unfortunately, this technique is still in use for modern video games that have dynamic environments. For example, acoustical occlusion is completely ignored in the popular first-person shooter Fallout 4, by Bethesda Game Studios. The idle conversations of the enemy units can be heard clearly through walls, and yet, the enemies are unable to hear gunfire and explosions through those same walls. Fallout 4 environments contain doors that can be opened and obstacles that can be destroyed which makes the precomputation of occlusion impossible. When the environment is static or unchanging during the course of the game, crude path information can be pre-calculated which allows the sound to be rendered with PBSL or virtual sound sources that simulate real-world sound propagation models.

1.2.2 The Difficulty of Rendering Spatial Sound

The generation of realistic spatial sound requires the propagation of sound through the environment to be traced from every sound source to every listener. High performance is required in virtual environments, and video games in particular

where, typically only a fraction (approximately 10%) of the total CPU resources is devoted to sound rendering [Wilde 2004]. Thus, interactive sound rendering techniques that stay within the allotted CPU budget need to be developed. Secondly, video game environments range from small rooms to large cities constructed from potentially millions of triangular surfaces. Such complex models are challenging to handle efficiently for current sound rendering techniques [Funkhouser et al. 2002]. Thirdly, the sound sources, listeners, and geometric objects in video games can move (i.e., they are dynamic). Thus, sound rendering techniques need to efficiently handle complex, dynamic, and general scenes further taxing limited computational resources. That being said, it may be acceptable to trade accuracy for higher computational performance in video games and simulations.

1.2.3 Precomputation

Auditory simulations that do not model sound propagation result in noticeably unrealistic sound cues that decrease the player's feeling of "presence" and immersion [Torres et al. 2001; Tsingos et al. 2001]. That being said, sound propagation is a difficult and computationally intensive task and is therefore, typically ignored. Real-time acoustical modelling algorithms do exist but are rarely utilised in video games or virtual environments due to the computational requirements or restrictions placed on the type of environment or the number of sounds.

Precomputation, whereby an acoustical algorithm calculates the many paths that sounds could travel from every room in the environment to every other room, can be used to improve performance. The intensive processing is performed offline before the game is shipped and stored as a file that can be loaded along with the game's environment. The resulting spatial sound representation can be fairly accurate, although, the acoustics of the environment cannot change while the game is played. Large moving objects such as vehicles will have no effect on the sound propagation model. For example, in a video game which seeks to simulate the flow of traffic in an urban setting, large busses and trucks won't block sound if precomputation was used. A large bus would not occlude sound, sound would not diffract around the bus causing the localisation of the sound source to shift, and the presence of the bus would not cause the near-by environment to become more reverberant.

1.2.4 Precomputation Is Not Always Possible

Many popular video games such as Minecraft or Fortnite, allow the player to alter or even create their own environments by placing components from a library of objects and structures. The dynamic nature of these environments makes precomputation impossible. Not all video game environments contain geometry that can serve as the basis for acoustical modelling. For example, 2D video games present a cross section of the environment which allows the player to see what is happening around their avatar. Enclosed spaces are rendered with an outer wall removed allowing the camera to peer inside so as not to hide the player's avatar from view. This one-sided environment will not provide enough information for

acoustical modelling even if the visible areas were constructed from polygonal geometry. In addition, not all game environment employ 3D geometry. It is still common for games to use beautifully drawn or hand-painted scenes. A hand painted depiction of a 3D scene may still benefit from the inclusion of spatial sound. For example, the player might expect sounds to echo if the environment consists of a hand painted 3D cave. However, a hand painted scene does not contain any geometry needed to calculate echoes.

Modern geometrical techniques (which will be described in greater detail in Chapter 2), such as image-sources, ray tracing or beam tracing, approximate PBSL by modelling sound propagation starting at the listener's position [Funkhouser et al. 1998, 1999; Savioja et al. 2015]. It is common for these techniques to conserve processing time by only propagating sound through the environment close to the listener which may not be sufficient for complex indoor environments. In addition, current real-time acoustical modeling methods often greatly simplify the environment in order to reduce the number of interactions that sound has with the environment. For example, a room filled with furniture may be too complex to model accurately in real-time and therefore, simplified versions of the room are used instead. Distance fields can also be used in place of polygonal geometry. A distance field is a 2D or 3D array of values representing the distance from the nearest collidable surface. The environment can then be ray-marched efficiently avoiding the need for ray vs. polygon collision checking.

Regardless of how the environment is represented, any solution will inevitably require that a large amount of data is processed in real-time (quickly enough that the user does not perceive any lag between the graphical rendering and the sound rendering). More specifically, a large scale complex environment may contain many simultaneously audible sound sources, and each sound must have a virtual position due to PBSL, an occlusion / diffraction component, and reverberation applied to it based on the geometry located between the sound source and listener.

1.3 Graph-based Sound Propagation

In order to model the flow of sound through an environment efficiently, it may be necessary to ignore interactions with each individual surface and focus on the aggregate movement of sound. It may be possible to remove the geometry completely from the computational process by converting the geometrical description of the environment to a structured graph whereby nodes are placed in open areas (areas that allow sound to pass through) throughout the environment. Nodes are connected to their neighbours only if sound is able to pass between them. An occlusion value representing the amount of sound energy that is able to pass between the nodes is stored for each connection. In addition, each node could store information about its surroundings such as the average reflectivity and the size of the space (small and enclosed vs. large and open). Graphs have the advantage of being efficient to parse and can potentially be processed in parallel to take advantage of the GPU architecture.

1.4 Problem Description

Despite the benefits of spatial sound, the faithful simulation of sound cues is often overlooked in immersive virtual environments/games, and virtual reality research. Many virtual environments including video games today still employ simple sound propagation models completely ignoring the presence of occluding geometry. [Murphy & Neff 2011]. Many modern sound cards now include HRTF filtering which improves the player's ability to localise sound. However, without PBSL, sound is still able to pass through solid objects such as walls unobstructed. HRTF filtering does not solve this problem, in fact it draws attention to it. In addition to being more realistic, spatial sound with PBSL provides additional information to the player, leading them to the source of the sound rather than leading them into dead ends.

The generation of spatial sound for dynamic, and interactive virtual environments using traditional software-based methods and techniques is computationally very expensive except for trivial environments that are typically of limited use. The diffraction, and reflection of sound can be approximated by applying artist created filters [Kastbauer 2010]. These filters are generally created manually by artists instead of employing a sound propagation model. For example, to model echoes in a cathedral, artists may use a filter which sounds like a cathedral, but only a single filter is created for the whole space and it does not vary with the relative positions of the sound source and the listener [Chandak 2011].

Although the situation has recently started to improve [Kuzminski 2016], historically, the emphasis has been placed on the visual senses instead [Nordahl, Turchet, & Serafin 2011]. However, with each new graphical improvement, the user will expect additional realism from other aspects of the virtual environment (e.g., video game) as well, including audio [Murphy & Neff 2011].

1.5 Thesis Contribution

A spatial sound framework was developed for video games and virtual environments utilising the inherent parallelism of modern GPU architectures. The main features of the framework are listed below:

- **Graph-Based Propagation Modeling**: Rather than modeling the many interactions of each sound with each surface, a graph is used to model the aggregate flow of sound energy. The sound propagation path or paths are traced through the environment connecting every sound source to the listener in real-time.
- **GPU Powered**: Most of the processing is done by the GPU allowing large-scale graphs to be processed efficiently with no limit placed on the number of active sound sources.
- **GPU Audio Convolution**: Effects can be efficiently applied to a sound in real-time or while the game is loading to create custom DSP effects or add variation [Cowan & Kapralos 2008, 2009a, 2013b].
- **Dynamic Occlusion Sampling**: The amount of acoustical occlusion between any two points in 3D space can be efficiently measured. This novel

approach involves rendering the geometry with a shaders that estimate the contribution of each surface between the sound source and listener [Cowan & Kapralos 2009b, 2010, 2011a, 2011b, 2013a, 2015].

- **Dynamic Reverberation sampling**: The geometry around a sound source is measured by rendering a six sided cube with a shader designed to sample the reflective properties of each surface. The resulting image is parsed to create the parameters used to drive a real-time reverberation DSP effect [Cowan & Kapralos 2011c].
- **Supports Every Type of Environment**: A graph can be used to model a variety of environments; 2D, 3D, or voxelised. Even non-geometric scenes could be supported by permitting the designer to manually construct the graph by plotting points and connections.
- **Parametric Filtering**: Reverberation, and occlusion are simulated using highly efficient DSP effects that do not require specialised hardware.
- **Compatibility**: This framework is built on top of the Fmod sound engine which supports all platforms (Windows, Mac, Linux, Android, iOS, and all major game consoles). In addition, the framework could easily be adapted to work with other sound engines because it does not rely on any features that are unique to Fmod.
- **Ambiguity**: Sound localisation cues provided to the listener (ITD, ILD, and HRTF filtering) would not be as strong when the sound source is occluded when compared to the direct sound. Modelling ambiguity improves the realism of the simulation when the sound source is occluded and sound is able to reach the listener from different directions.

- **AI**: The framework can also be used to provide a sense of hearing for NPCs which can allow them to behave in a more realistic manor by restricting their knowledge of the world to that which they could reasonably perceive with their senses.

In addition, this thesis provides a comparison of current methods used to simulate sound propagation for virtual environments with an emphasis on video games. A user study designed to test the effectiveness of the spatial sound framework, as well as its effect on player performance was conducted. The results of this study indicate that the inclusion of spatial sound improves player performance for both novice and experienced players.

1.6 Thesis Structure

The remainder of the thesis is organized as follows. In Chapter 2, a background literature review is provided. An overview of existing real-time spatial sound algorithms specifically is provided (both GPU- and non-GPU-based methods are described). Chapters 3, 4, and 5 each detail the creation of one component of the overall spatial sound framework. Chapter 3 introduces the concept of GPU audio processing with the implementation of a convolution filtering method. Chapter 4 describes a computationally efficient GPU-based method / technique to simulate acoustical occlusion and diffraction between two points in space by rendering the geometry graphically. Chapter 5 adds a simulated room impulse response (RIR) created by measuring the geometry around the sound source using a graphical rendering technique and then applying digital signal processing (DSP) effects to

the sound in real-time. Chapters 3, 4, and 5 are based on published journal articles. These previous works lead to the completed spatial sound framework presented in Chapter 6. This framework was designed to provide spatial sound for complex virtual environments with many concurrent sound sources in a computationally efficient manner. In other words, a trade-off between accuracy and efficiency is made. Although the method has been developed primarily for videogames, it is relevant for dynamic and interactive virtual environments where real-time operation is required and an accuracy vs. efficiency trade-off can typically be made. Chapter 7 includes a user study conducted to determine whether or not realistic spatial sound cues improve performance in virtual environments and games. Concluding remarks and plans for future research are provided in Chapter 8.

Chapter 2 Background

This chapter explores some of the current spatial sound technologies, as well as some of the advancements that led to their development. Specifically, this chapter begins with an overview of the graphics processing unit (GPU) given the use of the GPU in many spatial sound methods including the spatial sound framework presented here. The chapter will then focus on interactive algorithms that may be relevant to real-time applications such as video games and VR. This section does not attempt to provide an exhaustive analysis of past or current research, or readily available software solutions. Instead, the technologies highlighted here are intended to provide the reader with a context in which to frame the spatial sound framework introduced in Chapter six. A more complete overview of acoustic modeling techniques is available from Lakka et al. [2018], Savioja & Svensson [2015]; and Välimäki et al. [2016].

2.1 GPU Processing

Given the potential computational efficiencies that may be obtained with modern graphics processing units (GPUs), several initiatives have begun to employ the GPU to generate spatial (3D) sound in real-time for inclusion in videogames and virtual environments. In Chapter 2, various methods utilising the GPU for sound generation, digital signal processing, and spatial sound generation through the simulation of acoustical occlusion, diffraction, and reverberation are described. However, prior to doing so, a brief overview of the GPU is provided.

2.1.1 Graphics Processing Units: Overview

In computer graphics, rendering is accomplished using a graphics pipeline architecture whereby rendering of objects to the display is accomplished in stages and each stage is implemented as a separate piece of hardware. The input to the pipeline is a list of vertices expressed in object space while the output is an image in the framebuffer. The stages of the pipeline and their operation are as follows [Owens et al. 2007]:

- Vertex Stage: (1) Transformation of each (object space) vertex into screen space, (2) formation of triangles from the vertices, and (3) per-vertex lighting calculations.
- Rasterization Stage: (1) Determination of the screen position covered by each of the triangles formed in the previous stage and (2) interpolation of vertex parameters across the triangle.
- Fragment Stage: Calculation of the color for each fragment output in the previous stage. Often, the color values come from textures, which are stored in texture memory. Here the appropriate texture address is generated and the corresponding value is fetched and used to compute the fragment color.
- Composition Stage: Pixel values are determined from the fragments.

2.1.2 Shading Languages

In contrast to the "traditional" fixed-function pipelines, both the vertex and fragment stages are user-programmable on "modern" GPUs. Programs written to control the vertex stage are known as vertex programs or vertex shaders, while

programs written to control the fragment stage are known as fragment programs or fragment shaders. Early on, these programs were written in assembly language. However, higher level languages have since been introduced including Microsoft's high-level shading language (HLSL), the OpenGL shading language (GLSL) [Rost 2006], and NVIDIA's Cg [Mark et al. 2003].

The vertex shader is a program primarily responsible for calculating the position of each vertex in screen-space. Each vertex is rotated, scaled, and projected onto the 2D screen based on the position and orientation of the camera. All of the data processed by the vertex shader is passed on to the fragment or pixel shader. Each vertex forming a triangle, may also return additional information such as the surface normal (direction that the surface is facing at that point) for example. The output from the vertex shader is linearly interpolated across the surface of the triangle to generate the input for the fragment shader. The fragment shader processes individual fragments which become pixels in the rendered image if they pass the depth test (fragments hidden behind other fragments are discarded). The output from the fragment shader is usually colour and depth information.

The fragment stage is capable of fetching data from texture memory (memory gather) but cannot alter the address of its output, which is determined before processing of the fragment begins (incapable of memory scatter). In contrast, within the vertex stage, the position of input vertices can be altered thus affecting where the image pixels will be drawn (i.e., the vertex stage supports both memory gather and memory scatter) [Owens et al. 2007]. In addition to vertex and fragment

shaders, Shader Model 4.0 currently supported by Direct3D 10 and OpenGL 3.0 defines a new type of shader, the geometry shader. A geometry shader receives input from the vertex shader and can be used to create new geometry. It is also capable of operating on entire primitives [Sherrod 2008].

2.1.3 GPU Architecture

Driven by the video game industry, consumer computer graphics hardware has greatly advanced in recent years, outperforming the computational capacity of central processing units (CPUs). A graphics processing unit (GPU) is a dedicated graphics rendering device whose purpose is to provide a high performance, visually rich, interactive 3D experience by exploiting the inherent parallelism in the feed-forward graphics pipeline [Luebke & Humphreys 2007]. In contrast to consumer-grade audio cards, GPUs have moved away from the traditional fixed-function 3D pipeline toward a flexible general-purpose computational engine that can currently implement many parallel algorithms directly resulting in tremendous computational savings. Due to a number of reasons including the explosion of the consumer videogame market and advances in manufacturing technology, GPUs are, on a dollar-per-dollar basis, the most powerful computational hardware, providing "tremendous memory bandwidth and computational horsepower" [Owens et al. 2007]. In fact, instead of doubling every 18 months as with CPUs, GPU performance increases by a factor of five every 18 months or doubles every 8 months, far exceeding Moore's Law applied to traditional microprocessors [Ekman et al. 1994; Geer 2005]. Current GPUs are fully programmable and support vectorized floating point operations [Owens et al. 2007].

Architecturally, the CPU is composed of a few processing cores with a lot of cache memory. For example, the Intel CORE i9-7980XE Extreme Edition CPU includes 18 cores and 24.75 MB of cash memory [Intel 2018]. In contrast, a GPU is composed of hundreds or even thousands of cores. Nvidia's latest gaming GPU, the RX 5700 XT contains 2,560 cores [Walton 2019]. However, in order to take full advantage of modern GPUs, programmers must design programs with a high degree of parallelism. This means that the same code should run concurrently on many threads while avoiding the need for communication between threads.

2.1.4 General Purpose Computing on the GPU

Furthermore, a number of high level languages have been introduced to allow for the control of vertex and pixel pipelines [Buck et al. 2004]. In order to take advantage of the power inherent in GPUs in addition to their relatively low cost, recently a number of efforts have investigated the use of GPUs to a variety of non-computer graphics applications. Collectively, this effort is known as "general purpose computing on the GPU" (GPGPU) [Owens et al. 2007]. A thorough review including a detailed summary of GPGPU-based applications is beyond the scope of this thesis but is given by Hu, Che, & Zheng [2016]; and Owens et al. [2007] and will therefore not be provided here.

CUDA (Compute Unified Device Architecture) which was released in 2007 by Nvidia, is the extension of C programming Language designed to support of parallel processing on the GPU [Nvidia 2007]. CUDA programs are referred to as

"compute shaders" and unlike the other types of shaders discussed previously, they are not attached to any stage in the rendering pipeline [Ni 2009]. Instead, they allow the GPU to act as a general purpose processor [Nickolls, Buck, & Garland 2008]. Today there are several alternative compute shader languages available including OpenCL, OpenGL, and HLSL. All of the compute shader languages have similar capabilities and most have a syntax similar to C which makes them quick to learn for programmers already experienced at writing C/C++ code.

2.1.5 GPU Audio Processing

Today's video games, employ advanced graphics rendering techniques including: real-time radiosity, indirect illumination, volumetric shadow maps, and other advanced lighting and shading techniques [Laukkanen 2018, Tatarchuk 2009]. Such techniques are implemented using the graphical processing unit (GPU). There is an increasing demand for similar advanced techniques for sound rendering to enhance the immersive experience in video games [Haraldsen 2010]. Taking advantage of the power inherent in graphics processing hardware, a number of efforts have applied the GPU to spatial sound generation, leading to large computational savings and various interactive rate (feedback is perceived by the user to be continuous and immediate) spatial sound methods and techniques (see [Gallos & Tsingos 2003, Röber, Kaminski, & Masuch 2007, Cowan & Kapralos 2009b, 2010, 2011a, 2011b, 2013a]). A thorough overview of GPU-based spatial sound generation will not be provided here but is available from Savioja et al. [2015]; and Hamidi & Kapralos [2009].

2.2 Sound Generation and Signal Processing

GPUs have also been applied to a wide variety of sound-based applications. von Tycowicz & Loviscach [2008] describe the implementation of a flexible virtual drum that is simulated in real-time and with low latency on the GPU. A Musical Instrument Digital Interface (MIDI) controller with 16 pressure points is used for pressure recognition and a finite difference method is employed to synthesize sound based on location and pressure information. Matsuyama et al. [2007], describe a method for the automatic generation of real-time sound for graphics-based animation of sparks to simulate thunder and lighting effects. The implementation also makes use of GPU-based numerical methods introduced by Krüger & Westermann [2003]. Using the Cg shading language, Whalen [2005] implements seven common audio functions: chorus, compress, delay, high-pass filter, low-pass filter, noise-gate and normalization. A performance comparison was made between the GPU and corresponding CPU implementation using a Pentium IV (3.0 GHz CPU) and an NVIDIA GeForce FX 5200 video card. The GPU showed better performance for several of the functions (compress and chorus with speedups of up to a factor of four). However, the CPU implementation was better for other functions (high-pass and low-pass filters). It was suggested that the GPU performance was poorer for some algorithms because they required (computationally expensive) texture access. With more modern video cards, texture access has been improved and this will undoubtedly lead to greater improvements in these methods.

2.3 Acoustical Modelling

The Cambridge Dictionary defines the word “acoustic” as “relating to sound or hearing”. The same dictionary defines a model as “a simple description that can be used in calculations” [Cambridge English Dictionary 2019]. The algorithms discussed here provide a sense of hearing for real-time applications by simplifying the environment, by approximating the physics of sound using a simplified model, or in most cases, a combination of the two.

2.3.1 Acoustical Occlusion and Diffraction Modelling

The image source method models reflections by replacing reflecting surfaces with virtual sound sources, which are a mirrored reflection of the original source. This method assumes that sound waves propagate along straight lines from the original source and can therefore be modelled as rays. Each reflection creates a new virtual sound source with the volume attenuated based on the surface material properties and the distance travelled by the ray [Wang & Pan 2018].

Tsingos & Gascuel developed an occlusion and diffraction method that utilizes the GPU to perform fast sound visibility calculations that can account for specular reflections, absorption, and diffraction caused by partial occluders [Tsingos & Gascuel 1997]. Specular reflections were handled using the image source approach [Allen & Berkley 1979], while diffraction is approximated by computing the fraction of sound that is blocked by obstacles on the path between the sound source and the receiver (listener) by considering the amount of volume of the first Fresnel ellipsoid that is blocked by the occluders. A visibility factor is computed using computer

graphics hardware. A rendering of all occluders from the receiver's position is performed and a count of all pixels not in the background is taken (pixels that are "set" i.e., not in the background, correspond to occluders). Their approach handles a discrete set of frequency bands ranging from 31 Hz to 8 kHz and is primarily focused on sounds for animations. Although experimental results are not extensive, their approach is capable of computing a frequency-dependent visibility factor that takes advantage of graphics hardware to perform this in an efficient manner. Their approach is not completely real time, but it is "capable of achieving interactive computation rates for fully dynamic complex environments" [Tsingos & Gascuel 1997].

Tsingos and Gascuel later introduced another occlusion and diffraction method based on the Fresnel-Kirchhoff optics-based approximation to diffraction [Tsingos & Gascuel 1998]. The Fresnel-Kirchhoff approximation is based on Huygens' Principle [Hecht 2002]. The total unoccluded sound pressure level at some point $\boldsymbol{p}$ in space is determined by calculating the sound pressure of a small differential area $\boldsymbol{dS}$ and integrating over the closed surface enclosing $\boldsymbol{p}$ (see Tsingos & Gascuel [1998] for further details regarding this calculation in addition to an algorithm outlining the method [Tsingos & Gascuel 1998]. After determining the total unoccluded sound pressure arriving at point $\boldsymbol{p}$ from a sound source, diffraction and occlusion effects are accounted for by computing an occlusion depth-map of the environment between the sound source and the receiver (listener) using computer graphics hardware to permit real-time operation. Once the depth-map has been computed, the depth of any occluders between the sound source and the receiver

can be obtained from the Z-buffer [Foley et al. 1994] whereby "lit" pixels correspond to occluded areas. The diffraction integral described by the Fresnel-Kirchhoff approximation is then approximated as a discrete sum of differential terms for every occluded pixel in the Z-buffer. Comparisons for several configurations with obstacles of infinite extent between their method and between boundary element methods (BEMs) give "satisfactory quantitative results" [Tsingos & Gascuel 1998]. Tsingos et al. [2007] introduced a high quality, GPU-based acoustical first-order sound scattering modeling method that is based on a surface integral formulation and Kirchhoff's approximation. Their method is capable of modeling both diffraction and reflection in an arbitrarily complex environment. Their results indicate that the method fares well with boundary element methods (BEMs) although greater work remains to allow for higher-order sound scattering and to overcome the fact that the method is prone to aliasing.

Biot-Tolstoy-Medwin Diffraction

Although not GPU-based, Antani et al. [2010] developed a "fast algorithm" to calculate sound propagation in complex 3D environments. The method accounts for specular reflections and higher-order edge diffractions about finite edges in the scene. Diffraction modeling is based on the Biot-Tolstoy-Medwin diffraction model and they employ visibility culling to significantly reduce the number of edge pairs that need to be processed. Various other research efforts have examined non-geometric acoustics-based diffraction modeling. Torres et al. [2001] introduced a time-domain model based on the Biot-Tolstoy-Medwin technique [Biot & Tolstoy 1957], which computes edge diffraction components and combinations of specular

and diffracted components. Lokki et al. [2002], Calamia et al. [2005], and Svensson et al. [1999] have also investigated diffraction modeling based on the Biot-Tolstoy- Medwin technique. The method of Calamia et al. [2005] provides an integrated approach to acoustical modeling whereby intermediate values typically used in diffraction calculations using the Biot-Tolstoy-Medwin technique are exploited to find specular reflections as well. In addition to these techniques, a number of wave-based acoustical modeling methods, whose objective is to solve the wave equation in order to recreate the room impulse response that models a particular sound field, inherently account for occlusion/diffraction effects (see Murphy & Beeson [2003] for example). Such techniques are currently not generally applicable to real-time, interactive applications due to complexity issues and are therefore not considered further here.

2.3.2 Beam Tracing

The beam tracing acoustical modeling approach of Funkhouser et al. accounts for diffraction using a method based on the uniform theory of diffraction (UTD) [Funkhouser et al. 2004; Keller 1962; Tsingos et al. 2001]. Validation of their approach by Tsingos et al. involved a comparison between actual measured impulse responses in a simple enclosure (the "Bell Labs Box") and the impulse responses obtained by simulating the enclosure [Tsingos et al. 2001].

Tsingos et al. [2007] developed a new approach for high-quality modeling of first-order sound propagation that maps well to graphical hardware since it computes the scattering from detailed, dynamic geometry. It combines the Helmot-Kirchhoff

theorem with the Kirchhoff approximation to derive an expression for first order scattering effects. A GPU implementation in this case is suitable because the above formulation is similar to the reflective shadow map that is introduced to compute interactive first order global illumination effects. It allows the implementation of a level-of-detail approach that reduces the geometry processing for audio rendering while preserving the scattering behavior of complex surfaces by allowing bump or displacement mapping. There are two steps in the computation. During the first step all scattering surfaces visible from the source are determined and sampled. The second step involves the evaluation and summation of the differential contribution of clocked plus reflected wavefronts for all surface samples. The first task is implemented using a computer graphics shadow mapping source-view technique that renders the scene from the location of the sound source. For the second task, a hierarchical integration method known as “mip-mapping” is used that requires $\boldsymbol{log(rez)=log(k)}$ render passes (where $\boldsymbol{rez}$ is the rendering resolution and $\boldsymbol{k}$ is the reduction factor). At each pass a $\boldsymbol{k} \times \boldsymbol{k}$ block of values is summed to give a single value which is recursively integrated in the next pass until the value of the integral is reached. Both visual rendering and the calculation of sound scattering coefficients are done on the GPU. The auralization is achieved by re-equalizing that is performed asynchronously on the CPU. The method was tested on a Pentium IV 3.4 GHz CPU and an NVIDIA GeForce 8800 GTX graphics card and was compared to a C++ implementation on the CPU. For an interactive scenario the GPU-based method was found to be 40 times faster. The limitations of this method are that it is prone to aliasing due to the insufficient sampling at high frequencies. Furthermore, the method is limited to first-order scattering and

therefore cannot be used for some audio effects such as reverberation and occlusion. Also, by using a frequency domain approach (essential for an efficient implementation), this method introduces an approximation because of the limited number of frequency bands considered.

2.3.3 Voxel Based Propagation

Röber et al. [2006] present a (low-frequency) wave-based acoustical modeling method that makes use of the GPU and in particular, fragment shaders, 3D textures, and the OpenGL frame-buffer objects extension, in order to take advantage of the inherent parallelism of wave-based solutions to acoustical modeling. The one-dimensional mesh is extended by constructing a digital mesh from bi-linear delay lines that connect junctions that act as temporal and spatial sampling points. The programmable vertex shader is used to implement computations on a per vertex basis on a 3D space and the fragment shader is used to compute the final pixel color. Waveguide node data is stored in three buffers that are combined into one RGB texture with the data stored in the red and blue components and the geometry and boundary coefficients in the green channel. During each time frame, the fragment shader computes the difference equations for each node in the mesh and stores the result in the buffer. They have used a body centered cubic grid (BCC) which is a hexagonal lattice that requires only 70% of the sampling points compared to the usual rectilinear grid which is a cubic Cartesian lattice. This data structure reduces the computation load by $\sqrt{2}$ and can be decomposed into cubic grids that makes the GPU implementation straightforward. The limitations of this approach are a direction dependent dispersion error and the

finite mesh resolution to model boundary behavior. Also, the approach implements 2D meshes only. The system was tested on a PC with an AMD 64 4000+ dual-core CPU and an NVIDIA GeForce 7900 GT video card and showed speed-ups of factors of from 4.5 to 69 when compared to a software-based implementation. However, the CPU implementation was not optimized.

2.3.4 Ray Tracing

Audio-based ray tracing using the GPU was implemented by Jedrzejewski [2004] to compute the propagation of acoustical reflections in highly occluded environments and to allow for the sound source and the listener to move throughout the simulation without the need for a long pre-computation phase. The method consists of six phases, the first four of which constitute a short pre-computation stage that takes advantage of the fact that in acoustics, as opposed to graphics, objects other than walls do not significantly modify sound waves and therefore can be ignored during the computation. Due to this, only polygons that represent walls are taken into account. Furthermore, to make the system more efficient, each ray is intersected with a plane rather than a polygon. A comparison of the method implemented on a GPU and a CPU (2 GHz AMD CPU and an ATI Radeon 9800 video card) demonstrated that the GPU-based implementation was much more computationally efficient (32 vs. 500 ms to trace a ray of order 10 on the GPU and CPU respectively). Röber et al. [2007] describe a ray-based acoustical modeling method that employed the GPU to allow for real-time acoustical simulations. Their framework was designed along existing (computer graphics) GPU-based ray tracing systems suitably modified to handle sound wave

propagation. The system accepts a 3D polygonal mesh of up to 15,000 polygons and pre-processes it into an accessible structure. All signal processing is programmed as fragment shaders and for each task a single shader was developed. The 3D scene data is loaded into texture memory and decomposed into 10 bands that are assigned positions and an emittance pattern within the virtual room. Rays are cast into the scene and the value of acoustical energy received per frequency band is accumulated and stored within cube-maps. A frame-rate of up to 25 fps was achieved using a detailed model of a living room containing 1,500 polygons (using an NVIDIA GeForce 8800 GTX video card).

VR Works Audio

NVIDIA Corporation, which was founded in 1993, has become a leader in the design of cutting edge GPUs [Reference for Business 2019]. One of the ways Nvidia promotes its hardware is by writing software designed to take advantage of their latest hardware innovations. In addition to their graphics based APIs, they have released a series of free tools for developers that utilise their GPUs as general purpose processors [Nvidia 2019b]. In addition to graphics, AI, and physics, Nvidia has created an advanced spatial sound rendering system which was added to their VRWorks API in 2016 [James 2016].

VRWorks Audio makes use of Nvidia's pre-existing ray tracing engine OptiX in order to simulate sound propagation within a virtual environment based on the same polygonal models used to render the scene graphically [James 2016]. Each of the models may be assigned a set of material properties which include the surface

reflectivity and acoustical occlusion. Multiple paths are traced through the environment modelling reflection and occlusion at each collision with a surface. Attenuation is accounted for based on the length of each traced path and diffraction is estimated based on the shortest path only. Two filters are generated (right and left ear) by summing the contribution of each path and applying a HRTF filter. For each pair of source and listener, the library builds a pair of high-resolution filters. Each filter may be up to two seconds in length with a frequency as high as 48000 Hz. [Nvidia 2019b]. It is unclear how the correct HRTF filters are selected based on the documentation provided by Nvidia. However, given the computational cost associated with HRTF convolution, it is likely that each of the traced paths are not filtered separately. The HRTF filter is likely based on the angle of the shortest path only. Finally, the completed filters are applied to the waveform data using convolution.

Key Features:

- Sound propagation, direct and indirect paths
- Occlusion for direct and indirect paths
- Directionality/HRTF (Head Related Transfer Function)
- Attenuation (based path length)
- Approximate direct path diffraction
- Material reflection, absorption, & transmission

Limitations

VRWorks Audio was designed to work with Nvidia GPUs exclusively. In addition, a GeForce GTX 900 series and higher, or a Quadro M5000 and higher GPU is required to use the API. The Unreal game engine (Unreal Engine 4) as well as the Unity game engine have made VRWorks Audio available to their customers by way of an optional plugin. Despite offering a state-of-the-art spatial sound system for free, it has not seen wide spread adoption by the games industry or by game engine developers. The slow adoption rate can partially be explained by the hardware requirements. The processing time scales linearly with the number of sound sources. However, it may be possible to group two or more sound sources in close proximity allowing them to share a set of filters.

2.3.5 Sonel Mapping

Sonel mapping is a probabilistic-based acoustical modeling method that is based on photon mapping, a popular image synthesis method [Jensen 2001]. Sonel mapping accounts for occlusion/diffraction effects using a modified version of the Huygens-Fresnel principle [Kapralos et al. 2008]. Sonel mapping is similar to ray tracing but has the added advantage of accurate edge diffraction. Each sonel (a packet of information propagating from the sound source to the receiver) can change direction mid-flight in response to the presence of nearby edges. Quantitative results indicate that sonel mapping conforms to theoretical acoustical diffraction models and being probabilistic based, it allows for an accuracy versus efficiency trade-off to be made. However, despite the computational

improvements, sonel mapping is not applicable for real-time applications except when considering very simple environments.

2.3.6 Graph Based Sound Propagation

A graph is a structure created from a collection of nodes (or vertexes) which are connected to neighboring nodes by connections, also known as edges [Deo 2017]. Graphs are typically used to represent a complex system in a format that computers can store and process efficiently. A road map can be stored as a graph whereby the nodes represent destinations and intersections. The roads in this example would be represented by the connections. A computer program could use this graph to calculate the shortest route between any two locations. Graphs are generally an abstraction of a more complex system. They only store information relevant to the results that they are expected to calculate. Information can be stored in each node, or as a part of each connection. For example, a graph used to model sound propagation might store reverberation parameters at each node, and the amount of occlusion at each connection. Graphs used to represent a coherent structure such as an object's surface or a floor plan may be referred to as a "mesh".

Ubisoft

Jean-François Guay from Ubisoft Montreal gave a presentation at the Game Developers Conference (GDC) in which he described a proprietary spatial sound rendering system developed by Ubisoft to enhance the sound quality of their games [Guay 2012]. The system is used internally by Ubisoft and has not been made available to outside developers. Based on the presentation titled "Real-time Sound

propagation in video games", it is clear that Ubisoft is not waiting for engine developers to add realistic spatial sound, they have instead built their own spatial sound system that works with existing game engines used by Ubisoft.

Instead of ray tracing to generate the shortest path connecting each sound source and listener pair, the developers at Ubisoft used navigation meshes (nav-meshes) which were likely generated by the underlying game engine. A nav-mesh is a 2D plus height mesh, meaning that mesh (polygonal model) does not represent a 3D volume of space. Instead, the mesh represents the 2D areas (floor space) that may be traversed by in-game characters. The nav-mesh has a height representing changes in elevation caused by uneven surfaces, stairs, or ramps. The mesh may also have overlapping layers to represent different floors of a building. The vertexes and edges that make up the nav-mesh can be used by pathfinding algorithms such as A* to find the shortest path connecting the sound source and listener. Based on the path taken by the sound, acoustical diffraction can be approximated at each turn and occlusion can be estimated by applying attenuation based on the path length [Guay 2012]. Presumably, the edges of the nav-mesh could provide additional occlusion information, however, details were not provided. PBSL is computed based on the last-leg of the journey, or the direction from the listener to the point where the path first becomes occluded. A virtual sound source is placed in the direction of the virtual sound source with a distance equal to the total path length. It is not clear how reverberation is handled by the system. It is likely that the reverberation parameters are simply pre-calculated at many discrete points

(reverb nodes) and their properties are interpolated for sound sources originating between them.

The developers at Ubisoft looked at ray tracing and beam tracing but determined that those methods required too much processing time. Rather than generating a RIR filter and convolving it with the dry (unaltered) sound, Ubisoft settled on a parametric method whereby the environment as well as the path between the listener and sound source is sampled, and the result of that sampling is a set of parameters that can be applied to the dry sound using traditional DSP effects built into existing sound engines. DSP effects are generally more computationally efficient than convolution. In addition, many modern sound cards have built-in DSP effects such as reverberation which further reduces the computation time of parameter based spatial sound. Ubisoft's graph based spatial sound system is currently used in many of their first person shooter (FPS) games such as Rainbow Six Siege [Ubisoft 2019].

G-SpAR

Beig recently introduced the GPU-Spatial Audio Renderer ("G-SpAR"), for the Unity game engine [Beig 2019]. G-SpAR has two very different methods; an efficient 2.5D CPU implementation based on nav-meshes, and a 3D GPU implementation using voxels.

The CPU version works very similar to the Ubisoft method discussed above whereby a 2.5D nav-mesh is used to find the shortest path connecting the listener

with each sound source [Guay 2012]. The nav-mesh and pathfinding are implemented by Unity game engine. The path information computed by Unity is not guaranteed to return a path every frame. A path may be updated every few frames depending on the complexity of the environment. G-SpAR will enable games built with Unity to incorporate spatial sound that is comparable to popular commercial games released by Ubisoft.

Alternatively, the GPU accelerated voxel method is able to calculate the shortest path in 3D for a large number of sound sources every frame. G-SpAR is able to switch between the two methods at run-time. The application could then decide mid-game that the GPU has become a performance bottle-neck and switch to the CPU method without any interruption in the audio playback.

Both methods are parameter based. PNSL is used to place a virtual sound source relative to the listener and apply attenuation based on the path-length (distance traveled taking into account obstacles). The last-leg of the shortest path is used to approximate acoustical occlusion, diffraction, and reflection which are applied using DSP effects built into Unity's sound engine.

2.4 Sound Engines

Just as the term "graphics engine" refers to graphical rendering APIs used to develop games and other virtual environments, a "sound engine" is the portion of a game engine devoted to rendering the audio component of the simulation. With

respect to video games, a number of audio APIs and specifications are available that do account for occlusion and reverberation effects.

Direct Sound

The Creative Labs Environmental Audio Extensions (EAX) is a set of extensions for Microsoft's DirectSound3D API that allows for adjustment of sound source and listener parameters and occlusion effects. Occlusion effects are divided into three categories [Menshikov 2003]: (1) occlusions, where the sound bends around an insurmountable obstacle that divides the sound source and the listener, (2) obstructions, where the sound source and the listener are in the same room, the direct path is blocked but reflected sound still reaches the listener, and (3) exclusions, where the sound source and listener are in different rooms but there is an opening between the rooms and the direct sound and only partial reflections reach the listener. Each of the three effects is accounted for with a low-pass filtering operation. For occlusions and reflections, the resulting sound is essentially muffled using a low-pass filtering operation. With obstructions the direct path sound is muffled while the reflections are not altered. With exclusions, the direct sound reaches the listener and the reflections are distorted.

Open Audio Library (OpenAL)

OpenAL is another commonly used audio API and when it implements the Environment Effects Extension (EFX) (that provides the same functionality as EAX but tailored to OpenAL), and it also supports occlusion mapping using simple low-pass filtering operations [Menshikov 2003].

Fmod (Firelight Technologies)

FMod is an audio API (sound engine) for the creation and playback of interactive (positional) audio that also supports occlusion modeling. In the FMod specification, occlusion is described as the changes in volume, frequency content, and reverberation that occur when sound passes through an object. Occlusion is accounted for by attenuating the volume of the direct and reverberant sound (when the direct path between the sound source and the listener is occluded), and attenuation of the high-frequency components of both the direct and reverberant sound using a low-pass filtering operation [Menshikov 2003].

2.4.1 Spatial Sound Plug-ins

Some of the current spatial sound APIs are not full featured sound engines. Instead they work with existing sound engines such as Fmod to provide improved spatial sound for an existing user base. A plug-in is "a software add-on that is installed on a program, enhancing its capabilities" [Computer Hope 2018].

Resonance Audio

Resonance Audio is an open source project designed to add spatial sound to video games and VR applications. More specifically, Resonance Audio provides an efficient way to add HRTF filtering and a simulated RIR for virtual environments. The RIR is not generated based on the actual models used to render the scene. Instead, Resonance Audio calculates the RIR based on a collection of rectangular boxes created by the designer. The box describes the size and reflectivity of the space. The designer can assign different materials to each of the walls, the ceiling

and floor. The material is selected from a list of materials provided by the system. Each surface may optionally be assigned the material “Transparent” which means that sound is able to pass through this surface and does not reflect. In this way, the transparent material allows the designer to remove surfaces to create open spaces. The room properties are interpolated over time to allow for smooth transitions as the listener moves from one room to the next [Kelly, Gorzel, & Güngörmüsler 2017]. Simple occluding objects can also be added to the scene to simulate structures that would block some of the sound. Resonance Audio does not trace the path of sound propagation however, so the sound source location is not based on the path taken by the sound due to occluding geometry [Resonance Audio 2019].

Resonance Audio is computationally efficient enough to provide spatial sound to mobile games, and does not require GPU processing. Android, iOS, Windows, MacOS, Linux, and other platforms are currently supported. Plug-ins are available for popular game engines such as Unity and Unreal. In addition, Resonance Audio can be used as a plug-in for sound engines such as Fmod and Wwise replacing the spatial audio rendering systems included with those engines [Resonance Audio 2019].

2.5 Discussion

GPUs optimised for ray tracing are relatively new. GPU ray tracing will likely become standardised leading to hardware agnostic ray tracing APIs. When this happens, the use of path tracing for computation of spatial sound will likely become standard as well. However, the transfer of data between the CPU and GPU is a

potential bottleneck. In the future, sound cards will likely perform this processing in order to reduce communication time.

By contrast, beam tracing produces polyhedral volumes with diffracting edges. The random sampling of ray tracing typically ignores diffracting edges. Instead, each ray travels in a straight line between reflections [Tsingos et al. 2001]. Beam tracing is well suited to environments consisting mainly of large flat surfaces and rectangular openings. However, modern game engines combined with highly parallel GPU architectures have led to the creation of complex densely populated virtual environments containing irregular shapes and curved surfaces. Each beam is spilt into several beams when a curved surface is encountered leading to increased processing time requirements. With ray tracing, each reflection is calculated based on the angle of intersection and the material parameters at the point of intersection. Therefore, a separate function to deal with curved surfaces is not required [Savioja & Svensson 2015].

In the distant future, hybrid systems could combine GPU efficient ray tracing with beam tracing to identify diffracting edges located near the listener. The resulting list of diffracting edges could be used to better simulate edge diffraction by allowing the rays to change direction mid-flight in response to nearby edges similar to sonel mapping.

Although ray tracing solutions currently exist, they have not seen widespread adoption by the video game industry. This may be due to the added processing

requirements or the reliance on new specialised hardware (Nvidia's VR Works Audio only works with the latest Nvidia GPUs). Currently, video game companies tend to employ pre-calculation where possible, or utilise a graph based system similar to Ubisoft. Graphs abstract away the complexity of the environment to produce a model that can be searched efficiently. Ubisoft's system is proprietary and only used internally by the company. G-SpAR is functionally similar to the Ubisoft system as described by Guay [2012] however, the details have been published. Both of these systems use nav-meshes to trace the path taken by sound in order to reach the listener. Nav-meshes may be processed efficiently, however, building a nav-mesh from the geometry used to render the environment is computationally intense. In most cases, the nav-mesh does not require real-time updating. A nav-mesh may not require updating at all if constructed based on a static (unchanging) environment. One potential problem with using a nav-mesh to process spatial sound is that they are generally constructed to represent a traversable floor space. Some obstacles that block a path may allow sound to pass through them unhindered (e.g., a chain link fence). A separate nav-mesh can be built specifically for the simulation of sound propagation to solve this problem. However, the 2.5D nature of the mesh is also limiting. A nav-mesh represents a flattened view of a 3D space. More specifically, it represents the floor plan without any information about the height of the room or the objects in the scene. A simple nav-mesh cannot, for example, simulate a sound source that is moving orthogonally with respect to the floor plain, passing over or under objects in the scene. In addition, some games allow the player to fly, or jump excessively high. The shape of the boundary walls and obstacles may vary at different heights within

a 3D space. This has led to the development of the system detailed in chapter six of this thesis, a fully 3D graph connecting every space that is open to sound propagation.

Chapter 3 GPU-Based HRTF Convolution

Fundamental to the generation of spatial sound is the one-dimensional convolution operation which is computationally expensive, not lending itself to such real-time, dynamic applications. Driven by the gaming industry and the great emphasis placed on the visual sense, consumer computer graphics hardware, and the graphics processing unit (GPU) in particular, has greatly advanced in recent years, even outperforming the computational capacity of CPUs [Akkas 2019]. This has allowed for real-time, interactive realistic graphics-based applications on typical consumer-level PCs. Given the widespread use and availability of computer graphics hardware and the similarities that exist between the fields of spatial sound and image synthesis, in this chapter, the development of a GPU-based, one-dimensional convolution algorithm whose efficiency is superior to the conventional CPU-based convolution method, is described. The primary purpose of the developed GPU-based convolution method is the computationally efficient generation of real-time spatial sound for dynamic and interactive videogames and virtual environments [Cowan & Kapralos 2008].

Convolution is a computationally expensive operation particularly when considering long filters associated with HRTFs and RIRs (filters with 512 coefficients are not uncommon) thus limiting their use to non-real-time applications (the operation described in Equation 3.1 must be performed for each input signal sample). Convolution is primarily performed in software in the time domain.

$$s[n] * r[n] = \sum_{k=-N/2+1}^{N/2} s[n-k]r[k] \quad \Leftrightarrow \quad S(f)R(f), \qquad (3.1)$$

Where **s** is the input signal, **r** is the filter, **n** denotes the sample index, **S** and **R** denote the discrete Fourier transform (DFT) of **s** and **r**, respectively, **f** denotes the DFT index and **N** denotes the filter sample size.

As shown in Equation 3.1, convolution in the time domain is equivalent to multiplication in the frequency domain and therefore, performance improvements can be made by performing the convolution operation in the frequency domain [Gardner 1995]. However, in order to accomplish this, the input and filters may have to be converted to their frequency domain representation using the *Fast Fourier Transform* (FFT); this can be a time consuming process when performed using software potentially limiting its real-time use. Recent work in image processing has established a GPU-based convolution method capable of performing a two-dimensional convolution operation in real-time [Fialka & Cadik 2006]. In contrast to the consumer-grade audio cards currently available, dedicated computer graphics hardware has evolved at a tremendous pace.

In an attempt to reduce computational requirements, a number of initiatives have investigated simplifying the HRTFs and RIRs. With respect to the HRTF, dimensionality reduction techniques such as *principal components analysis*, *locally linear embedding*, and *isomap*, have been used to map high-dimensional HRTF data to a lower dimensionality and thus ease the computational

requirements [Kapralos & Mekuz 2007; Kistler & Wightman 1992]. Despite the improvements with respect to computational requirements, even dimensionality reduced HRTFs are still not applicable for real-time applications and although the amount of reduction can be increased thus improving performance, reducing the dimensionality of HRTFs too much may lead to perceptual consequences that render them impractical [Kapralos & Mekuz 2007]. Further investigations must be conducted in order to gain greater insight. With respect to the RIR, it is usually ignored altogether and approximated by simply including reverberation generated with artificial reverberation techniques instead. These techniques are not necessarily concerned with recreating the exact reflections of any sound waves in the environment. Rather, they artificially recreate reverberation by simply presenting the listener with delayed and attenuated versions of a sound source. Although these delays and attenuation factors do not necessarily reflect the physical properties of the environment being simulated, they are adjusted until a desirable effect is achieved. Given the interactive nature of virtual environments and their need for real-time processing, when accounted for, reverberation effects in such environments are typically handled using such techniques.

Given the importance of the convolution operation in the generation of spatial sound but its demanding computational requirements, a GPU-based convolution method was developed that is capable of filtering a one-dimensional signal in real-time allowing for the generation of plausible spatial sound for dynamic, interactive applications such as videogames and virtual environments. The GPU-based convolution method detailed below was first published in 2008 [Cowan & Kapralos

2008]. The CUDA programming language was initially released by Nvidia Corporation in June of 2007 [Nvidia 2019b]. At that time, only the latest Nvidia GPUs supported CUDA. GPGPU programming was very new. Prior to the release of CUDA, researchers had experimented with using shading languages such as the OpenGL Shading Language (GLSL) for general purpose processing. GLSL is intended for rendering graphical effects, meaning that the output was typically a red, green, and blue (RGB) image consisting of one byte per colour channel. Despite the difficulties associated with converting data to and from images, GLSL was selected over CUDA because it was widely supported in that generation of GPUs.

3.1 Implementation Details

The implementation is based on the *OpenGL shading language* (GLSL). The input (un-processed) signal can be of any type (i.e., floating point, integer including the "short integer" 16-bit resolution common with WAV files used in many videogames). Of course, given an input signal that does not conform to this assumption, it can still be processed but at additional computational cost. The filter coefficients can be of any type (e.g., float, integer, etc.). For this work, an NVIDIA GTX-280 video card is being used. Although the implementation is applicable to all video cards, the GTX supports double precision floating point numbers allowing data to be stored with full accuracy thus avoiding the introduction of artifacts in the final (filtered) result (see *Results and Discussion* section).

GPUs have been designed to work with two-dimensional images as the output of typical computer graphics applications is a two-dimensional image. Therefore, prior to performing the convolution, there is a set-up phase to convert the one dimensional audio signal and filter, into a two-dimensional format as required by the GPU. The source (shader) code is provided below. With the GTX-280, the 16-bit (integer) input sound signal is stored in single channel 16-bit intensity maps (images) while the HRTF filter is sent to the GPU as an array of floats (i.e., it is not stored as an image). Although this is not completely necessary and the data can be divided into two 8-bit channels, it does lead to reduction in both computational requirements and round-off errors. A texture is then created from the data in OpenGL. However, this is accomplished using the CPU and not the GPU. To return the data back from the video card, the video card output must be copied from the display (screen) into arrays of bytes. These byte arrays must then be combined to form the desired output (see Algorithms 3.1 and 3.2).

Algorithm 3.1: GLSL Vertex Shader

```
varying vec2 texCoord;

void main(void){
      gl_Position = ftransform();
      texCoord = gl_MultiTexCoord0.xy;
}
```

Algorithm 3.1: This vertex shader renders a sprite orthographically (GLSL).

Convolution Fragment Shader

```
uniform vec2 imageSize;
varying vec2 texCoord;
uniform sampler2D image;
const int MAX = 200;
uniform float hrtf[MAX]; //float array sent from CPU

void main (){
      float x = 1.0 / imageSize.x;
      float y = 1.0 / imageSize.y;
      int   length = MAX;
      vec2  currentPos; //position being sampled
      float oldY;
      float total = 0.0; //running total
      vec3   base = vec3(0.0, 0.0, 0.0);
      float temp = 0.0; //used in calculations

      //Setup
      currentPos = texCoord;
      currentPos.x -= float(length) * x;
      if(currentPos.x < 0.0){
            currentPos.x = currentPos.x + 1.0;
            currentPos.y = currentPos.y - y;
      }
      oldY = currentPos.y;

      int i;
      for(i=0; i<length; i++){
            currentPos.y = oldY + floor(currentPos.x)*x;
            temp = texture2D(image, currentPos).r;
            temp = (temp * 64.0-32.0)* hrtf[i];
            total += temp;
            currentPos.x += x;
      }

      total += 128.0;
      int intTotal = int(total);
      base.r = float(intTotal)/256.0;
      base.g = total-float(intTotal);
      if(total > 128.0){
            base.b = 1.0;
      }
      else{
            base.b = 0.0;
      }

      gl_FragColor = vec4(base,1.0);
}
```

Algorithm 3.2: The fragment shader performing the HRTF convolution (GLSL).

Conversion Code

The code below runs on the CPU and is executed after the vertex and fragment shaders are executed. This code reads the pixel data from the frame buffer and converts it to a 1D array of unsigned integers (Algorithm 3.3). The blue channel is used to correct the output in the red channel. More specifically, the blue channel is used to prevent a strange bug in the output from the card. If a channel outputs a value greater than 128 out of 255, the channel's data is off by one. This can be a large problem when it is the red channel because it is multiplied by 256 meaning that the result would be off by 256. The blue channel acts as a flag to indicate if this has occurred in the red channel so that output can be corrected (this may be video card specific).

Image to audio file conversion code

```
glReadBuffer(GL_BACK);

glReadPixels(520, 600-wave.length/256-2,
tex3.width,wave.length/256+1, GL_RGB,  GL_UNSIGNED_BYTE,
data);

for(GLuint j=0; j<256*(wave.length/256+1); j++){
      Output[j] = data[j*3]*256 + data[j*3+1];
      if(data[j*3+2] >= 127)Output[j]+=256;
}
```

Algorithm 3.3: This code reads the pixel data back to the CPU and colours to an array of integers (C++).

3.2 Results

Here, a comparison of the computational running time requirements for both the conventional (software-based) and GPU-based convolution methods is made. This is accomplished by measuring the computational time requirements when

convolving a particular input signal with a filter for each method (the same input signal and HRTF was used for both methods). The filter considered for this test was an actual HRTF filter taken from the CIPIC HRTF dataset, measured from a KEMAR manikin using the "large ear" with the sound source positioned on the horizontal axis directly in front of the KEMAR [Algazi et al. 2001]. The filter coefficients were floating point numbers (i.e., "float") and of size 200 (although the filter coefficients considered here were floating point values, the proposed method can handle filter coefficients of any type). The input signal was a one-dimensional sine-wave signal (each sample had a resolution of 16 bits and of type "short int"). The size (number of samples) of the input signal ranged from 5,000 to 60,000, increasing in increments of 5,000. The tests were performed using a Dell XPS 720 PC with an Intel Core2 6700 (2.66 GHz) Processor with 2 GB of RAM and an NVIDIA GTX-280. The GTX supports double precision floating point numbers thus allowing data to be stored with full accuracy.

The results of this test are summarized in Table 3.1, and Figure 3.1 where a comparison of the average computational time requirements (x-axis) vs. the size of the input signal of conventional CPU-based (software) convolution and GPU-based (hardware) convolution using both the NVIDIA GTX 8800 and GTX 280 video cards is illustrated. Each point on the graph (both GPU and CPU-based implementations) represents computational time requirements averaged over 1,000 iterations. The GPU-based computational time requirements includes processing on the CPU which was performed to convert the data into the format required by the GPU.

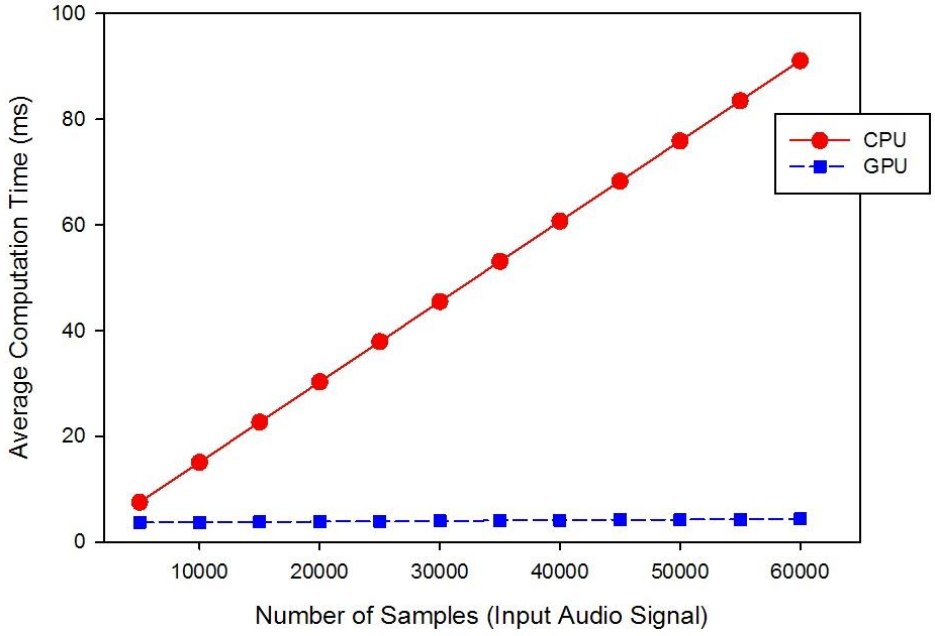


Figure 3.1: Comparison of input sample size vs. average running time for GPU- and CPU-based convolution. Filter size was constant at 200 samples.

Number of Samples	Time (ms) CPU	Time (ms) GPU
5,000	3.72	0.45
10,000	7.50	0.50
15,000	11.27	0.58
20,000	15.06	0.61
25,000	18.84	0.69
30,000	22.63	0.70
35,000	26.39	0.75
40,000	30.19	0.80
45,000	33.95	0.89
50,000	37.70	0.92
55,000	41.50	0.94
60,000	45.27	0.99

Table 3.1: Average computational time requirements. The first column represents input signal size (number of samples), the second column represents the average computational time requirements of the conventional CPU-based convolution method, and the third column represents the average computational time requirements GPU-based convolution method.

3.3 Discussion

As shown in Figure 3.1, GPU-based convolution is clearly superior to CPU-based convolution with respect to average computation time. In particular, the average computational running time for the GPU-based method ranged from 0.45 ms (input sample size of 5,000) to approximately 1 ms (input sample size of 60,000) to compute the convolution of an input signal and a filter with 200 coefficients. In contrast, the CPU computational time requirements ranged from 3.72 ms (input sample size of 5,000) to 45.27 ms (input sample size of 60,000). An average computational time of just under 1 ms for the convolution of an input signal with 60,000 samples and an (HRTF) filter with 200 coefficients corresponds to approximately 1,000 fps; clearly applicable for real-time operation. In contrast, the same convolution using the CPU-based method leads to a frame-rate of 22 fps. The results presented here are also an improvement from the prior results of Cowan and Kapralos (2009a) where, a running time of approximately 2 ms was observed for the convolution of an input signal with 60,000 samples and an (HRTF) filter with 200 coefficients. The implementation of that work was optimized leading to the results obtained here.

Given that the convolution operation involves floating-point number calculations, it is important that the video card support double-precision arithmetic to avoid any floating point-related errors which will manifest themselves in the final (filtered) result. As previously described, the NVIDA GTX-280 video card supports double precision computations and therefore floating point errors were not an issue here. However, previous work by Cowan and Kapralos [2008] previously investigated

GPU-based convolution using the NVIDIA GeForce 8800 which does not support double-precision arithmetic. The method introduced noise/artifacts to the lower-order bytes of the resulting GPU-based convolution output. This noise resulted from the limitations specifically with the NVIDIA GeForce 8800. More specifically, the GeForce 8800 was returning values with an 8-bit accuracy thus not allowing "images" to have 16-bits per channel. As a result, the 16-bit input sound signal was divided into two 8-bit values (via the red and green channels of the image), combined in the shader and stored as floats. Furthermore, the 8-bits per channel implied that the input had to be divided between two channels. Although this required slightly more computation, it did not interfere with accuracy.

3.4 Comparison with Compute Shader Implementations

In 2013, the GLSL convolution method detailed above was compared with a CUDA and OpenCL implementation in order to determine whether the hardware support guaranteed by GLSL was worth the added processing time [Cowan & Kapralos, 2013b]. The implementation and results of that study are included below.

The GPU-based convolution method outlined above was implemented using GLSL. However, there are several other widely used shader languages including NVIDIA's Compute Unified Device Architecture (CUDA) and OpenCL. In this section, a comparison between the running times of five different one-dimensional convolution implementations is provided. Three of these implementations employ the graphics processing unit, one consists of the traditional, software-based

implementation, while the other employs (software-based) threads. The following five implementations were considered:

1. Linear (traditional CPU software-based).
2. Concurrent (multi-threaded CPU implementation).
3. OpenGL Shading Language (GLSL).
4. CUDA, NVIDIA's parallel computing platform and programming model.
5. Open Computing Language (OpenCL) open, royalty free, cross-platform, parallel programming platform.

3.4.1 GLSL Implementation

The OpenGL shading language (GLSL) implementation was updated from the implementation described above [Cowan & Kapralos, 2009a]. GLSL is designed for graphics processing and primarily uses two-dimensional images as both input and output. Therefore, prior to performing the convolution, there was a set-up phase to convert the one dimensional audio signal into a two-dimensional image format as required by GLSL. The 16-bit (integer) input sound signal is stored in single channel 16-bit intensity maps (images) while the HRTF filter is sent to the GPU as an array of type float (i.e., it is not stored as an image). Although this is not completely necessary and the data can be divided into two 8-bit channels, it does lead to a reduction in both computational requirements and round-off errors. A texture is then created from the data in OpenGL. To return the data back from the video card, the rendered output must be copied from the frame buffer into an array

of unsigned short integers. The array must then be converted from an unsigned type to a signed type before it can be played back as audio.

3.4.2 CUDA Implementation

NVIDIA's CUDA is a parallel computing platform and programming model that makes utilizing a GPU for general purpose computing simple by allowing programmers to write code using high level languages based on C, C++, and Fortran [Ebersole 2012]. The CUDA programming model consists of a traditional CPU or host, and one or more 'compute devices' which are massively data-parallel coprocessors [Nvidia 2008]. In order to perform the convolution algorithm on the GPU using CUDA, the arrays must be copied from the host (CPU) to the device (GPU). Unlike GLSL, CUDA accepts all of the standard data types as input. The array containing the audio samples can be processed as unsigned short integer types without the need to store any of the data as images. The output array must then be copied back to the host where it can be played back as audio. A kernel function defines the process that is executed in parallel and cannot be called directly from C++ code. A separate CUDA function that controls the memory allocation and transfer is also responsible for creating the worker threads (instances of the kernel function) and copying the output data back to the CPU once all of the threads have completed.

3.4.3 OpenCL Implementation

The Open Computing Language (OpenCL) is a framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, DSPs and

other processors [OpenCl 2013]. The OpenCL framework is built on top of CUDA and DirectX 11. The OpenCL compiler attempts to optimize the kernel code (i.e., unrolling loops) and although CUDA has similar optimizations, they are turned off by default. Programmers must enable individual optimizations using pragma statements (i.e., 'pragma unroll'). OpenCL implementations may outperform the CUDA equivalent of the same algorithm, unless careful attention is given to tuning or optimizing the CUDA compilation settings [Fang et al. 2011]. In the comparisons, the default compilation settings were used for both CUDA and OpenCL, and the kernel code was written to be as similar as possible with respect to both function and syntax.

3.4.4 Linear (CPU-Based) Implementation

The CPU-based convolution method was performed completely in software without any parallelism (i.e., a single thread).

3.4.5 Concurrent (CPU multi-threading) Implementation

The threaded implementation was also performed entirely in software but now parallelism via multi-threading was employed. More specifically, the pThreads library was employed to divide the task equally between eight threads.

3.5 Computation Time Comparison

A comparison of the five convolution implementations consisted of performing the convolution operation with each implementation on the same computer (using the identical configuration) and using the identical input and filter coefficients. The

input (un-processed) signal consisted of a “short integer” data type with 16-bit resolution (common with .WAV files used in many videogames for example). Of course, the convolution implementations will work with an input signal that does not conform to this assumption, although depending on the input signal type, additional processing may need to be performed. The input signal consisted of a one-dimensional sine-wave (each sample had a resolution of 16 bits and of type “short int”). The size (number of samples) of the input signal ranged from 5,000 to 100,000, increasing in increments of 5,000 for the purpose of this comparison. The filter consisted of 200 floating point-type coefficients taken from the CIPIC HRTF dataset [Algazi et al. 2001]. The resulting filtered signal was saved to the hard-disk to allow for a comparison of the accuracy of the resulting output.

The tests were performed using an Alienware M14X laptop with an Intel Core i7-2670QM (2.2 GHz) Processor with 8 GB of RAM and an NVIDIA GeForce GT 555m graphics card with a 144 core GPU and 3 GB DDR3 dedicated video memory. The laptop was running the Windows 7 Home Premium operating system.

A summary of this test is provided in Table 3.2 and a graphical summary is provided in Figure 3.2, where the x-axis represents the size of the input signal (number of samples), and the y-axis represents computational time (in milliseconds). Each point on the graph (both GPU and CPU-based implementations), represents the computational time requirements averaged over 1,000 iterations. The three GPU-based computational time requirements includes the time spent transferring the HRTF and audio signal to the GPU, in addition to

the time required to transfer the output (filtered) data back to the CPU. The processing time for the GLSL implementation includes the additional time required to convert the array of audio data from unsigned to signed short integers and back. The additional processing is performed linearly (one thread) on the CPU and takes approximately 0.625 ms for an input file containing 100,000 samples. It is important to note that the conversion from the signed to the unsigned type may not be necessary for use with audio APIs that support unsigned data as input. Examination of Figure 3.2, clearly indicates that the Linear (software-based) implementation is by far the most inefficient method (which is linearly proportional to the size of the input), followed by the threaded (pThreads) implementation. The GPU-based implementations are far more efficient, each of which varies only slightly as the number of input signal samples is increased.

Each of the three GPU-based methods are able to perform the convolution operation in far less time than the fastest CPU-based implementation. CUDA was able to process 100,000 samples in 2.81 ms compared to GLSL which required 4.68 ms to process the same amount of data. OpenCL processed 100,000 samples in 2.03 ms, 28% less time when compared to our CUDA implementation. However, CUDA outperformed OpenCL for smaller sample sizes. For example, CUDA processed the 5,000 sample audio file in 0.55 ms, while OpenCL required 0.78 ms. Both OpenCL and CUDA outperformed GLSL at every sample size considered proving that general purpose GPU programming languages are better suited to processing non graphics related data. However, all three GPU methods were able to meet the real-time requirement of 33 to 67 ms. Even the threaded CPU method

was able to process the 100,000 sample audio signal in real-time (25.19 ms which roughly equates to 40 fps). An examination of the filtered data output/generated by each of the five implementations revealed that the output from the GLSL implementation contained a small amount of rounding error. More specifically, 14.8% of the filtered samples were off by one when compared to the output from the other methods. A small amount of rounding error is to be expected when using GLSL for general purpose computation. However, the amount of error introduced would not alter the sound enough to be perceived, and can therefore be considered insignificant.

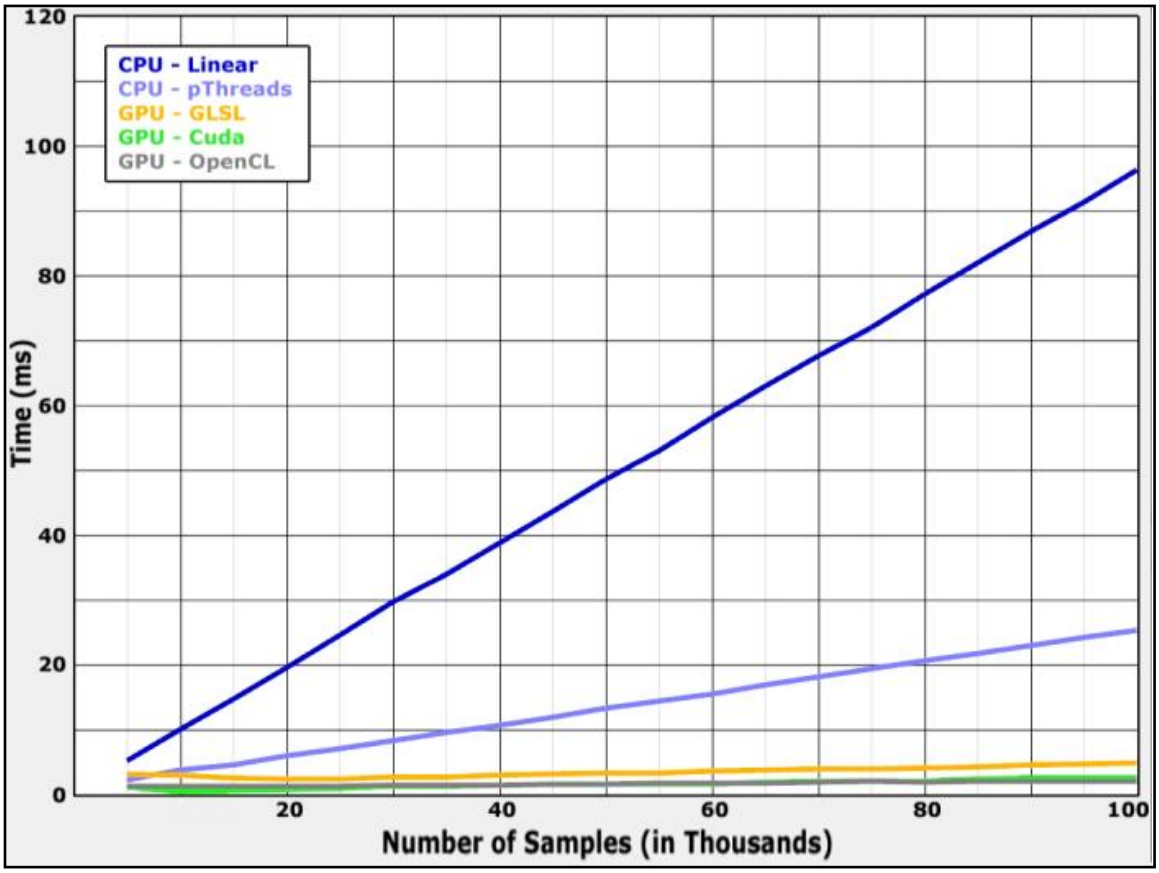


Figure 3.2: Computational time requirements for each of the five methods.

Input Size			Running Time (ms)		
Linear (CPU)	Single Thread (CPU)	pThreads (CPU)	GLSL (GPU)	CUDA (GPU)	OpenCL (GPU)
5,000	4.84	1.95	2.58	0.55	0.78
10,000	9.60	3.28	2.03	0.55	0.94
15,000	14.12	4.29	1.95	0.62	1.01
20,000	18.57	5.54	2.18	0.78	1.02
25,000	23.17	6.87	2.27	0.78	1.09
30,000	27.77	8.12	2.42	0.71	1.10
35,000	32.53	9.28	2.58	0.86	1.17
40,000	37.05	10.53	2.81	1.02	1.25
45,000	41.66	11.70	3.05	1.17	1.33
50,000	46.41	12.95	3.20	1.25	1.33
55,000	51.40	14.12	3.20	1.41	1.40
60,000	55.54	15.37	3.43	1.48	1.56
65,000	60.37	16.54	3.59	1.64	1.56
70,000	65.45	18.02	3.75	1.72	1.64
75,000	69.34	19.19	3.82	1.95	1.64
80,000	73.95	20.36	4.06	2.03	1.80
85,000	78.55	21.53	4.21	2.11	1.87
90,000	83.46	22.78	4.37	2.34	1.87
95,000	87.83	24.03	4.52	2.50	1.87
100,000	92.59	25.19	4.68	2.81	2.03

Table 3.2: Average computational time requirements in milliseconds. The first column represents the input signal size (number of samples), while the rest of the columns represents the average computational time requirements for each of the five methods. The filter size was kept constant at 200 samples.

One major disadvantage to GPU-based sound processing is lag. The GPU is capable of processing a large amount of data concurrently. However, GPU's are less efficient at processing small amounts of data. The amount of time required to read processed data back from the GPU may be greater than the processing time for small data sets. For example, OpenCL required 0.78 ms in order to convolve 5,000 samples, and 2.03 ms to convolve 100,000 samples. It is far more efficient to process a few seconds of audio at a time instead of processing a small number of

samples each frame. However, processing sound in large blocks would mean that the sound source's position is updated less often. The relative position used to calculate the HRTF filtering would lag behind the actual position of the sound source. It should be noted that the convolution work and processing time comparisons discussed here were made several years ago and newer cards will likely result in noticeably better performance [Cowan & Kapralos 2013b].

The convolution methods discussed here have become less relevant now that many modern sound cards have some form of HRTF filtering built into their hardware. However, GPU-based audio processing still has many practical applications. The GPU could be used to process complex DSP effects in real-time such as altering the player's voice in an online game. The GPU can also be used to create multiple versions of a sound effect while the game is loading. Replaying the same sound effect each time the player jumps, or fires a weapon can make the audio boring or even annoying to listen to. GPU sound processing would several variations of an effect to be created from a single sound file while the game is loading.

3.6 Summary

To overcome the limitations associated with software-based convolution, here, a hardware-based convolution method that takes advantage of the tremendous computational ability of the affordable and commonly available graphics processing units (GPUs) was presented. The method was implemented using the OpenGL shading language (GLSL). Results indicate that the method is far more computationally efficient when compared to conventional, time-domain, software-

based convolution and is in fact capable of performing convolution of a filter containing 200 coefficients and a one-dimensional signal of up to 60,000 samples, in real-time (approximately 1 ms). Given that the generation of spatial sound hinges on the convolution operation and the widespread availability of computer graphics cards with onboard programmable GPUs, the generation of accurate virtual audio for games and virtual environments is now plausible.

Chapter 4 Occlusion Rendering

The convolution work detailed in Chapter 3 has demonstrated that the GPU can be used as a general purpose processor that can be applied to non-graphic-based processing, including audio processing. Although the convolution algorithm was not optimised in any manner, the computation time was reduced solely by taking advantage of the GPU's ability to process data in parallel.

GPUs are designed to render images based on geometry. Shader effects are typically used to simulate the way that light is absorbed or reflected by each surface in the environment. Simple game lighting does not always take into account multiple reflections (real-time ray tracing) or even whether the path between the surface and the light source is blocked (shadow mapping). Similar to light, sound emanates outward in all directions from the source (assuming an omni-directional sound source). Sound can also be absorbed or reflected by objects in the environment. Specially designed shaders can be created to approximate the effect that each surface has on the propagation of sound in order to approximate acoustical occlusion. In addition, the resulting 2D occlusion map can be post-processed in order to approximate acoustical diffraction.

In this section, implementation details of the GPU-based occlusion modeling method (i.e., the shader) are provided (the method is introduced in by Cowan & Kapralos [2010]). This method utilises the actual scene geometry, which may include moving objects, to approximate occlusion/diffraction effects. Rather that

attempt to build a complete sound engine, here the focus is on approximating occlusion effects efficiently and passing the results to an existing commercial sound engine (Fmod) in order to increase realism. The implementation is based on the OpenGL shading language (GLSL) [Rost 2006] (see Appendix A.1 and A.2 for the GLSL shader source code). The input to the shader is the scene/model and the position of the sound source and listener/receiver. The scene itself can be arbitrarily complex (i.e., any number of objects) although for illustrative purposes, the test environments presented here have been purposely kept simple.

For computational efficiencies, the distribution of sound frequency in a given sound source is typically approximated by a fixed number ($\boldsymbol{N}_{\mathbf{freq}}$) of frequency bands [Mehta et al. 1999]. Here, only two frequency bands are considered; low and high frequencies. Low frequencies are defined to be less than 4 kHz, and high frequencies are greater than or equal to 4 kHz. However, as will be described later, the proposed occlusion modeling method operates in real time; average running time of 0.39 ms irrespective of the scene complexity (Nvidia GeForce GTX 280 GPU). The method can, therefore, be easily extended to account for a greater number of frequency bands and considering only two frequency bands is purposely done to simplify the explanation and resulting illustrations. The method is completely dynamic and operates irrespective of the sound source, the listener, and object position and orientation; the sound source, the listener, and objects can freely move about within the environment. The proposed method is a two-stage technique; acoustical ray casting followed by receiver scaling. Both stages are detailed in the following sections.

4.1 Acoustical Ray Casting Stage

The purpose of the acoustical ray casting stage is to produce a listener occlusion map which, for every position in the scene (pixel), provides an occlusion weighting ($\boldsymbol{w}_{\mathbf{occlusion}}$) within the range of [0…1]. zero ("black") indicates the sound source is completely blocked (non-visible) and no sound reaches the listener, while a value of one ("white") indicates the sound source is completely visible to the listener and direct sound is able to reach the listener. Shades of gray in between represent partial occlusion whereby some of the sound is able to reach the listener. The scene is rendered (using the GPU) from the sound source's perspective. To construct the occlusion map, I initially began with an empty ("white") canvas. A total number ($\boldsymbol{N}_{\mathbf{vis}}$) of visibility rays are emitted from the sound source to the receiver in order to determine the visibility between the sound source and the receiver (i.e., the visibility between the sound source and the receiver is determined by calculating the number of rays that that are not occluded and can therefore reach the listener) using the GPU. Basically, a three-dimensional area between the sound source and the listener is sampled using ray-casting. This area represents a view frustum with the apex located at the sound source and the far viewing plane centered on the listener's position. The number of visibility rays emitted depends on the resolution of the occlusion map. More specifically, one ray is emitted for each pixel in the occlusion map. The size of the occlusion map was set to 128 × 64 ($\boldsymbol{N}_{\mathbf{vis}}$ = 8,192 rays) and although the size can be adjusted, increasing the size of the occlusion map does not improve the result noticeably but it does increase the computational requirements. That being said, a very low-resolution occlusion map (e.g., 32 × 16) does not closely represent the shape of the occluding objects. The occlusion map is

recomputed every time a sound source, listener, or object in the environment moves.

Since the scene is being rendered from the sound source's position, the sound source will be "in front of" any surface being rendered. The dot product $\boldsymbol{o_s}$ of the surface's normal $\boldsymbol{s_n}$ and a "test vector" $\boldsymbol{v_t}$ (a normalized vector starting at the listener and facing the point on the surface currently being rendered) is calculated to determine the amount of occlusion

$$o_s = v_t \cdot s_n \quad (4.1)$$

when $\boldsymbol{o_s} \leq 0$, the listener is in front of the surface being rendered and the surface does not occlude any sound, and therefore, $\boldsymbol{o_s}$ is assigned a value of 0 (i.e., $\boldsymbol{o_s} = 0$). If the surface's normal $\boldsymbol{s_n}$ is facing away from the sound source, then the surface is not rendered due to back face culling. When $\boldsymbol{o_s} > 0$, rather than assuming the surface occludes 100% of the sound, the occlusion weighting ($\boldsymbol{w_{occlusion}}$) is assigned the value returned by the dot-product operation ([0 … 1]). The scene is rendered using a perspective projection relative to the sound source. To calculate the occlusion/diffraction between one sound source and one listener, the camera is placed at the location of the sound source and pointed toward the listener. The distance of the far clipping plain is set to match the listener's distance from the sound source so that the listener is located in the center of the flat bottom of the view frustum (the frustum has a flat bottom but the shader discards fragments beyond a certain distance thus rounding the bottom). Thus, the listener, which is not rendered, is in the exact center of the rendered scene. An occlusion value is

calculated for each pixel in the render by the shader based on the occluding geometry.

Rendering the scene using a perspective projection causes the sudden and complete occlusion of the sound when the "camera" (representing the sound source) is close to an occluding object. Rendering the scene orthographically would correct this problem but would subsequently introduce new problems. More specifically, rendering a large enough view orthographically will cause the near clipping plane to intersect solid objects but, since the back facing surfaces are not typically drawn, objects that should occlude would be depicted as empty space. To account for this "camera blocking" problem, when the distance between the sound source and the surface/object ($\boldsymbol{d_s}$) is less than a pre-defined amount $\boldsymbol{\delta}$, the sound is further scaled as follows:

$$o_s = o_s \times \min\left(\frac{d_s}{\delta}, 1.0\right), \qquad (4.2)$$

When $\boldsymbol{d_s} < \boldsymbol{\delta}$, the sound level is reduced in a distance-dependent manner such that the closer the object/surface is to the sound source, the lesser the reduction. If $\boldsymbol{d_s} = 0$ (i.e., the object is touching the sound source), the sound is allowed to "pass through" without being reduced. Allowing sound to pass through objects that are close to the camera in this manner is only an approximation and not necessarily based on real-world acoustics. However, it can be computed quickly and it prevents small objects very close to the camera from completely blocking sound, which would be unrealistic and noticeable.

Once o_s is computed, it is subtracted from the current occlusion map pixel value (i.e., subtracted from the scene using OpenGL blending). The order that objects are subtracted does not matter. Beginning with a scene that is completely open (i.e., a "white" occlusion map), layer after layer of occluding objects are added by subtracting the occlusion weighting ($w_{occlusion}$) of the object, on a pixel-by-pixel basis, from the corresponding pixel value in the occlusion map, simulating the fact that overlapping occluding objects have a cumulative effect on occlusion. A perfect occluder is represented in the occlusion map as solid black while partial occluders are represented partially transparent. When multiple partial occluders are placed one in front of the other, they collectively cause the occlusion map to become darker. The occlusion data are then placed in the green channel, one byte per pixel (the red and blue channels were not used initially). Figure 4.1 is a top-down view of a simple test environment consisting of two rooms connected by an open doorway. One of the rooms contains the sound source and four cylinder shaped pillars. The other room contains the listener and several box shaped objects. An example illustrating the construction of the occlusion map in a "step-by-step" basis is illustrated in Figure 4.2 (for illustration purposes, occlusion is represented in shades of gray). All of the occluding objects in the scene (walls, pillars, and boxes) are set to occlude 100%, meaning that no sound is able to pass through them. However, based on the surface angle, they may reflect sound in a direction that may lead to the listener. The grayscale value assigned to each surface can be considered to represent the probability of sound reaching the listener after being reflected by this surface.

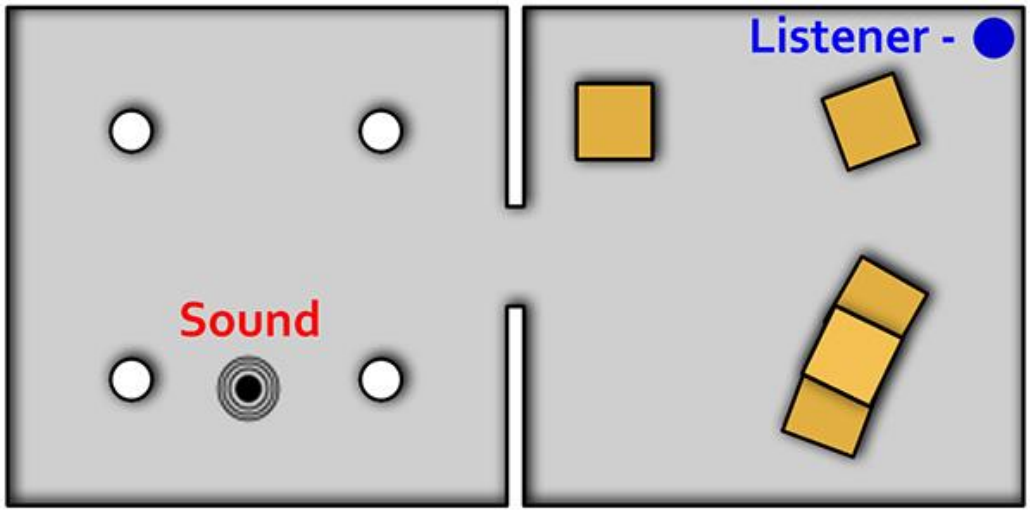


Figure 4.1: A top-down map of the test environment showing the placement of the sound source, listener, and several sound occluding objects.

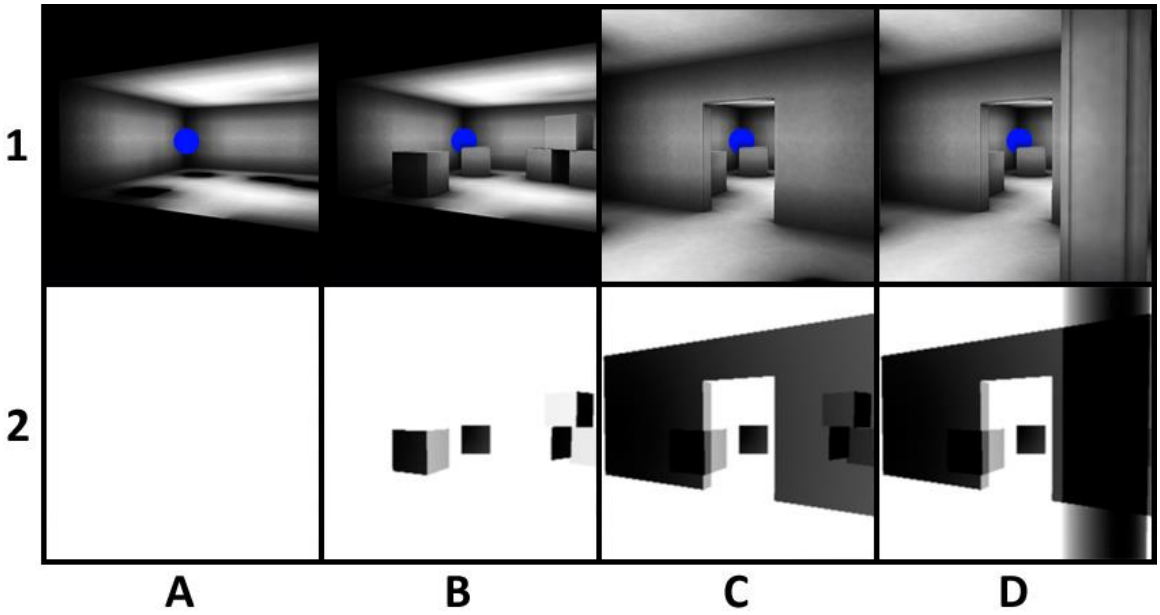


Figure 4.2: Sample occlusion map construction. Row 1 shows the scene rendered normally with a grayscale ambient occlusion texture applied to models (generated with Maya, a commercial 3D modelling software). A blue circle marks the location of the listener. Row 2 shows the same models as row 1 but rendered with the occlusion shader. The scene is rendered from the sound source to the listener. Beginning with (A), objects are added to the render until the complete scene with all of the objects is rendered in (D). When multiple partial occluders are placed one in front of the other, they collectively cause the occlusion map to darken

4.2 Receiver Scaling

The output of the acoustical ray casting stage is the grayscale occlusion map. The occlusion map provides, from the sound source's perspective, a quick approximation to the amount of sound that will reach the listener. The occlusion map is then scaled using a 2D scaling filter. The size of the scaling filter is identical to the size of the occlusion map (i.e., 128 × 64 pixels), and the scaling is performed by overlaying the scaling filter on top of the occlusion map (since the scene is rendered from the sound source to the listener, the listener is always positioned exactly in the center of the render, and likewise is always positioned in the center of the scaling filter) and multiplying each pixel of the occlusion map by its corresponding scaling filter value

$$O_{scaled}(i,j) = (O_{map}(i,j) \times f(i,j)), \qquad (4.3)$$

where, $\boldsymbol{O_{scaled}(i,j)}$ is the value of the scaled pixel at location $\boldsymbol{i,j}$, $\boldsymbol{O_{map}(i,j)}$ is the occlusion weighting factor ($\boldsymbol{w_{occlusion}}$) of the pixel in the occlusion map pixel at location $\boldsymbol{i,j}$, and $\boldsymbol{f(i,j)}$ is the scaling filter coefficient value at location $\boldsymbol{i,j}$. After the occlusion map has been filtered, the values of all the pixels in the map are summed and this sum is then scaled by the sum of all filter coefficients, yielding a single floating point value ranging between 0 and 1 and denoted by $\boldsymbol{s_{map}}$.

$$s_{map} = \frac{\sum_{i,j} O_{map}(i,j)}{\sum_{i,j} f(i,j)}. \quad (4.4)$$

As Hillesland and Lastra [2004] describe, although GPUs have floating point representations similar to and at times matching the IEEE standard, GPUs do not adhere to the IEEE standards [IEEE 1987], and do not provide the necessary

information to establish a bound on floating point operation errors. Therefore, given that the scaling filters contain "very small" floating point values, to avoid potential floating point errors, the filtering operation is performed using the CPU. Since the scaling filters remain constant, they are calculated once during the "initialization period" and re-used as needed; the filtering operation requires minimal computational resources. Scaling filter coefficient values range from one (center of the scaling filter) to zero (each of the four corners). Each "pixel" within $\boldsymbol{O}_{\mathbf{scaled}}$ represents the sound level for a corresponding location in three-dimensional space. The scaling filter is used to approximate the dependency of frequency and obstacle size on diffraction. This filtering operation is specific to one sound source and one receiver pair and must be repeated for each sound source/receiver combination.

Currently, frequency-dependent effects of diffraction are approximated using two scaling filters only; one for high frequencies (≥4 kHz) and another for low frequencies (<4 kHz) (see Appendix A.3 for the source code used to build both the low- and high-frequency scaling filters). A graphical illustration of low- and high-frequency scaling filters is illustrated in Figure 4.3a and 4.3b, respectively. The low-frequency scaling filter contains a wider lobe ensuring that a wider area of sound reaches the listener (i.e., a greater number of non-zero scaling filter coefficients). In contrast, the high-frequency scaling filter has a narrower lobe thus blocking a greater amount of the overall sound given that a large number of the scaling filter coefficients are zero. Currently, the parameters specifying the lobe width of the

scaling filters have been determined through "trial and error" and through informal listening tests.

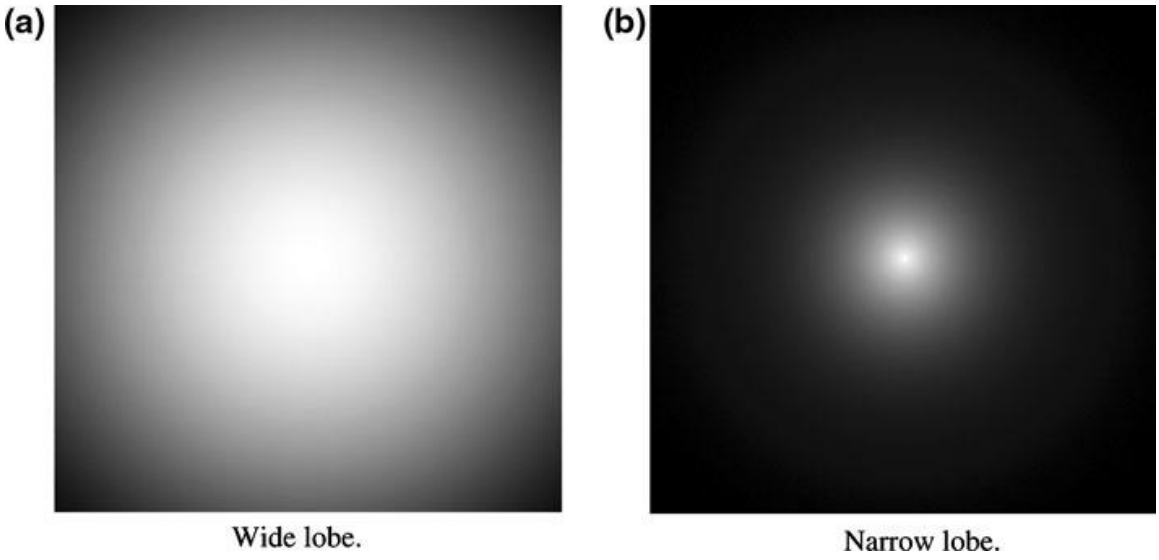


Figure 4.3: 2D sinusoidal scaling filters used to filter (scale) the occlusion map. (a) low-frequency (wide) lobe filter and (b) high-frequency (narrow) lobe filter

Once the scaling has been performed, the sound level for a particular location in the three-dimensional space can be scaled to account for distance-dependent attenuation affects by the medium (air). Assuming planar sound waves, the attenuation of sound energy due to absorption by the air follows an exponential law (Equation 4.5) [Kuttrff 2000]

$$E_r = E_o e^{-mr}, (4.5)$$

Where, $\boldsymbol{E_o}$ is the original sound energy, $\boldsymbol{E_r}$ is the energy after the sound has traveled a distance $\boldsymbol{r}$, and $\boldsymbol{m}$ is the air absorption coefficient that varies as a function of the conditions of the air itself (e.g., temperature, frequency, humidity, and atmospheric pressure). Expressions for the evaluation of **m** are provided by Bass et al. [1990]. Alternatively (and as done in this work), rather than explicitly

attenuating the sound to account for attenuation effects by the medium, the values from the occlusion map can be passed to the audio engine (e.g., Fmod as used in this work, OpenAL, etc.) where further processing can be performed including distance-dependent attenuation.

4.3 Results: Comparisons to Real-World Acoustical Properties

Several simulations were performed with various sound source, receiver, and environmental configurations to demonstrate that the proposed acoustical occlusion modeling method conforms to real-world acoustical occlusion / diffraction models. Although desirable, validation involving human user tests is beyond the scope of this work. Validation that involves comparisons between measurements made in an actual (controlled) room and the results of the same measurements made in a simulation of the room are also not included. Such tests require the use of a controlled room (environment), where various parameters (e.g., surface reflection and absorption coefficients) can be carefully controlled in order to allow meaningful comparisons to be made. Given the lack of such available data for a simple room whose parameters can be easily controlled, constructing such an environment is also beyond the scope of this work.

All the simulations described in this section were performed using a Dell XPS 720 workstation with an Intel Quad Core 2.4 GHz processor, 4 Gb RAM, Sound Blaster X-Fi sound card, and an NVIDIA GeForce GTX 280 video card (over-clocked, 1 GB-

GDDR3 SDRAM) that supports double precision floating point numbers. The size of the occlusion map was 128 × 64 pixels (8192 pixels). Rendering the occlusion map is equivalent to casting a single ray for each pixel of the occlusion map.

4.3.1 Graphical Illustrations

In this section, the results of modeling the occlusion effects of an environment (room) consisting of a sound source, receiver, and occluder configuration are presented graphically. As shown in Figure 4.4, the dimensions of the room were 10 m × 8 m × 10 m. A low-frequency (500 Hz with $\boldsymbol{\lambda}$ = 0.685 m) sound source was used (although only low and high frequencies are considered by the method, a 500 Hz source was used to ensure comparisons could be made with the sonel mapping approach as described later). The sound source was positioned at location (0.685, 4.0, 4.83) and remained stationary while the position of the receiver was varied across the $\boldsymbol{x}$–$\boldsymbol{y}$ plane (i.e., $\boldsymbol{z}$-axis remained constant at $\boldsymbol{z}$ = 4) in equal increments equal to $\boldsymbol{\lambda}$/2 or 0.685 m (the $\boldsymbol{x}$ coordinate ranged from 1.37 to 8.926 m, while the $\boldsymbol{y}$ coordinate ranged from 0.345 m to 8.97 m). The dimensions of the occluder were 3.0 9 5.0 m and the occluder was positioned such that it formed a plane in the $\boldsymbol{x}$–$\boldsymbol{z}$ axis. The coordinates of the vertices comprising the occluder were (3.45, 0.0, 3.45), (3.45, 5.0, 3.45), (3.45, 0.0, 6.45) and (3.45, 0, 6.45).

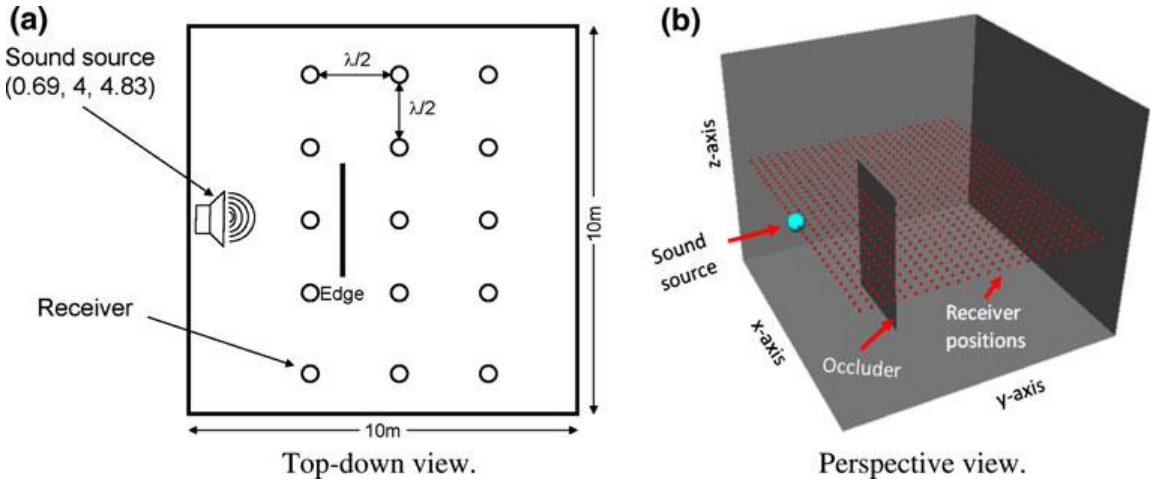


Figure 4.4: Room set-up for the individual component graphical demonstrations (not to scale). The height (z-coordinate) of the sound source and the receiver positions was constant at $z = 4$. Receiver positions varied across the x–y plane in equal increments of λ /2 or 0.685 m (the x coordinate ranged from 1.37 to 8.926 m while the y coordinate ranged from 0.345 to 8.97 m)

The results of this simulation are shown in Figure 4.5a. Sound level (power) is coded in shades of gray where white denotes full sound level (90 dB), black complete silence (0 dB), and shades of gray between represent levels in-between. In this example, for the purposes of illustration, attenuation effects of both the air and distance traveled were completely ignored. As shown in Figure 4.5a, although the direct path between the sound source and the listener is completely blocked, sound is still able to reach the listener due to diffraction. As a comparison, using the same room, sound source, and receiver configuration, the occlusion effects were modeled using the sonel mapping method that accounts for occlusion/diffraction effects using a modified version of the Huygens-Fresnel principle [Kapralos et al. 2008] (see Figure 4.5b). A comparison between the results obtained with both approaches illustrates that both methods allow sound to be diffracted so that sound does reach the area behind the occluder although the shadow region is far more pronounced for the sonel mapping method. In contrast,

the shadow region for the occlusion modeling method presented here is smoother but extends for a larger region beyond the occluder.

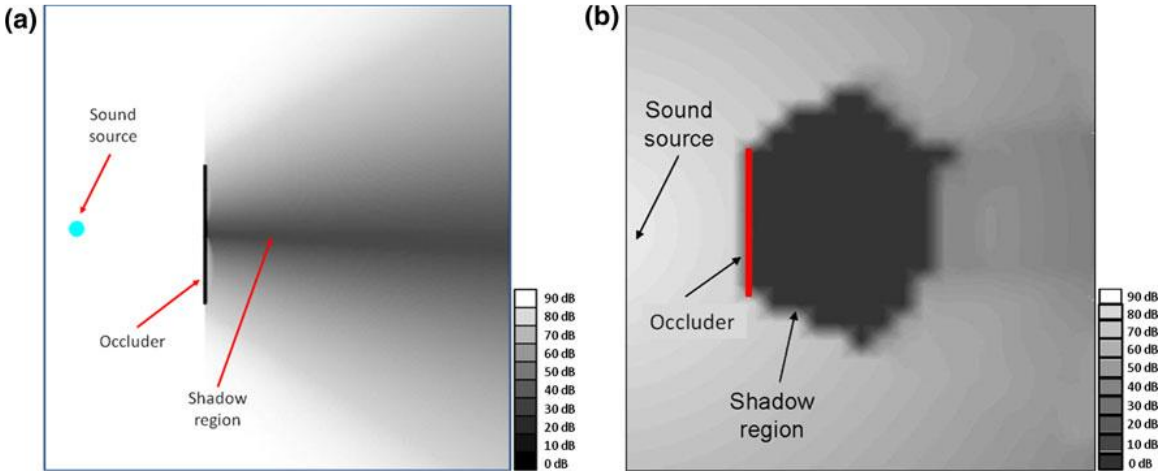


Figure 4.5: A comparison of the proposed occlusion modeling method and the sonel mapping method that accounts for occlusion/diffraction effects using a modified version of the Huygens-Fresnel principle (Kapralos et al. 2008). The room configuration illustrated in Figure 4.4 was simulated using both methods. Results using the proposed GPU-based occlusion modeling method are shown in (a), and results using the sonel mapping method are shown in (b). For both simulations, sound level (power) is coded in shades of gray where white denotes full sound level, black complete silence, and shades of gray between represent levels in-between

4.3.2 Graphical Comparison To the "Base-Case"

In this section, the effectiveness of the occlusion mapping method is presented visually by comparing it to the "base-case" ray-casting method, whereby a single "visibility ray" from the sound source to the listener (or from the listener to the sound source) is cast to test for occlusion. If the ray is unobstructed, the sound is played normally. However, if the visibility ray encounters an object, it is assumed that the sound is completely blocked and therefore, the corresponding sound is not played (i.e., silence). In other words, occlusion is treated as a binary state: the sound is either ON or OFF. Although such an approach is simple, it is problematic for a number of reasons including the fact that switching sounds between these two

states is noticeable and unrealistic (e.g., consider a listener standing in front of an open doorway; sound emanating from inside the room may be completely unoccluded but if the listener were to take one step to the side, the sound may instantly switch to the occluded state).

The scene illustrated in Figure 4.6 consisted of four objects (two cubes, a sphere, and a cylinder), and a sound source positioned in the center was rendered using both the base-case approach and the proposed occlusion modeling method. The dimensions of each of the two (red and blue) cubes were 10 cubic meters, the (green) cylinder was 10 m high with a 10 m diameter, while the (yellow) sphere had a diameter of 10 m. The center of each of these objects was placed 25 m from the sound source. All objects rested on a plane measuring 100 m × 100 m and the sound source was positioned 4 m above this plane. The volume was sampled at a constant height of 4 m across the entire plane. A low-frequency sound source was used (250 Hz with $\boldsymbol{\lambda}$ = 1.37 m).

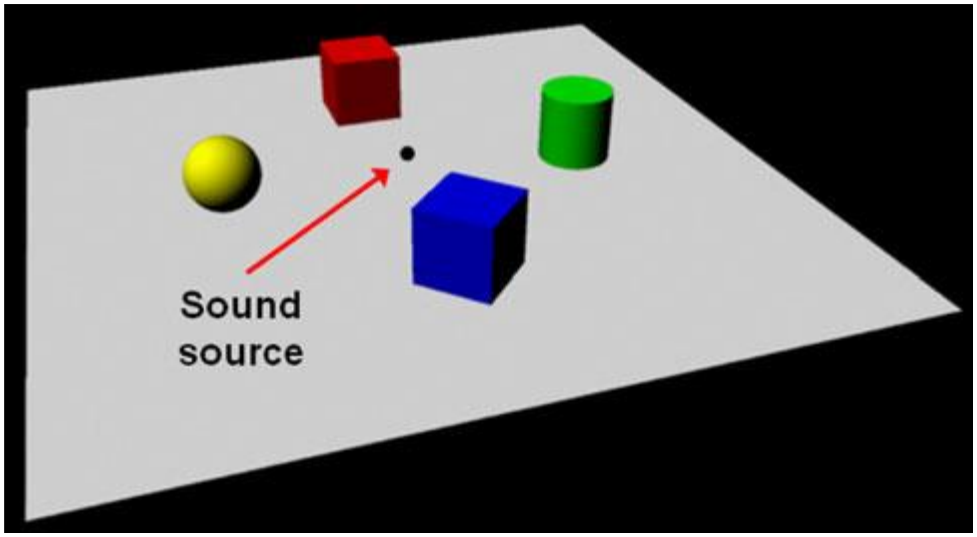


Figure 4.6: A simple scene consisting of four objects (two cubes, a sphere, and a cylinder) and a sound source

For each scenario and method tested, once the occlusion map was constructed, stage two (receiver scaling) was performed for every position in the scene, leading to the graphical output illustrated in Figure 4.7a ("base-case approach") and Figure 4.7b ("proposed method"). As with the simulation described in the previous section, sound level (power) is coded in shades of gray where white denotes full sound level, black complete silence, and shades of gray represent levels in-between. In this example, the far clipping plane was set equal to the distance of the listener thus objects behind the listener were not considered. Attenuation effects introduced by both the air and distance traveled were completely ignored for purposes of illustration (so that occlusion effects are not confused with attenuation resulting from sound propagating through the air and with distance). As shown in Figure 4.7a, with ray-casting of the base-case method, diffraction effects were not accounted for and thus, if the direct line-of-sight between the sound source and receiver is occluded, the receiver will be in complete silence despite the fact that in our natural environment, sound will reach the receiver via diffraction. In contrast,

as shown in Figure 4.7b, although the sound level was reduced, the sound still reached the shadow regions despite the occlusion of the direct line-of-sight between the sound source and the receiver, better representing real-world sound propagation.

To illustrate the conformance of the method to real-world sound propagation models, this simulation was repeated twice. In the first case, all parameters remained the same except for the dimensions of the four obstacles which were doubled. The updated object dimensions were changed as follows: the dimensions of the two (red and blue) cubes measured 20 cubic meters, the (green) cylinder was 20 m high with a 20 m diameter, while the (yellow) sphere had a diameter of 20 m. The center position of each of these objects remained the same at 25 m from the sound source. Given the increase in obstacle size (while sound source frequency remained the same i.e., 250 Hz), less sound should be diffracted. In the second case, all parameters remained the same as the original simulation (the dimensions of each of the two (red and blue) cubes measured 10 cubic meters, the (green) cylinder was 10 m high with a 10 m diameter, while the (yellow) sphere had a diameter of 10 m) except that a high-frequency (4 kHz) sound source was used. Given the inverse relationship between diffraction and frequency, increasing the frequency of the sound source should lead to a decrease in diffraction.

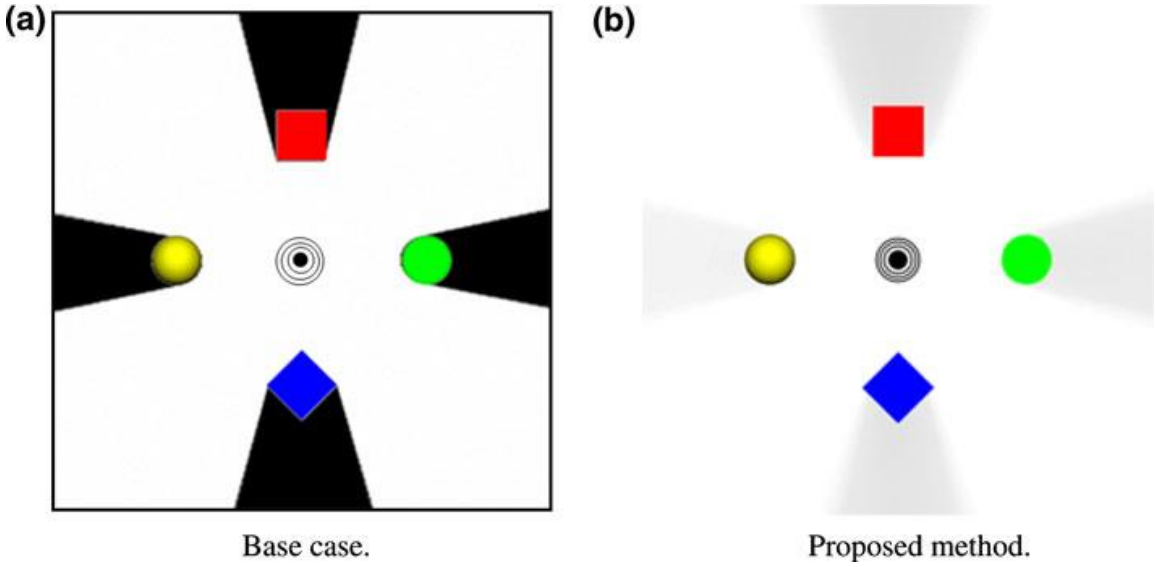


Figure 4.7: Acoustical occlusion modeling. (a) Base-case acoustical occlusion approximation. Computing occlusion effects by rendering from the sound source's perspective using ray casting (single ray) while completely ignoring all reflection phenomena. (b) Proposed method

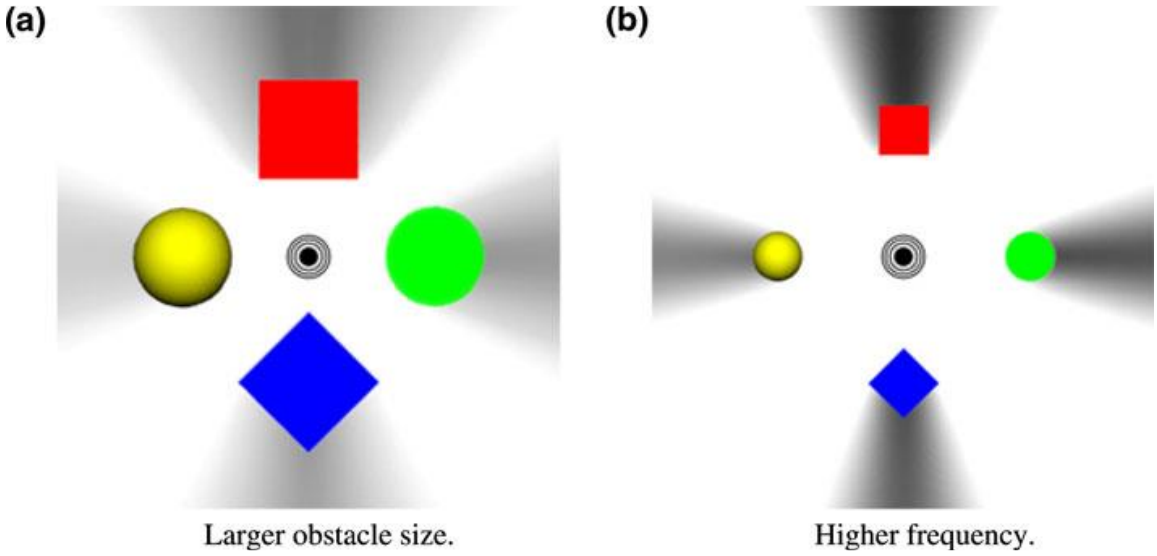


Figure 4.8: Acoustical occlusion with the proposed method. (a) Increasing (doubling) the size of the obstacles and (b) increasing (doubling) the frequency

The results of this test are illustrated in Figure 4.8. As shown in Figure 4.8a, when the size of the obstacles was increased, it is clear that less sound was diffracted into the shadow region (the shadow region is darker than the corresponding shadow region illustrated in Figure 4.7b where the dimensions of the four objects were

halved). Similarly, by increasing the frequency of the sound source, as shown in Figure 4.8b diffraction was decreased allowing less sound to reach the shadow region (the shadow region is darker than the corresponding shadow region illustrated in figure 4.7b where a low-frequency (250 Hz) sound was used).

4.4 Applying Acoustical Materials as Texture Maps

As previously described, each object has two (floating point) values associated with it: i) a reflectivity coefficient that denotes the amount of sound that is reflected, and ii) an occlusion coefficient that denotes the fraction of sound that is blocked by the object (i.e., the fraction of sound that is permitted to pass/go through the object). These values are sent to the "shader" as uniform variables implying that the values are uniform throughout the "mesh" that represents the object. A single mesh could be used to represent a wide variety of objects (e.g., a character, tree, building, etc.). However, many objects are constructed from a variety of materials that interact with sound in different ways (i.e., the reflectivity of an object can vary across the object's surface) thus it can be limiting to associate uniform values for these two parameters across an object when in fact, parts of a single object may have different properties. For example, consider a tree with its trunk, branches, and leaves. Sound may be able to pass through the tree's leafs/branches but may not be able to pass through the tree trunk. However, modeling the intricacies of sound passing through a tree can be difficult and computationally heavy. One solution is to incorporate a volumetric rendering approach whereby the 3D scene is divided into voxels (volumetric elements), each voxel with individual reflectivity and occlusion coefficients. Although with such an approach it may be possible to model the

propagation of sound in the presence of complex objects/structures and provide a high level of accuracy, it will lead to large memory and computational requirements making it impractical for real-time, dynamic environments.

With respect to the graphics/visuals of an environment, texture mapping whereby a texture map (a one-, two-, or three-dimensional pattern that can range from regular patterns such as stripes and checkerboards to complex patterns that characterize natural materials and may include 2D patterns obtained via an image/photograph [Angel 2009]) is mapped to the surface of an object, is a commonly used approach. The coordinates x, y, and z, specify the location of each vertex in a 3D Cartesian grid. Two additional coordinates, u, and v, specify the wrapping of texture by placing each vertex in the 2D space of an image. uv coordinates usually range between 0 and 1 (values can be higher than 1 to allow for texture repeating) and allow the uv coordinates to be independent from the image size. Texture mapping allows for the enhancement of the visual richness of the generated computer graphics images with only a relatively small increase in computation [Heckbert 1986]. Texture maps are generally used to specify properties of the object's surface such as color, specularity, or surface perturbation (e.g., bump map, and height map). With respect to 2D textures (images in particular which are commonly used), texture coordinates are stored with the model and can define how a 2D image wraps the mesh that represents the object/model. Ultimately, a 2D texture map is an image and stores three values that represent the intensity level of the red, green, and blue channels. Although

used for graphical rendering, there is no reason these three values can't be used for other purposes.

Acoustical texture mapping was introduced in [Cowan & Kapralos 2011b]. An acoustical texture is a 2D image designed to store audio properties instead of visual details which allows different components of an object to have unique sound properties. Each of the red, green, and blue color channels store a separate acoustical parameter specific to the part of the surface it corresponds too. More specifically, the green channel stores the occlusion coefficient of the material o_m and ranges from 0 to 1; 1 represents a material that does not allow any incident sound to pass through it while 0 represents material that allows all incident sound to pass through it. When the green channel is less than 1, sound is able to reach the listener by passing directly through the object; this is proportional to the channel's brightness, implying that as the value of the green channel is reduced, less sound is blocked by the object. The blue channel stores ambient occlusion value o_a that has been pre-calculated using, for example, the ray tracing option available with the Maya animation software. With a pre-calculated ambient occlusion map, the shader can determine if the point on the model being sampled in the fragment shader is occluded by other objects. The red channel stores the material reflection coefficient r_s (only specular reflections are considered here). $0 \leq r_s \leq 1$ is considered only for objects between the sound source and the listener and is used to apply a simple reverberation effect using a sound engine.

Alternatively, rather than employing acoustical texture maps, audio properties could also be applied to objects by assigning attributes to each vertex. The properties can then be linearly interpolated across the surface defined by the corresponding vertices. Attaching audio properties to the vertices has several advantages over using texture maps to store audio data. More specifically: i) most objects/models will contain fewer vertices than the pixels of a texture map, ii) "special" audio texture files do not have to be created and loaded into the program, and iii) there may be a minor efficiency gain by shifting calculations from the fragment shader to the vertex shader whereas when using acoustical texture maps, the properties need to be accessed in the fragment shader. However, despite the benefits of attaching audio properties to vertices as opposed to employing acoustical texture maps, there are two major problems that make such an approach impractical. In particular, many model formats do not allow additional information to be attached to each vertex, and not all modeling software allows attributes to be manually "painted" on to the vertices. In order to keep our method compatible with existing tools, low resolution textures were employed to store audio parameters and these parameters can be "painted" onto the model using standard modeling software such as Maya, or created using any art program such as Photoshop. The ambient occlusion portion of the texture map is shown in Figure 4.9.

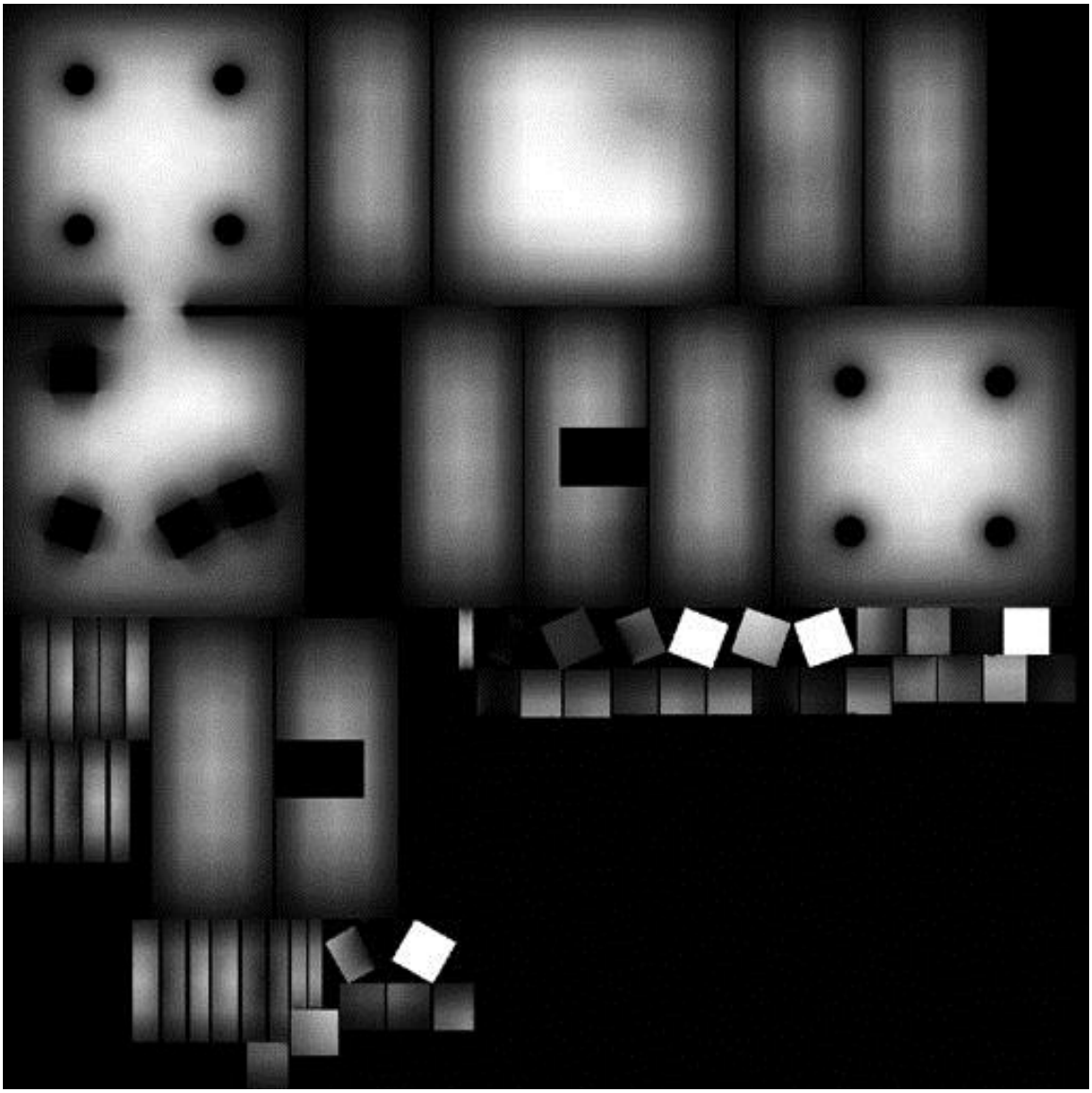


Figure 4.9: The ambient occlusion texture (only the blue channel is shown) that was applied to the test scene in order to provide the shader with information about surface occlusion.

Once the acoustical texture has been created and then loaded into the shader, a listener occlusion value is calculated based on the angle of reflection, the occlusion property of the object, and the ambient occlusion value sampled from the acoustical texture map.

$$o_s = (v_t \cdot s_n) \times o_a - (1 - o_m), \quad (4.6)$$

Where, $\boldsymbol{o}_{\mathbf{a}}$ represents the extent that the surface is blocked by other objects. When $\boldsymbol{o}_{\mathbf{a}} = 0$, the surface is completely blocked by other objects in the scene. For example, part of the floor under a box is completely hidden by the box. When $\boldsymbol{o}_{\mathbf{a}} = 1$, the surface is unoccluded, or completely exposed meaning that there are no nearby objects that are likely to block any rays reflected by this surface. $\boldsymbol{O}_{\mathbf{m}}$ represents the material's ability to block sound. Sound is able to pass directly through the object where $\boldsymbol{O}_{\mathbf{m}} = 0$. Where $\boldsymbol{O}_{\mathbf{m}} = 1$, no sound is allowed to pass through the surface. However, that does not mean that all sound is blocked, sound reflected by the surface may still reach the listener. Based on formula (4.6) the output occlusion may be negative. It is important to note that the output is a colour channel which fragment shaders automatically clamp the values of each channel to be between zero and one.

When the scene is rendered using the technique described above (i.e., emitting the "acoustical visibility rays" from the sound source towards the receiver), each pixel in the scene represents a slightly different direction. Given each acoustical ray goes through one pixel, there can be hundreds and even thousands of rays cast into the scene. Tracing each one of these rays (i.e., a complete ray tracing solution) will lead to an exponential running time; impractical for real-time applications even with the use of the GPU. With the addition of pre-calculated ambient occlusion maps, the effect of additional reflections can be approximated by taking into account the probability of a reflected ray intersecting another object. The more occluded a surface is, the more likely it is that the ray will be reflected, and the ray is likely to be reflected multiple times in heavily occluded spaces.

4.5 Simulating Reflections by Mirroring Geometry

Multiple reflections may also be accounted for by mirroring the geometry. When rendering large reflective surfaces graphically, it is common to create a reflection by rendering a mirrored version of the geometry. If one wanted to render a reflective floor in a game environment for example, the illusion by rendering everything in the room with the y-axis reversed so that the mirrored version is below the floor could be created. The floor would then be rendered partially transparent creating the illusion that we are seeing the reflection of the room on the floor's surface. This same technique could be used to account for acoustical reflections as well [Cowan & Kapralos 2009b]. An example illustrating this mirroring process is provided in Figure 4.10, where a side view of a simple room containing a sound source and a listener separated by a wall is shown. By mirroring the geometry of this room, the secondary reflection can be accounted for (Figure 4.10b). When this mirrored ray travels into the mirrored portion of the room, it contacts the mirrored version of the wall and calculates that the wall is an occluder. This is equivalent to a ray that has been reflected by the floor striking the wall as shown in Figure 4.10c.

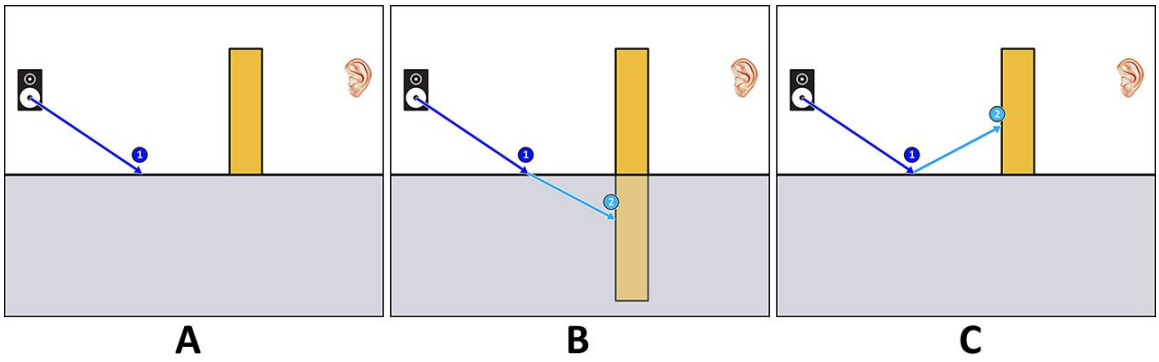


Figure 4.10: Mirrored geometry used to simulate reflections.

4.6 A Custom Shaped Clipping Volume

A modification to the occlusion method was later introduced in [Cowan & Kapralos 2013a] which reshaped the area sampled by the renderer. Previously, the method involved rendering the geometry between the sound source and listener using a view frustum with the sound located at the apex and the listener centered in the far clipping plane (Figure 4.11a). The problem with such an approach is that the occlusion factor $f_{occlusion}$, calculated for objects close to the listener, tends to be underestimated given that objects close to the listener are far from the sound source and therefore appear smaller in the rendered scene. Rendering the scene orthographically can correct this problem but will subsequently introduce new problems. More specifically, rendering a large enough view orthographically will cause the near clipping plane to intersect solid objects. Since the back facing surfaces are typically not drawn, objects that should occlude could be depicted as empty space. Ideally, the sampled area should be roughly spherical with a cone shape protrusion on either side (Figure 4.11b). One of the pointed ends would be centered on the sound source and the other would be centered on the listener. Occluding objects close to either the sound source or listener would be rendered larger and would therefore occlude more than objects that lie between the listener and sound source but are not close to either of them. It is not possible to render volumes with such custom shapes by way of the fixed function pipeline. Although such a shape could be constructed from multiply view frustums, this would require multiple renders of the geometry in the scene and would therefore be inefficient. Vertex shaders allow geometry to be reshaped by repositioning each vertex in an object. The fragment (or pixel) shader, is capable of discarding (not rendering) any

portion of an object that is calculated to be outside of any boundary that can be defined mathematically.

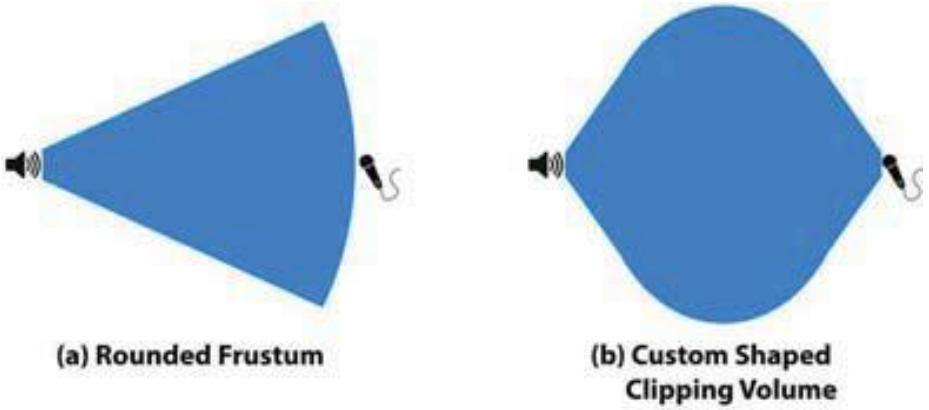


Figure 4.11: Volume of space sampled in order to render the occlusion map. (a) Original method using a frustum with a rounded bottom. (b) A custom volume which warps geometry to create curved projection.

Here a volume of space with a custom shape by warping the geometry within that space and discarding any geometry that lies outside the desired space, is rendered. The area between the sound source and the listener is rendered using an orthographic projection with the sound source located directly in the center of the near clipping plane and the listener located in the center of the far clipping plane. The width and height dimensions are both set equal to the distance between the sound source and the listener ensuring that the area sampled is in the shape of a cube. With the model-view matrix applied, the sound source is located at the origin and the listener is located at (0, 0, -width). The z coordinate is left unchanged while the x and y coordinates are scaled as follows:

$$S = \frac{1}{(\sin\left(\left(\frac{z}{d}\times 0.8+0.9\right)\times\pi\right)-a)\times\frac{1}{1-a})}, \qquad (4.7)$$

Where, $\boldsymbol{S}$ is the scale factor applied to the x and y coordinate of the vertex, z is the z coordinate of the vertex, $\boldsymbol{d}$ is the distance between the sound source and listener, and $\boldsymbol{a}$ controls the size of the resulting near and far clipping planes (informal experiments suggest $\boldsymbol{a}$ be assigned a 0.2 for good results).

4.7 Performance

As a measure of running time requirements, the scene illustrated in Figure 4.1 which consists of a listener in one room and the sound source in the adjoining room was rendered using the "acoustical texture" method and "custom clipping volume" methods described above [Cowan & Kapralos 2011b, 2013a]. The simple test environment was constructed from 1,434 triangles and was rendered as a "single model". In other words, no scene graph was used. The occlusion calculation was performed 10,000 times using a Dell XPS 720 workstation with an Intel Quad Core 2.4 GHz processor, 4 Gb RAM, Sound Blaster X-Fi sound card, and an NVIDIA GeForce GTX 280 video card (over-clocked, 1 GBGDDR3 SDRAM) which supports double precision floating point numbers. The average running time was 0.48 ms (further computational savings can be achieved by executing the method twice each second rather than every frame as was done here) which included the time required for both the acoustical ray-casting and receiver scaling stages. The running time of the method increases linearly with the number of sound sources but this shouldn't pose any problems as the number of sound sources within a typical virtual environment will generally be limited.

It should be noted that this performance test was performed in 2013 using hardware that was four years old at that time. The results could be expected to improve with the use of modern hardware.

4.8 Interactive Demo

An interactive first-person style demo that demonstrates the operation of the proposed method and provides a comparison between the proposed method and the occlusion modeling capabilities of the Fmod audio API is available online via the following website: http://faculty.uoit.ca/kapralos/OcclusionModeling. The demo consists of a simple environment with two rooms joined by an open doorway (see Figure 4.1). Each room contains occluding obstacles such as boxes and pillars. A sound source plays (continuously) a looping snippet of music. Using the keyboard's arrow keys, the "player" is free to move about the environment, sampling the effect that obstacles have on the sound they hear. Press "1" to use the proposed occlusion modeling method (the demo begins in this state), press "2" to use Fmod's occlusion modeling method. Moving the mouse rotates the camera, and pressing the "Esc" key exits the demo.

4.9 2D Sound Propagation

Most of the spatial sound methods discussed in Chapter 2, model sound propagation based on 3D geometry. In addition, the occlusion and diffraction method discussed in this chapter works by sampling 3D geometry (by rendering a 3D scene). Despite the ability of modern hardware to render highly detailed 3D environments, 2D games remain popular today [Kay 2018]. Some of the sound

propagation modeling methods and techniques discussed in chapter 2 may be relevant to 2D environments as well. For example, a graph-based system such as G-SpAR [Beig 2019] could easily be adapted to work with a 2D environment provided that, a nav-mesh could be generated from the 2D scene. However, to my knowledge, no work has addressed two-dimensional sound propagation modeling using the GPU. The method discussed here uses the GPU to efficiently model occlusion, diffraction, and attenuation by tracing the path (or paths) travelled by sound in order to reach the listener [Cowan & Kapralos 2015]. In addition, this method can be used to add a sense of hearing for non-player characters (NPCs), which may improve the realism of their behavior (artificial intelligence).

4.9.1 2D Sound Propagation Overview

The method begins by computing two two-dimensional collision maps (e.g., two images/matrices). Each location in the first map (resistance map) stores a value representing the resistance to sound propagation at a location in the environment. In other words, it represents the ability of sound to pass through the space (an "occlusion value"), whereby solid objects appears white while empty space appears black. The second map (distance map) represents a distance field whereby each location in the map stores the distance from the nearest solid object. The distance field allows the resistance map to be searched more efficiently. For example, if the sampled value returned by the distance field equates to ten pixels, then the nearest collidable object is ten pixels away. The next sample can skip over the next ten pixels because they are guaranteed to be empty, this is a process referred to as "ray-marching". Since the collision maps represent environmental properties, it doesn't

matter where the sound source(s) or the listener(s) are located. For a static environment, the collision map remains static. A sample populated two-dimensional collision mask used for preliminary testing is illustrated in Figure 4.12. Based on this collision map and the position of the listener, this method generates four "information" output maps at interactive rates (see Figure 4.13). These maps provide a lookup table for the distance travelled (the distance from the listener based on the shortest path), the direction from which sound would reach the listener, the amount of occlusion as perceived by the virtual listener as well as the perceived direction to the listener (listener direction) from every point in the environment. The data returned by the information output maps is completely independent from the location of any sound source; any location in the map could contain a sound source. The information output map lookup tables can then be used to acquire data regarding any sound source in the environment, thereby allowing for hundreds of dynamic sound sources. The distance travelled by each sound is based on the route taken by the sound in order to reach the listener thus allowing for more accurate sound attenuation modeling.

The method calculates the direction from which each sound would be heard by the listener (sound direction). For example, if the listener where located inside an anechoic chamber with only one opening, all sounds originating outside of the room would appear to be emanating from the direction of the opening regardless of their actual direction relative to the listener. Data is retrieved from the information map lookup tables based on the sound source's location. The data is then used to place a virtual sound source in an audio application programming

interface (API) such as Fmod based on the relative distance, perceived direction, and occlusion of the sound. By calculating the direction from which the listener would perceive each sound, the method is behaving similar to a “one to many” path finding algorithm implemented on the GPU. However, the method also accounts for the fact that sound may be able to travel through some obstacles such as a wire fences which would prevent NPCs from passing. The method also calculates the perceived direction and distance to the listener (listener direction) from every point in the environment. In this way, the method is able to simulate human-like perception of sounds originating at the listener’s (player’s) location for any number of NPCs. By following the perceived direction of sounds made by the player (originating at the listener’s location, NPCs are effectively performing pathfinding (locating the shortest route between two points), assuming there are no obstacles to travel that do not block sound. This method is essentially a GPU-based pathfinding technique capable of calculating the path (taken by sound) from every point in the environment to the listener, and from the listener to every point in the environment simultaneously and in real-time.

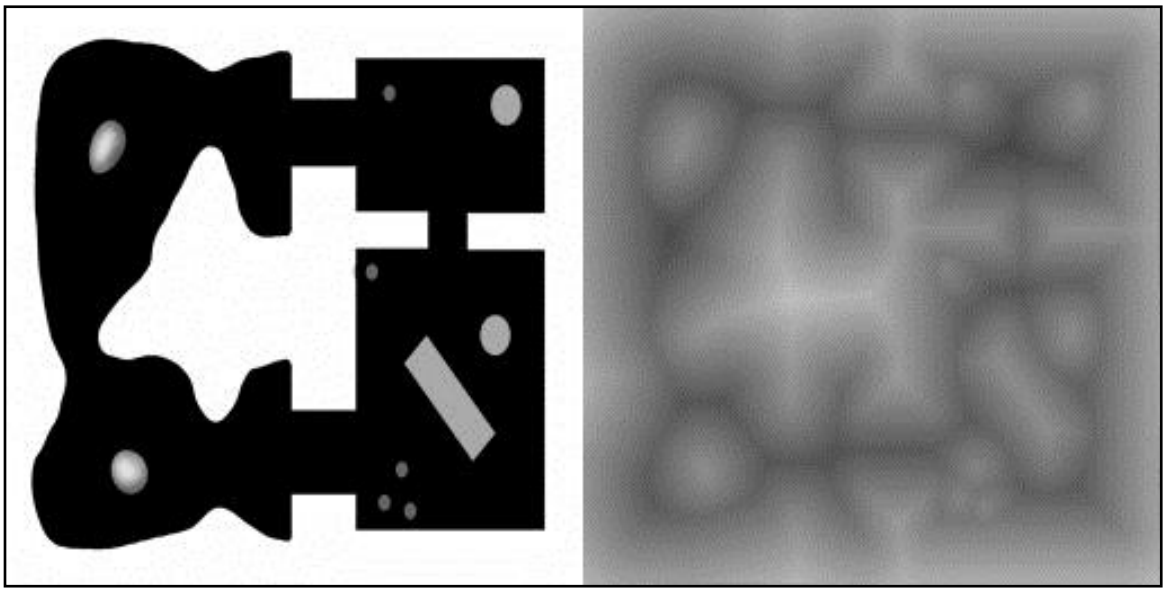


Figure 4.12: Sample input collision map. The left image illustrates the resistance map while the right image illustrates the distance map.

4.9.2 2D Sound Propagation Implementation

The objects in the environment are represented as two-dimensional resistance and distance maps (Figure 4.12). Each location (pixel) in the distance map represents the distance from the nearest surface. We employ a two-dimensional signed distance field with values ranging from -127 to 128. Negative distances represent pixels that are inside obstacles, positive values represent pixels that are outside of any object, and a distance of zero represents the surface of an object.

Acoustical occlusion is calculated in three stages: i) direct propagation, ii) indirect propagation, and iii) a distance comparison. As shown in (Figure 4.13, top-left) direct propagation fills in every area of the map that has a direct line of sight with the listener. For indirect propagation (areas that do not have a direct path to the listener), each fragment (pixel value being processed by a fragment shader), checks

the surrounding pixels to find the pixel with the shortest path. The indirect propagation algorithm is repeated many times in order to reach every part of the map. When the map has been filled, it will serve as a lookup table. The occlusion estimation is based on a comparison between the straight line distance and the path distance for each point in the path distance map (Figure 4.14).

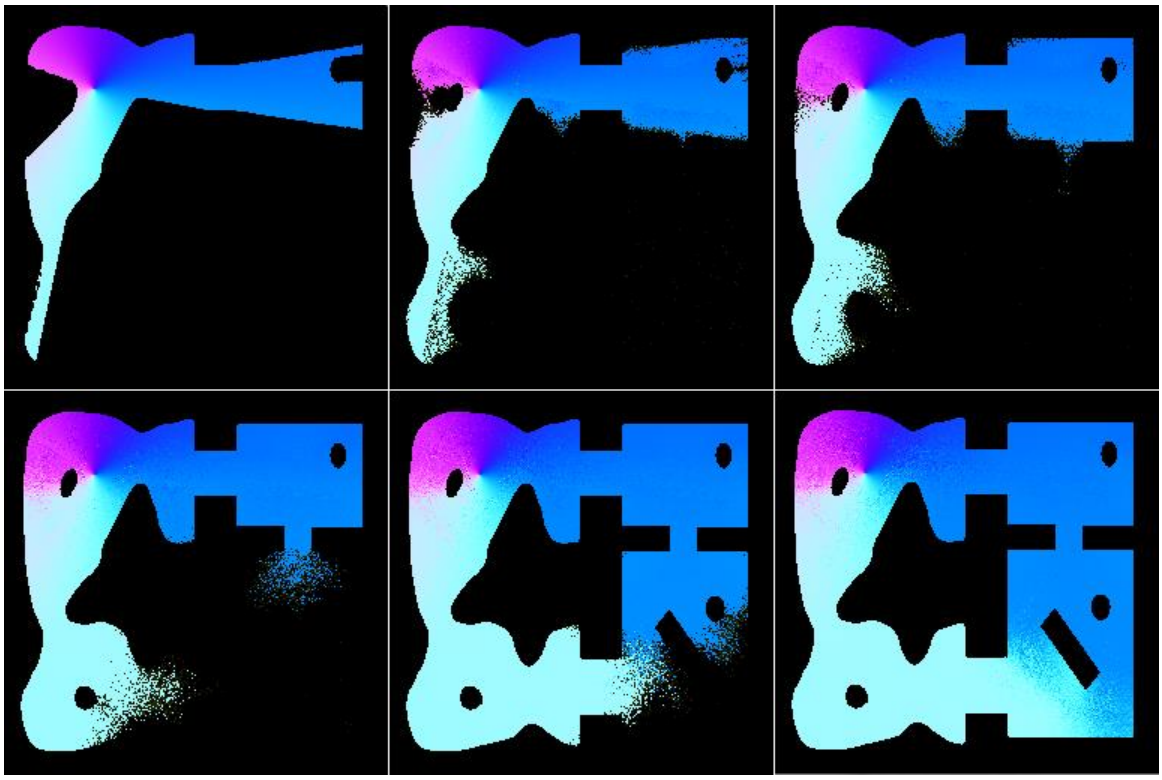


Figure 4.13: Direct propagation (top-left), Indirect propagation, 1 pass (top-center), Indirect propagation, 2 passes (top-right), Indirect propagation, 4 passes (bottom-left), Indirect propagation, 8 passes (bottom-center), Indirect propagation, 16 passes (bottom-right).

The information output map is comprised of four separate maps (images) (see Figure 4.14). The top-left image stores the direction of the sound similar to the way that a normal map stores surface direction vectors. The top-right image (Figure 4.14) stores the path distance from the listener using the red, blue and green

channels where by colors progress from cool to warm (black → blue → cyan → white → yellow → red) with black representing a distance of zero and red representing a pre-defined maximum distance. The direction to the listener allows NPCs to perceive sounds originating at the listener's location (Figure 4.14, bottom-left). The fourth image is an occlusion map which represents additional attenuation caused by occluding objects in the environment (Figure 4.14, bottom-right).

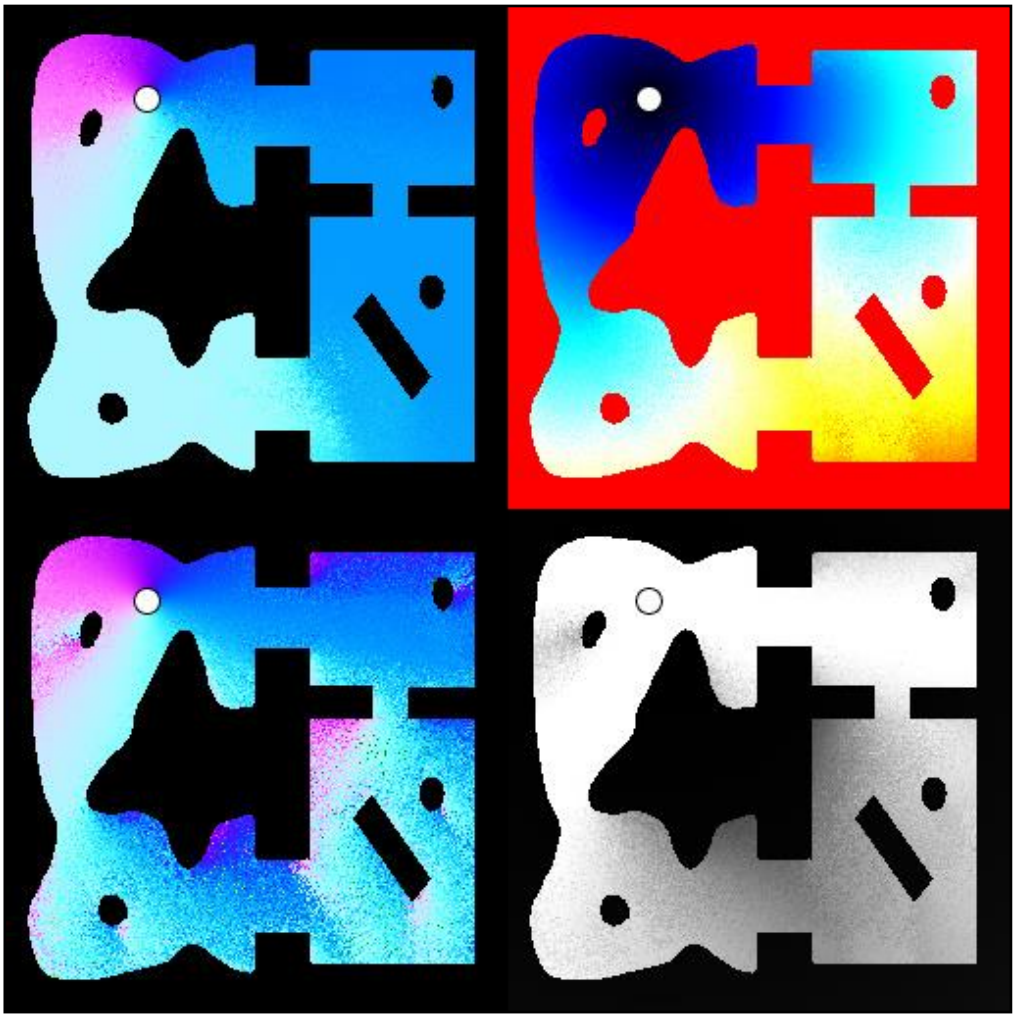


Figure 4.14: Four information output images based on the input image in Figure 4.12. The white dot in each image marks the position of the listener. The top-left image stores the sound direction vector, the top-right image stores the path distance from the listener, the bottom-left image stores the listener direction vector, and the bottom-right image stores the amount of sound occlusion present.

Direct Propagation

The static objects in the environment are used to generate a resistance map and distance field similar to the example shown in Figure 4.12. Dynamic objects in the scene are also represented by resistance maps and distance fields. First the static scene is rendered to a frame buffer with the depth of each fragment set to the inverse of the distance value supplied by the distance field. Dynamic objects are then rendered as two-dimensional sprites with their resistance maps and distance fields applied as textures. Fragments are discarded if their depth is greater than the existing depth at each fragment location. The resulting distance map contains both static and dynamic objects and can be parsed efficiently in a fragment shader in order to test for occluding objects. The input image consisting of a resistance map and distance map does not have to be updated at run-time if the environment remains static or if the dynamic objects are small enough such that their impact on sound propagation is minimal.

Calculating the direct propagation is similar to the implementation of a two-dimensional light shader with shadows (Figure 4.13, top left). For every pixel in the map, a ray is traced back to the listener's position by ray marching. The environment is converted to a two-dimensional resistance map and a distance field with each pixel storing its distance from the nearest solid object. This allows the two-dimensional ray marching function to take larger steps while sampling open areas thereby reducing the number of texture samples required by the fragment shader. Fragments that are located inside solid (occluding) objects are discarded immediately (sound cannot reach these areas). Fragments are also discarded if the

direct path between the fragment and the listener is found to be blocked (the path is blocked if any of the point samples contain occluding objects). The red and green color channels of the sound direction map collectively form a normalized two-dimensional vector. For every location within the environment, the pixel color can be used to determine the direction from which a sound would appear to be coming from if the sound was output at this location. A separate 16-bit grayscale image stores the path distance from the listener at each point in the map. After the first stage, the sound direction map stores distance and direction information for every open area that has a direct line of sight to the listener's position (Figure 4.13, top left).

Indirect Propagation (Diffraction)

The second stage of the method allows for accounting for the indirect propagation of sound through diffraction. For every fragment in the sound direction map, nearby pixels are searched. If a nearby pixel is found to contain a more up-to-date value or would lead to a shorter distance from the listener, the distance is updated using the formula in Equation 4.8.

$$d_L = d_n + d_s, \qquad (4.8)$$

where, $\boldsymbol{d_L}$ represents the path distance from the listener, $\boldsymbol{d_n}$ is the nearby pixel's path distance from the listener, and $\boldsymbol{d_s}$ is the distance between the current fragment and the neighbor (search distance). The search distance is based on the distance field, implying that fragments in open spaces will search farther than fragments in

confined spaces. Five neighbors are searched in a circular pattern around the current fragment being processed, beginning at a random angle (see Figure 4.15).

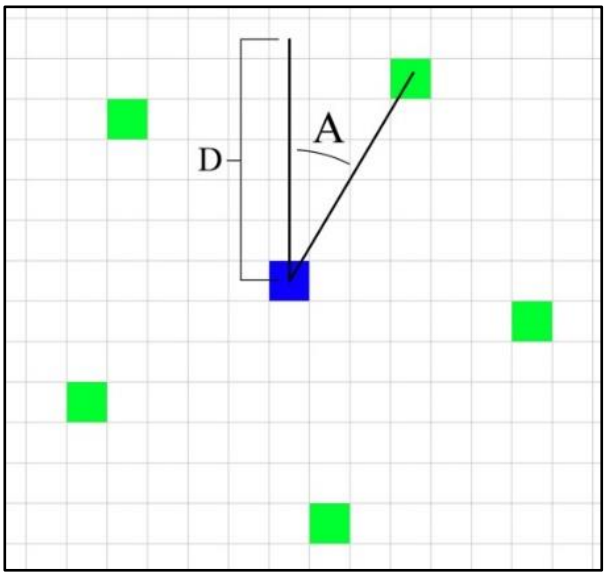


Figure 4.15: Five pixels around the current fragment being processed are searched in a circular pattern beginning at a random angle **A**, with a distance or radius $\mathbf{d_L}$ calculated from a stored value in the distance field.

The sound direction is updated to be a weighted average of the sound direction vector for the "best neighbors" (neighbors with shortest path distance). The listener direction is calculated as the direction vector from the current fragment to the best neighbor. An example is provided in Figure 4.15. Here, the listener is located in the top-left corner of the map. The majority of the environmental sounds do not have a direct path to the listener and will therefore reach the listener by passing through the opening south of (below) or east (to the right) of the listener. The direction vectors are color coded so that areas of the map where sound reaches the listener from the south are colored cyan, and areas where sound would reach the listener from the east are colored blue. Notice that in Figure 4.13, when the map has been

fully traversed by the method that the majority of the map is either blue or cyan. This is because sound originating in these areas would be perceived by the listener as having arrived through either the opening to the south or to the east.

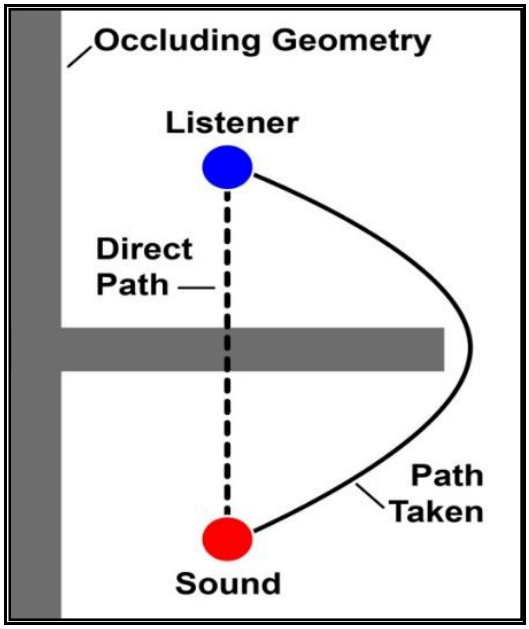


Figure 4.16: The amount of occlusion is approximated by comparing distance between the sound source and listener with the length of the path taken by the sound.

Distance Comparison

Comparing the straight line distance between the sound source and the listener with the path distance (the distance traveled by the sound), reveals whether the path taken by the sound emitted by the sound source will reach the listener directly or indirectly. The straight line distance and the distance of the path traveled are equal when there are no occluding objects directly between the sound source and the listener. If the direct path is blocked by occluding geometry, the length of the indirect path taken by the sound will be greater than the straight line distance (Figure 4.16). The amount of occlusion can be approximated as follows:

$$O = \left(\frac{d_L}{P}\right)^2, \qquad (4.9)$$

where, $\boldsymbol{O}$ represents the approximate occlusion between the sound source and listener (ranges between 0 and 1), $\boldsymbol{d_L}$ represents the direct distance between the sound source and listener ignoring all obstacles, and $\boldsymbol{P}$ represents the length of the shortest path available. As previously described, the occlusion value can be used by an audio API such as Fmod to approximate real-world acoustical occlusion/diffraction effects in real-time.

The output images shown in Figure 4.14 contains noise caused by the randomness encountered in the point sampling process. For example, when sampling neighboring pixels, it is important that the angle of rotation is random to avoid checking the same neighbors at every pass. This noise can be reduced temporally by exponential smoothing, or spatially by sampling adjacent values and finding the average.

4.9.3 2D Sound Propagation Performance

As a measure of running-time requirements, the collision mask shown in Figure 4.12 with a resolution of 256 × 256 pixels was used to generate the four output images which were then copied from the GPU to a buffer stored on the CPU. This task was completed a total of 5,000 times with the listener placed at random positions in the environment. The average running time (averaged over the 5,000 renders) was 7.6 ms (approximately 131 fps). Although for this test the method was

executed for each frame, it is not necessary to do so. Instead, the method can be executed once every few frames and the results can be interpolated between frames, thus increasing the frame rate. The tests were performed using an Alienware M14X laptop with an Intel Core i7-2670QM (2.2 GHz) Processor with 8 GB of RAM and an NVIDIA GeForce GT 555m GPU with 3 GB DDR3 dedicated video memory. This performance test was performed using a laptop computer in 2015. The results could be expected to improve with the use of modern hardware.

The 2D sound propagation method presented here is able to approximate acoustical occlusion and diffraction for large two-dimensional environments (or three-dimensional environments where most of the movement takes place on a two-dimensional plain). The method is capable of calculating the distance and direction from which sounds appear to originate based on occluding objects and structures in the environment. In addition, the direction and distance to the listener (player) is calculated for NPCs allowing for more realistic artificially intelligent opponents that use a human-like sense of sound to track the player. The method operates in real-time (interactive) rates regardless the number of sounds being output or the number of NPCs listening.

4.10 Conclusions

Acoustical diffraction is an elementary means of sound propagation in the real-world but due to the demanding computational requirements, modeling acoustical occlusion and diffraction effects is typically ignored. The graphics processing unit (GPU) is a dedicated graphics rendering device that exploits the inherent

parallelism in the feed-forward graphics pipeline to provide the very high processing rates necessary for interactive computer graphics applications. The inherent power of GPUs has been applied to various spatial sound applications including occlusion modeling. Here, the development of a GPU-based occlusion modeling method capable of approximating plausible acoustical occlusion/diffraction effects in real time for use in interactive and dynamic videogames and virtual environments was discussed. The 3D occlusion and diffraction method discussed here was implemented using the OpenGL shading language thus allowing it to be employed on commonly available programmable GPU video cards. Given the importance of diffraction with respect to sound propagation in our natural environment and the widespread availability of computer graphics cards with onboard programmable GPUs, occlusion/diffraction effects can now be easily and efficiently accounted for in videogames and virtual environments.

The occlusion values produced by the original algorithm were an approximation only and did contain flaws when compared to more accurate methods such as sonel mapping [Kapralos et al. 2008] (Figure 4.5). However, those flaws were far less apparent when listening to the output in a virtual environment. The sound behaved as the player would expect it to in the real world based on the environment. Work continued on the algorithm resulting in additional publications. The addition of acoustical textures that map audio-based parameters (e.g., specular reflection coefficient, a parameter that denotes the fraction of sound blocked by the object, and the pre-calculated ambient occlusion value) over the surface of an object,

allows these audio properties to vary across the surface of an object (in contrast to the original method where the audio properties remained constant across the entire object) ultimately providing more flexibility. With the addition of the custom shaped clipping volume, the output closely matched that of sonel mapping [Kapralos et al. 2008] while remaining computationally efficient. This extra accuracy was added with limited cost to efficiency. In addition, the algorithm was originally intended to calculate the amount of occlusion present between one sound and one listener, but it could be used also be used to approximate occlusion between any two 3D points in space. This implies that the algorithm could be used to pre-calculate node-to-node occlusion in a complex graph structure in addition to the estimation of occlusion in dynamic settings [Cowan & Kapralos 2011a].

Chapter 5 Acoustical Reverberation

In the previous chapter, acoustical occlusion and diffraction effects were approximated by rendering nearby geometry using a specially designed GLSL shader program. Inspired by the success of the occlusion algorithm, a GPU-based reverberation method was developed to provide plausible acoustic reflectivity for video games and virtual environments [Cowan & Kapralos 2011c]. Similar to the occlusion/diffraction work discussed previously, the reverberation method sampled the environment by rendering nearby geometry at low-resolution in order to produce a 2D mapping. The 2D map was filtered by a C++ program running on the CPU. The estimated values were then rendered acoustically using Fmod's real-time DSP effects. Collectively, the goal of both the acoustical occlusion/diffraction and reflectivity methods is to approximate room acoustics for dynamic and interactive virtual environments.

5.1 Overview

In this section, the implementation details of the GPU-based acoustical modeling method (i.e., the shader) are provided. The implementation is based on the OpenGL shading language (GLSL) [Rost 2006]. The input to the shader is the scene/model which can be arbitrarily complex (i.e., any number of objects). The method uses the actual scene geometry (including moving objects) to approximate reverberation effects irrespective of the environment (scene geometry). In contrast to many of the existing geometric acoustics-based approaches, the method does not require the continuous tracing of (acoustical) rays which makes such

approaches impractical due to their exponential running time requirements. Essentially, the output of the method is two values: i) the *room size* index ($\boldsymbol{rs_{index}}$) that describes the overall size of a room, and ii) the reverberation index ($\boldsymbol{reverb_{index}}$) that describes the overall room reflectivity. These two values are then used to drive the reverberation effects of the Fmod audio API. Given the popularity and widespread use of Fmod, particularly in the gaming field, it ensures the method can be easily employed with minimal effort.

Reflection of sound off of a surface is frequency dependent and therefore any acoustical modeling method should account for frequency-dependent reflection effects. For computational efficiencies, the distribution of sound frequency in a given sound source is typically approximated by a fixed number of frequency bands (typically eight) [Mehta et al. 1999]. Although the reverberation method presented here can be easily executed once for each of these eight frequency bands, being an approximation, in this work a single frequency is assumed. Accounting for multiple frequency bands can be accomplished by executing the method independently for each of the frequency bands albeit with an increase in computational resources.

Each surface in the scene is assigned an absorption coefficient ($\boldsymbol{\alpha}$) that denotes the fraction of incident sound energy absorbed by the surface ($\boldsymbol{\alpha}$ is frequency dependent although as described, frequency effects are ignored). A value of 1 implies that the surface does not reflect any sound, (i.e., perfect absorber) whereas a value of 0 implies that the surface reflects 100% of the incident sound (i.e., perfect

reflector). The fraction of sound energy that is reflected off of the surface is denoted by $\boldsymbol{r_s}$ which is calculated as follows:

$$r_s = 1 - \alpha \quad (5.1)$$

The shader stores several pieces of information within the red, green, and blue pixel/fragment channels. The red channel stores the distance to the nearest object per pixel and is determined during the rendering phase. It can be thought of as the percentage of the maximum distance considered. For example, if the maximum distance considered is 50 m, a value of 1 returned by the shader represents a surface that is 50m away, a value of 0 represents a distance of 0 m, and a value of 0.5 would correspond to a distance of 25 m. The blue channel holds the fraction of reverberation that can reach the object from the sound source (denoted by $\boldsymbol{e_r}$) and takes into account the surface's reflectivity ($\boldsymbol{r_s}$), and the surface's normal vector $\boldsymbol{N}$. Thus, a surface that reflects sound directly back to the sound source contributes a greater amount of reverberation as opposed to a surface that reflects sound away from the sound source. Mathematically, $\boldsymbol{e_r}$ is calculated (at the fragment level, and stored in the blue channel) as follows:

$$blueChannel = e_r = r_s(V_{o,s} \cdot N) \quad (5.2)$$

where $\boldsymbol{V_{o,s}}$ is the normalized vector from the sound source to the object/surface, $\boldsymbol{N}$ is the surface normal vector, and $\boldsymbol{r_s}$ is the fraction of sound energy reflected by the object. Examination of Equation 5.2 reveals that the listener's position is not accounted for at all and it is unlikely that the listener will be at the same location as the sound source. Rather (and as previously described), acoustical occlusion

effects are ignored entirely (in order to focus solely of reverberation effects), and it is assumed that the reflected sound energy (reverberation) will reach the listener via an unoccluded path (occlusion effects can be handled using the GPU-based (real-time) acoustical occlusion modeling method described in Chapter 4). The angle between the sound source and the surface is accounted for as it provides a quick measure of scene and surface complexity. The reverberation index (***reverb**$_{index}$*) is determined by calculating the average blue pixel value:

$$reverb_{index} = \frac{\sum_{i=0}^{M}\sum_{j=0}^{N} blueChannel_{i,j}}{M \times N}, \qquad (5.3)$$

where, ***M*** and ***N*** denote the number of framebuffer rows and columns respectively, and ***blueChannel**$_{i,j}$* denotes the blue channel value of framebuffer pixel ***i***, ***j*** (e.g., ***e**$_r$*). As previously described, the red channel represents the distance to the nearest object for each direction. Mathematically, this distance (denoted by ***d**$_o$*) is calculated in the fragment shader as follows:

$$redChannel = d_o = \frac{d_{s,o}}{d_{max}}, \quad (5.4)$$

where, ***d**$_{s,o}$* is the distance between the sound source and the object, and ***d**$_{max}$* is the maximum distance considered (i.e., distance between the two most distant points in the environment/scene; currently this value is manually assigned). Both ***e**$_r$* and ***d**$_o$* are computed for each “pixel" of the six renders. Algorithm 5.1 summarizes the procedure to calculate the reverberation index (***reverb**$_{index}$*).

Calculating the reverberation index ($reverb_{index}$).

```
reverb_index = 0.0


{Loop through every pixel in the rendered image and find
the average of the blue pixels}
for i = 0 to i = M × N - 1 do
      reverb_index += blueChannel[i]
end for


reverb_index = reverb_index / (M × N)
```

Algorithm 5.1: The procedure used to calculate the reverberation index ($reverb_{index}$).

The room size index (rs_{index}), is then determined by averaging the individual blue channel values (i.e., e_r that represents the fraction of reverberation that can reach the object from the sound source) and red channel values (i.e., do that represents the distance to the nearest object per pixel) values across each of the six renders. Objects close to the camera are rendered far larger than objects in the distance due to perspective. To account for this, and to ensure that objects both far and close contribute (approximately) equally, distance is cubed. This is summarized in the pseudo-code provided in Algorithm 5.2.

It is important to note that only reflective surfaces contribute to the room size calculation. For example, if one were standing in an open area with a reflective wall 50 m away (assuming the ground was not reflective), all of the sound reflections (echoes) would be coming from the wall. Here, the average room size would be the distance from the wall (50 m). The ground (floor) is not taken into account because it would absorb sound and not contribute to the echogram. So a surface that reflects

50% would be weighted at 0.5 when calculating the room size (the room size output is a float between 0 and 1 where 1 is the maximum distance considered).

Calculating the room size index (rs_{index})

```
distance[256]
roomSize_index = 0.0
weight = 0.0

{Sort the pixels by distance and assign a weight factor
based on the distance and reflectivity}
for i = 0 to i = M × N - 1 do
      distance[redChannel[i] × 256] += blueChannel[i] ×
      (redChannel[i])^3
end for

{Find the average distance by weight}
for i = 0 to i = 255 do
      roomSize_index += (i/256.0) × distance[i]
      weight += distance[i]
end for

roomSize_index = roomSize_index / weight
```

Algorithm 5.2: Calculating the room size index (rs_{index}).

The reverberation index and room size index are then passed to Fmod's reverberation effect ("FMOD DSP TYPE REVERB") using the following Fmod function calls:

dspReverb → setParameter(FMOD_DSP_REVERB_ROOMSIZE, rs_{index});

dspReverb → setParameter(FMOD_DSP_REVERB_WETMIX, $reverb_{index}$);

5.1.1 Graphical Illustration

Here, the detailed results of two simulations that demonstrate the operation of the reverberation method are presented. The first simulation demonstrates the operation of the method graphically while the second provides an indication of the computational running time requirements. For illustrative purposes, a distogram, which is a 2D plot of distance (x-axis) vs. sound energy (y-axis) is calculated. The distogram denotes the sound energy relative to the sound source as a function of distance and is calculated by rendering (from the sound source's perspective) the environment (using the GPU) from six different directions. Instead of the end result consisting of a “colorful picture" as in graphics-based rendering, the six renderings contain information regarding the distance between the “camera" (corresponding to the sound source) and objects in the environment (scene).

The scene illustrated in Figure 5.1 (as seen from the sound source's perspective) was rendered to approximate the reverberation effects. The scene consists of a building (constructed from 2D squares placed on different angles with different textures applied to them) and an open area and the sound source is assumed to be out in the open area and away from the building. Despite the low polygon count (1624 triangles), it still contained 812 models. This was purposely done to illustrate the operation of the method with a non-trivial environment. The surfaces of the scene were comprised of stone, metal, and grass. The absorption coefficients ($\boldsymbol{\alpha}$) of these three surfaces were arbitrarily assigned values of 0.5, 0.05, and 0.95 respectively. Included in the figure (the “red-black" top-left inset) are the six renderings of the environment that represent the shader output (red, green, and

blue channels summed together). The size of each of the six renderings was 64 × 64 pixels; informal experiments reveal that this resolution is actually greater than needed (32 × 32 or 16 × 16 will suffice) but this higher resolution provides a more detailed distogram for display.

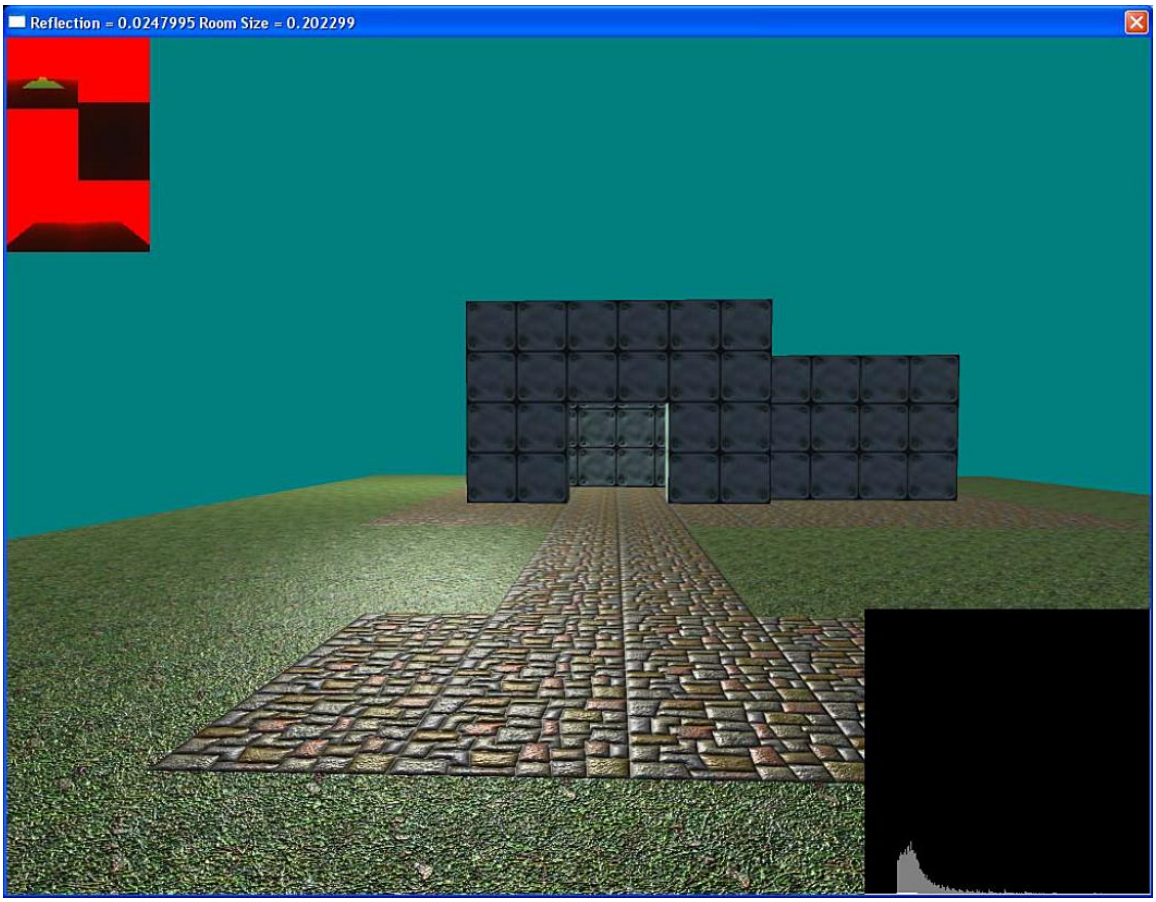


Figure 5.1: Example rendering of a simple environment. The “red-black" top left inset illustrates the shader output (red, green, and blue channels summed together). The inset on the bottom right illustrates the distogram where the x-axis represents the distance from the source/observer (the graph origin represents a distance of 0) while the y-axis represents the amount of geometry at each distance. The white bars indicate the amount of sound that would be reflected back toward the source.

In the bottom right inset of the screen capture above (Figure 5.1) is a distogram which is rendered in real-time. Notice that the first peak in the distogram is dark grey; this represents the distance to the ground under the listener's feet (the grass

does not reflect much). The dark grey areas of the distogram mark the distance of the surfaces rendered (the number of pixels in the render at each distance). The white portion of the distogram is the amount of sound reflected back from each distance (the number of pixels at each distance weighted by the surface reflectivity returned by the blue channel). As shown, the white regions are small given the fact that the primary reflections arrive from the floor only given that the building and other surfaces are far away. For example, the stone reflects 50% of the sound energy. If the sound was completely inside of a stone room (where 50% of the energy is reflected), the white bumps in the distogram would be slightly less than 50% of the height of the grey bumps (less than 50% because of the weighting effect of the surface angle).

To illustrate the difference in the resulting method output, the sound source was moved closer to the building as shown in Figure 5.2. Once again, the top-left inset illustrates the summed output of the red, green, and blue channels while the bottom-right inset illustrates the distogram. Given that the sound source is closer to the building and therefore closer to the reflecting surfaces, the overall energy levels shown in the distogram are larger and there is a second peak. The first peak in the distogram still represents the ground outside of the building and under the sound source although in this example, this ground is stone, which reflect a greater amount of sound energy in contrast to the grass of the scene illustrated in Figure 5.1. The second peak in the distogram is caused by the building itself which is almost a perfect reflector (95% reflectivity) while the third peak is due to the far wall inside of the building.

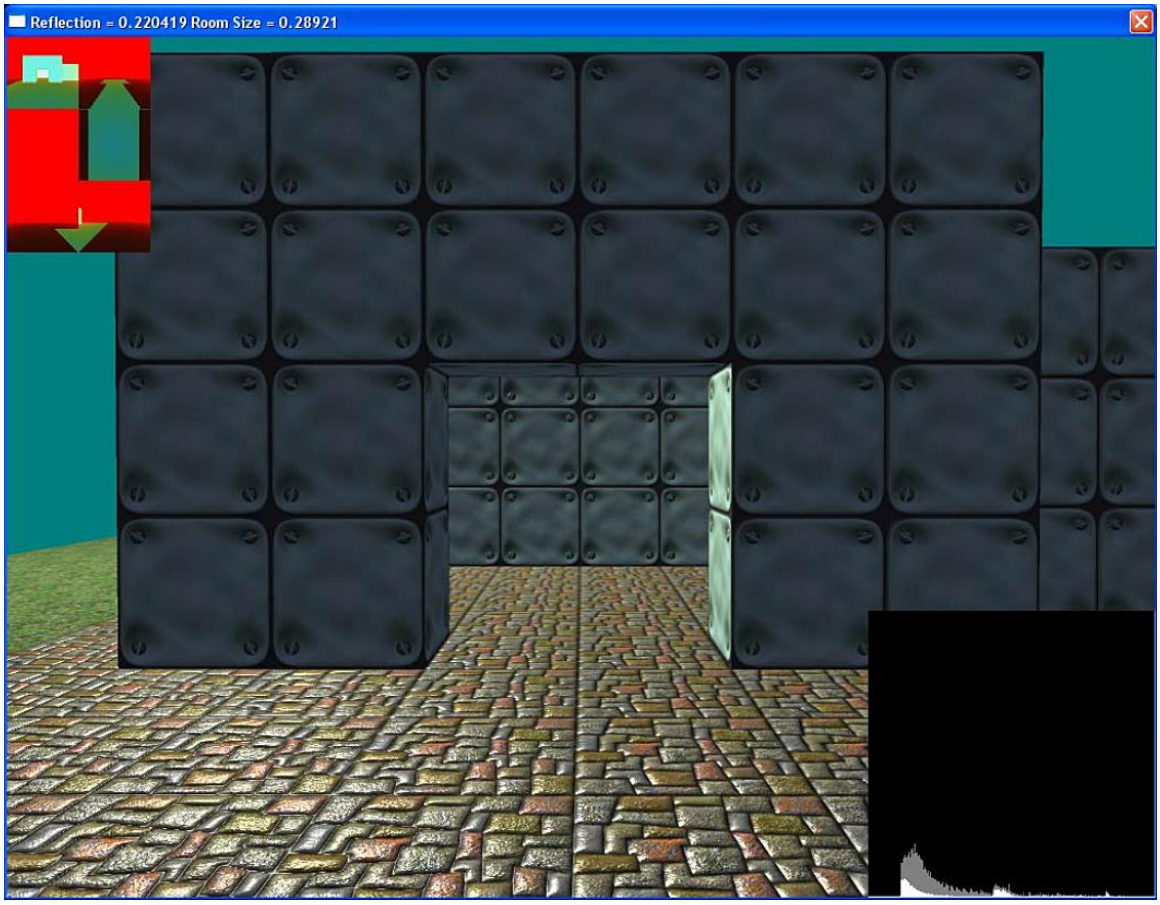
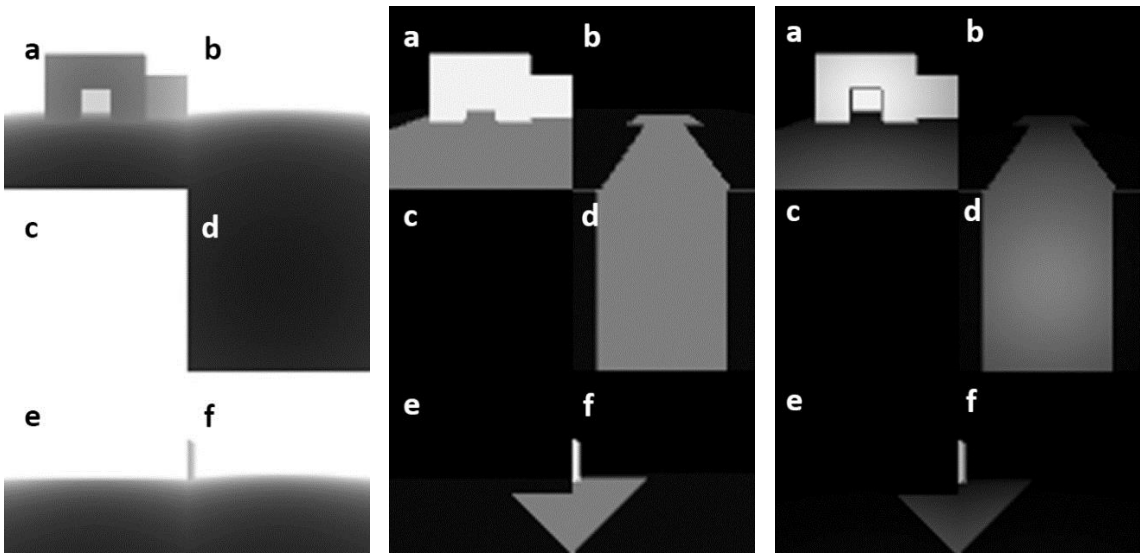


Figure 5.2: Rendering of the scene shown in Figure 5.1, but here the sound source is now much closer to the building.

A magnified version of the top-left inset (summation of the red, green, and blue channel output) of Figure 5.2 is provided in Figure 5.3. More specifically, the red channel output which represents the distance to objects in the environment relative to the sound source is shown Figure 5.3a. Figure 5.3b illustrates the output of the green channel and represents surface reflectivity, while Figure 5.3c illustrates the output of the blue channel and also represents surface reflectivity which is however calculated taking into account the angle between the sound source and the surface. The separate channels are shown in gray-scale for comparison purposes only. Values range from 0 to 1 denoting 0 to 100% brightness respectively. With respect

to the sound source (looking towards the building), the six renders represent the following: a) positive z-axis, b) negative z-axis, c) positive y-axis, d) negative y-axis, e) positive x-axis, f) negative z-axis (these views are axis aligned and therefore do not rotate with the camera).



(a) Red channel (b) Green channel (c) Blue channel

Figure 5.3: Magnified views of the red, green, and blue channel shader output of the scene illustrated in Figure 5.2. With respect to the sound source (looking towards the building), the six renders represent the following: a) positive z-axis, b) negative z-axis, c) positive y-axis, d) negative y-axis, e) positive x-axis, f) negative z-axis (these views are axis aligned and therefore do not rotate with the camera). (a) Red channel output. The red channel represents the distance to objects in the environment relative to the sound source. (b) Green channel output representing surface reflectivity. (c) Blue channel output representing surface reflectivity but taking into account the angle between the surface and the sound source.

Finally, Figure 5.4 illustrates the output of the method when the sound source is inside the building. As with Figure 5.2, the top-left inset represents the red, green, and blue channel summed output while the right-bottom inset represents the distogram. As the distogram illustrates, given the fact that the building is now much closer, there is a larger white region" in contrast to the distogram when the

sound source is outside of the building and far away (Figure 5.1) and the distogram corresponding to the sound source is closer to the building but not inside (Figure 5.2). As previously described, the white bars that appear in the distogram indicate the amount of sound that would be reflected back toward the source.

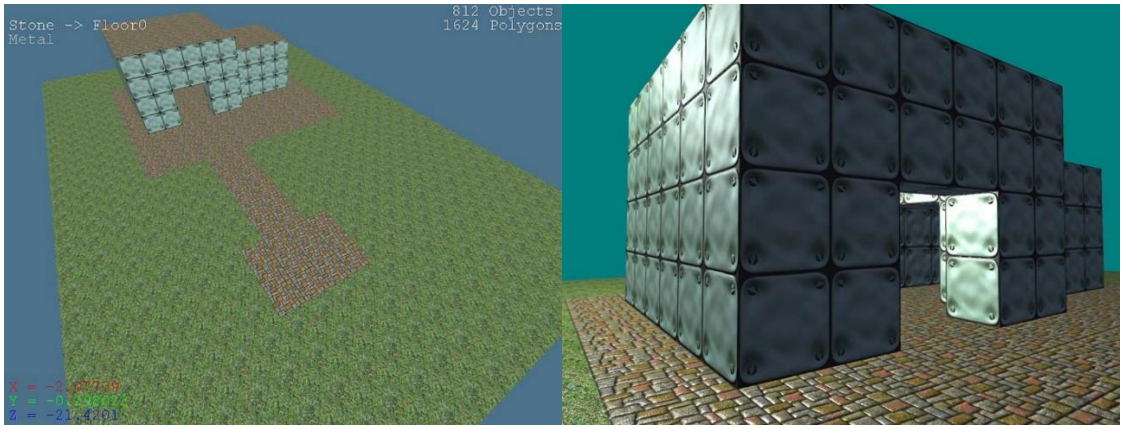
Figure 5.4: Inside the building. The top-left inset illustrates the shader output (red, green, and blue channels summed together), while the distogram appears in the bottom-right inset.

5.1.2 Running Time Requirements

As a measure of running time requirements, the scene (level) illustrated in Figure 5.5 was rendered a total of 5,000 times using a Dell XPS 720 workstation with an Intel Quad Core 2.4 GHz processor, 4 Gb RAM, Sound Blaster X-Fi sound card,

and an NVIDIA GeForce GTX 280 video card (over-clocked, 1 GBGDDR3 SDRAM) which supports double precision floating point numbers. The average running time (averaged over the 5,000 renders) to render the scene and determine the amount of reverberation, was 24.08ms (approximately 40 fps). Rendering of the scene included the time required to recalculate the scene-graph, create the six individual renders (each render of size 64 × 64 pixels) using the shader, and process the result. This also included the 0.36ms required to calculate the octree-based scene graph (16 × 16 × 1 node) and the corresponding view frustum culling six times; once for each of the six renders. The difference between the running times of each of the 5,000 renderings was insignificant and a plot of rendering vs. running time essentially yields a straight line. It should be noted that this performance test was performed in 2011. The results could be expected to improve with the use of modern hardware.

As previously described, the scene contained low poly-count objects (1624 triangles) but higher poly-count objects would only marginally effect running time because the slowest tasks are reading pixel data back from the buffer, parsing the data on the CPU, and recalculating the visibility using the scene graph. It is also important to note that the distance taken into account by our algorithm is adjustable. It is not necessary to render objects that are far away from the sound source as would be done when rendering a scene for visual display. Rendering only objects that are near the source drastically reduces the number of polygons rendered and thus computational requirements.



(a) Overview (b) Close-up

Figure 5.5: The scene (level) used to compute the running time requirements. The level was constructed from 2D squares placed on different angles with different textures applied to them. Despite the low polygon count (1624 triangles), it still contained 812 models.

5.2 Interactive Demonstration

Although the graphical examples can provide some indication regarding the operation of the reverberation method, an interactive, real-time demo whereby the user can hear the results is available online (https://faculty.uoit.ca/kapralos/ReverbModeling/). The demo simulates the environment illustrated in Figure 5.5. To avoid occlusion (i.e., an object obstructing the direct line of propagation between the sound source and listener) effects, the sound moves with the player/user. The reverberation calculations are made twice each second because the sound source is very close to the listener. Sounds that are further away do not need to be calculated as often thus reducing computational requirements. Furthermore, the demo uses the same models for calculating reverberation and for rendering the scene. This would not be the case in a videogame application where the method would use level-of-detail meshes or

simplified meshes made specifically for the sound calculations leading to more efficient computation.

5.3 Discussion

The reverberation method discussed here is capable of approximating acoustical reverberation effects for use in interactive and dynamic videogames and virtual environments. Although the method does not faithfully reconstruct acoustical reflection phenomena (i.e., it is a heuristic-based approach), it provides a plausible approximation in a computationally efficient manner. The method operates independent of the listener's position thus ignoring occlusion/diffraction effects although such effects can be handled using existing GPU-based methods [Cowan & Kapralos 2010]. Furthermore, although the assumption that the listener is in an environment similar to the environment of the sound source is made, the method could be extended to sample the environment surrounding the listener as well. The method is implemented using the OpenGL shading language thus allowing it to be employed on commonly available programmable GPU video cards. As demonstrated, the method is capable of operating at interactive rates within a complex environment making it applicable to real-time application. The bottleneck with respect to running time lies with the rendering, and not caused by the shader (reverberation calculations) which itself is very simple. In the example described in Section 5.1.2, a frame-rate of 40 fps was achieved despite the fact that the level comprising the scene illustrated in Figure 5.5 was constructed from many small “pieces" and each piece was rotated separately using its own transformation matrix. Although rendering time could be significantly reduced by simplifying the

environment, such a simplification would not be indicative of the typical modeled environments in which this method is intended to be used. In the interactive demo available online, the reverberation calculations are calculated only twice each second and the results are interpolated for the frames in between thus easily providing over 60 fps. In addition, although not considered here, sounds that are farther from the listener do not have to be calculated as often given that they contribute less to the overall reverberation. Furthermore, sounds that are stationary do not have to be reconsidered at all unless the effect of moving objects near them must be accounted for), thus leading to further computational efficiencies.

Alternatively, the reverberation method discussed here can be used to track the reflective properties of the environment around the listener only instead of each individual sound source. Each sound would be played with the same reverberation properties based only on the location of the listener. Although far more efficient, it is a noticeably less realistic approximation.

Given the importance of acoustical reflection/reverberation with respect to sound propagation in our natural environment and the widespread availability of computer graphics cards with onboard programmable GPUs, acoustical reverberation effects can now be easily and efficiently accounted for in videogames and virtual environments. Despite being an approximation which does not faithfully model acoustical reflection phenomena, the results of a number of sample simulations presented here in addition to the interactive demonstration,

indicate that the method does provide plausible and effective reverberation effects in a time-efficient manner.

5.3.1 Limitations

As previously described, the method is heuristic-based and thus only approximates acoustical reflection phenomena; it is not intended to provide a faithful reconstruction of sound propagation in an environment. Although the approximation is perceptually plausible, hearing is a subjective process and therefore, what may “sound good" to one individual may not necessarily “sound good" to another. Furthermore, there are a number of other limitations to the method. More specifically, the method renders a cube (i.e., six sides) and this may cause some directions to have more representation in the texture space. Pixels in the corners for example represent less of an angle than pixels in the center of each of the six faces. Surfaces rendered in the corner of the cube receive a higher weighting. This could be compensated for by multiplying the image by a pre-computed “compensating weighting factor" but this is not done here. Furthermore, currently the method requires a pre-defined maximum distance and this maximum distance is directly proportional to the computational/running time requirements. In addition, although the method is capable of operating at interactive rates (real-time), increasing the number of sound sources will lead to increased computation/running time requirements. Furthermore, as previously described, frequency effects are completely ignored to limit computational requirements. Finally, the method is intended to be used in dynamic environments and may be

unnecessary in static environments where the reflectivity could be pre-calculated, stored, and interpolated as required.

5.3.2 Potential Uses

The output from the shader could be used to dynamically sample a particular (virtual) environment/scene at the location of the sound source and use the corresponding output to dynamically adjust the reverberation settings in an audio API such as Fmod (e.g., “reverb" and “room size" settings). Finally, the method could also provide a quick way to pre-calculate reverberation for an entire level (stationary objects only) by sampling many points in a grid pattern and storing the result. Only a few statistics would be stored at each point and used with a sound engine. At run-time, these pre-defined values can be fetched and interpolated for any point in 3D space.

Chapter 6 The Spatial Sound Framework

Ubisoft [Guay 2012] and G-SpAR [Beig 2019] employed a nav-mesh to calculate the shortest path between the listener and the sound source. The shortest path is then parsed in order to calculate the amount of diffraction at each turn. The length of the path is used to calculate the attenuation and estimate the occlusion. Reverberation is applied based on the environment around the listener and is either pre-calculated or sampled periodically. The justification for using the shortest path only is the "precedence effect" [Zurek 1987]. Sound localisation is primarily based of the "first perceived incidence". When two or more similar sound signals reach the listener separated by a sufficiently short time delay (<1 ms), the listener will perceive them as belonging to a single auditory event. The perceived spatial location of the sound source is primarily based on the first-arriving sound siganl [Brown, Stecker, & Tollin 2015]. Direct sound will reach the listener first provided that the direct path is unoccluded. In the case that the direct path is blocked, sound may reach the listener by way of reflection and diffraction [Serafin et al. 2018]. In the case where the direct path is occluded, the precedence effect will cause the listener to perceive the direction of the sound based on the first-arriving signal.

6.1 Ambiguity

Basing the sound localisation of the shortest path works well in most cases, but it can cause problems when sound has reached the listener from different directions with a similar arrival time and intensity. Consider the diagram illustrated in Figure

6.1, where the direct path between the sound source and the listener is occluded by a wall that stretches from the floor to the ceiling. Sound is still able to reach the listener by reflecting or diffracting around both sides of the barrier. The sound will reach the listener by following two different paths; A and B. The paths are approximately the same length which means that the sound will reach the listener at the same time and with the same intensity by following either path.

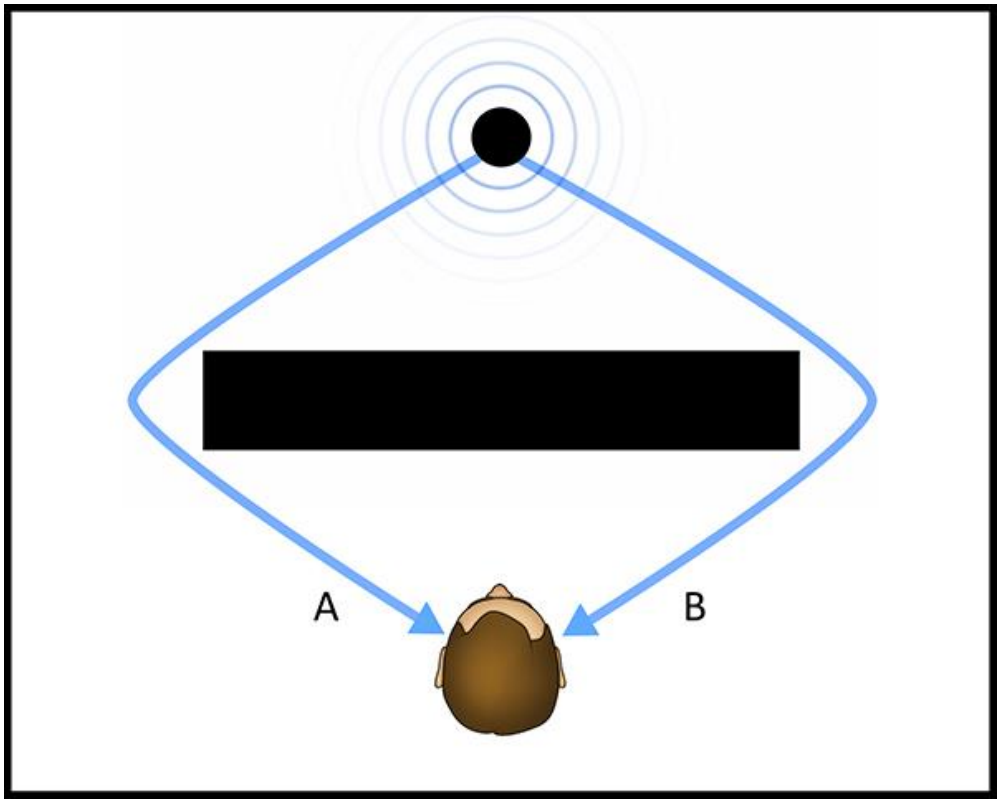


Figure 6.1: A barrier is blocking the direct sound. Sound is able to reach the listener by following path A or path B.

If the shortest path only is used for sound localisation, the listener will perceive all of the sound coming from either A or B. The listener will hear all of the sound coming from their left or right based on a miniscule difference in the calculated path lengths. The simulation may become noticeably inconsistent if the listener

were to shift to the left or right slightly. For example, if the sound localisation were based on path A and the listener moved lightly to their right, the localisation would suddenly switch to path B. Such a sharp transition will be noticed by the listener (player) and will likely have a detrimental effect on their sense of immersion. In reality, the listener pictured in Figure 6.1 will perceive the source of the sound to be directly in front of them. To calculate the perceived direction properly, all paths connecting the sound source to the listener must be explored. A weighted average could be calculated with each path assigned a weigh factor based on its length. Paths A and B in Figure 6.1 are of approximately equal length and therefore result in an equal weight. Averaging the direction of the incoming sound signals will result in the listener perceiving the sound source to be located directly in front of them. The direction should also be perceived as being somewhat ambiguous, implying that the localisation cues provided to the listener (ITD, ILD, and HRTF filtering) will not be as strong when compared to the direct sound with the barrier removed. Modelling ambiguity improves the realism of the simulation when the sound source is occluded.

6.2 Sound Propagation in Non-Geometric Environments

Most of the methods discussed in Chapter 2 base their calculations on the geometry making up the game world. Ray tracing algorithms such as Nvidia's VRWorks Audio [Nvidia 2019a] use the same geometry used to render the scene graphically (polygons) to calculate the path(s) taken by sound. Beam tracing algorithms often use a simplified (low polygon count) version of the environment, and graph based

systems such as G-SpAR use the traversable floor space for path finding [Beig 2019].

However, not all virtual environments including video games, employ 3D geometry. Despite the ability of modern hardware to render highly detailed 3D environments, 2D games remain popular today. 2D side scrolling games such as Super Mario World render the graphics as a set of 2D layers comprised of sprites. A sprite is a 2D image, often a component of a larger scene [Techopedia 2019]. They can be used to render the blocks that comprise the environment as well as the characters moving and animating within that space. For example, the rendered scene in the Super Mario World video game represents a cross section of a 3D space. Other 3D environments are depicted using beautifully drawn or hand-painted scenes. These environments do not contain polygonal shapes or models that can be used by any of the spatial sound algorithms discussed in Chapter 2. They may however still benefit from the inclusion of spatial sound. For example, the player might expect sounds to echo if the environment consists of a hand painted 3D cave or other enclosed space. In such cases, a manually plotted graph can describe the scene accurately enough to model sound propagation.

6.3 A Graph-Based Spatial Sound Framework

The framework introduced here is built on top of the Fmod sound engine. Fmod has the ability to decode many audio formats, it maintains compatibility with various operating systems and devices including game consoles, and it provides efficient DSP effects such as parametric reverberation. Given the computational

cost associated with generating a RIR filter and convolving the filter with a dry sample, this framework makes use of traditional parametric reverberation and HRTF filtering provided by the Fmod sound engine. Fmod utilises the sound card's DSP processing where there is hardware support. In addition, efficient CPU processed DSP effects and simulated HRTF filtering are provided by Fmod if not supported in hardware.

By building on top of an existing sound engine, focus can be placed exclusively on spatial sound. An efficient GPU processed 3D graph provides the PBSL, path length attenuation, and approximates occlusion and reverberation. Virtual sound sources are placed and updated using Fmod, and the reverberation and occlusion are rendered using Fmod's DSP effects based on parameters provided by the graph.

In addition to offering an efficient alternative to geometric methods such as ray tracing or beam tracing, the framework introduced here is able to provide spatial sound for non-geometric environments and to those environments rendered from polygonal meshes or models.

6.3.1 Manually Plotting Graphs

Each node comprising the graph has a 3D location with an $\boldsymbol{x}$, $\boldsymbol{y}$, and $\boldsymbol{z}$ coordinate. The framework does not place any restriction on the range of these coordinates beyond the inherent limitations of the float data type in C++. 2D graphs can be created simply by placing all of the nodes on the same plane. The easiest way to accomplish that is to set the $\boldsymbol{z}$ coordinate to zero for every node in the graph. In

order to test the systems compatibility with 2D non-geometric environments, a separate tool was developed for manually plotting 2D graphs based on a picture of the environment. The picture can be a screen-shot, a floorplan, or any other image depicting the layout of the environment to be modelled. Consider the sample 2D environment shown in Figure 6.2.

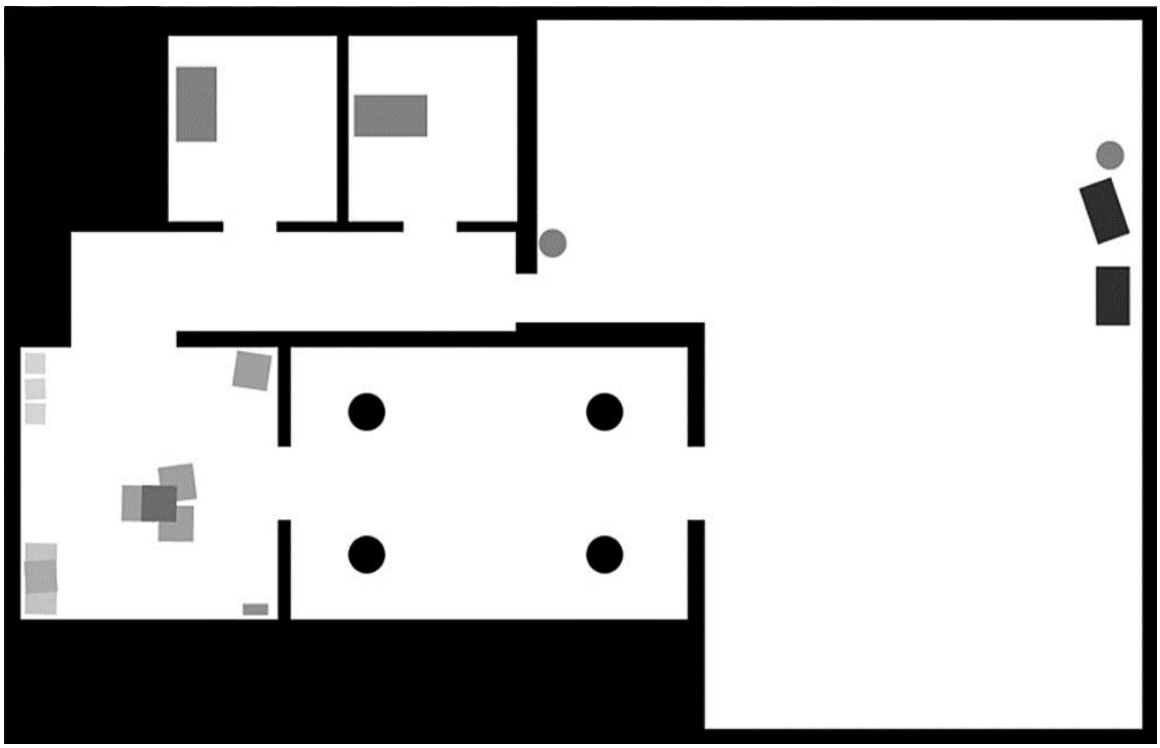


Figure 6.2: A top-down view of a simple game environment consisting of several interconnected rooms with obstacles. The shade of grey represents the amount of acoustical occlusion present. Black areas block sound and white areas allow sound to pass freely (air). Grey objects are partial occluders allowing a fraction of the sound to pass through them.

To create a graph capable of modeling sound propagation, the nodes must be placed throughout the environment representing every space that sound can reach. By connecting nodes we are informing the system that sound is able to pass between the two connected points. Once all of the nodes are connected to their

neighboring nodes, the graph describes how sound can move through the environment (see Figure 6.3).

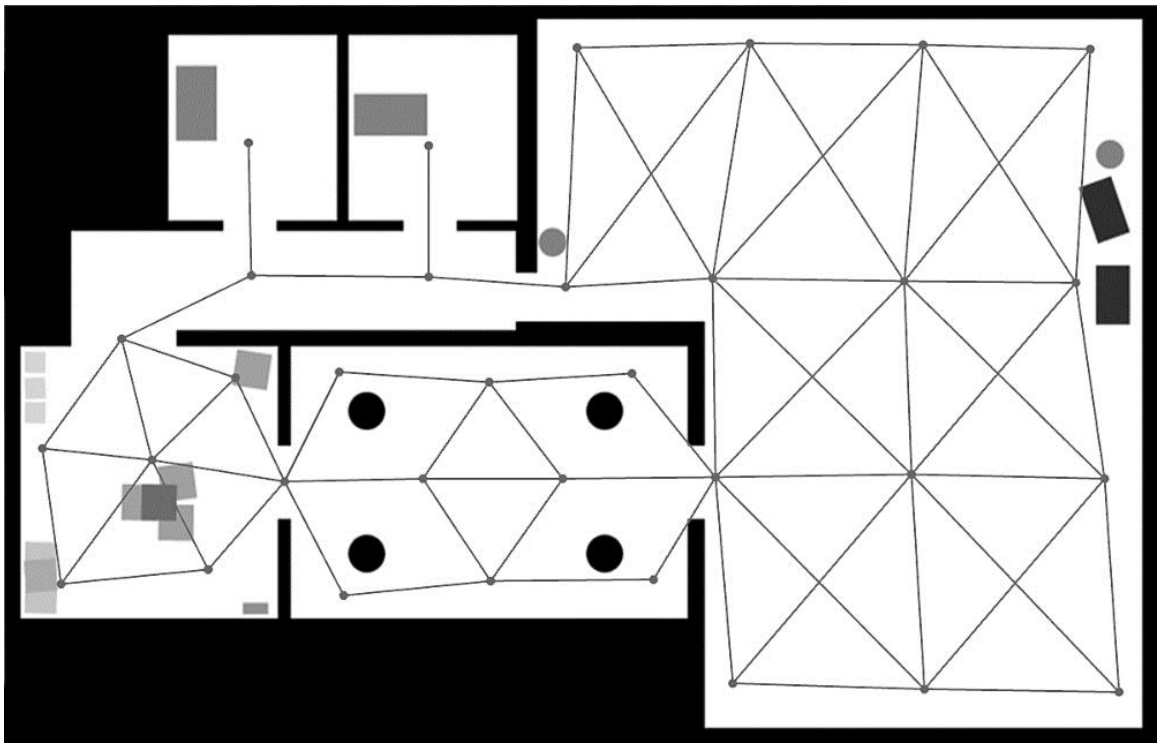


Figure 6.3: A manually created graph that describes the environment by informing the framework where sound can and cannot travel.

The manually plotted graph shown in Figure 6.3 is very sparse compared to the automatically generated graph shown in Figure 6.4. A dense graph with more nodes located closer together will improve the accuracy of the simulation at a cost of additional memory. Once the nodes are in place, each node can be assigned values representing room size and reflectivity. Room size is a value used by Fmod's DSP reverb effect. It is a floating point value between 0 and 1 and it controls the amount of delay between echoes. The reflectivity property is used by Fmod to specify the strength of the echoes. Each connection may also be assigned an occlusion value between 0 and 1. By default, the framework sets the occlusion value for all of the connections to 0 implying that there are no objects blocking the path

between the connected nodes. Occlusion can be used to specify that sound passing between two nodes would have to pass through an occluding material such as drywall or glass. Occlusion will reduce the amount of sound passing through this connection similar to adding a resistor to an electrical circuit to reduce the current flowing through the circuit. Occlusion affects the PBSL, attenuation, as well as muffing the sound using a combination of DSP effects provided by Fmod.

6.3.2 Grid-Based Graphs

Alternatively, a graph can be automatically generated in 2D or 3D by dividing the world into a discrete number of blocks. The volume of space is filled with nodes arranged in a grid pattern. Each node is then connected to the neighboring nodes as shown in Figure 6.4.

Nodes that have been placed inside solid objects such as walls, will set the occlusion value of all of their connections to 1 (100%), the highest occlusion value. This informs the system that these connections should not be used for sound propagation. In Figure 6.4, the green coloured connections have an occlusion value of 0 while the red connections have an occlusion value of 1. The yellow connections signify areas that are partially occluded by obstacles. The occlusion values for this graph were automatically generated based on the background source image. The pixels were sampled between each node and an occlusion value was set based on the lightness of the pixel with lighter pixels assumed to be open space and dark pixels assumed to be solid obstacles. Figure 6.5 shows the same graph with the source image removed.

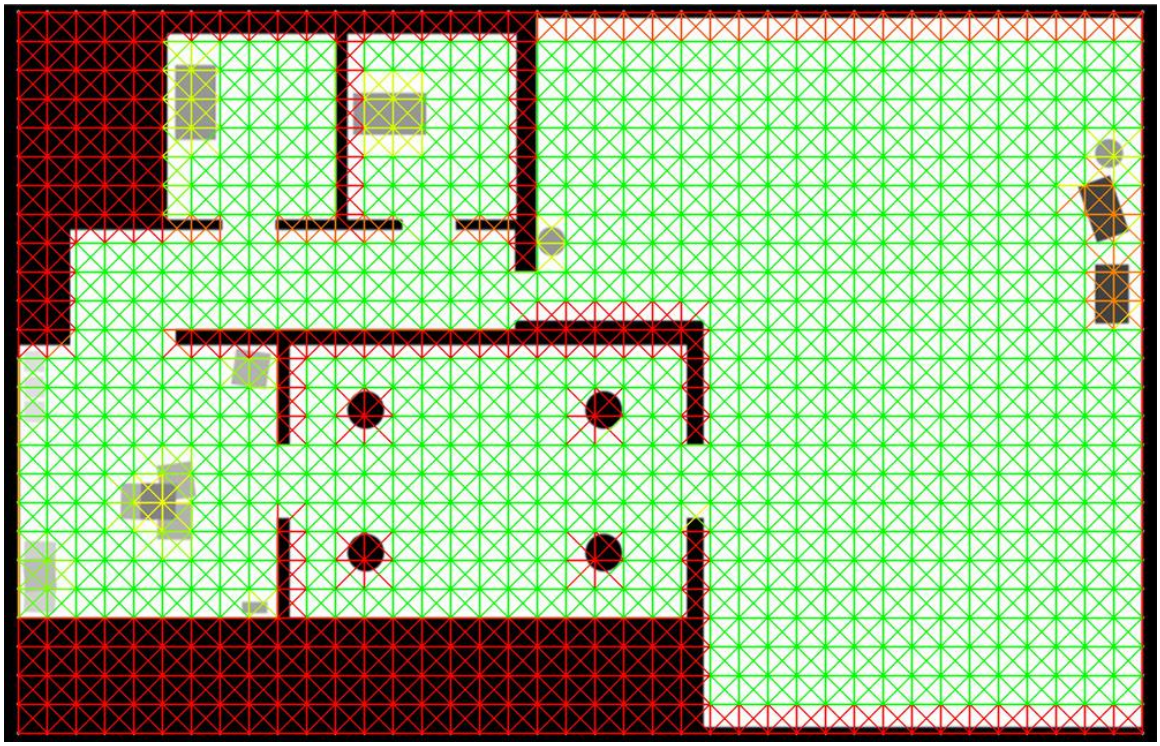


Figure 6.4: An automatically generated graph using the standard grid pattern. The connections are colour coded to show the amount of occlusion assigned to each connection. Sound is able to pass freely through green connections, but blocked completely by connections shown in red. Yellow connections allow a fraction of the sound to pass through them.

This grid representation requires additional memory and additional processing over the sparse manually plotted graph illustrated in Figure 6.3. However, the grid representation can easily be updated simply by altering the occlusion value of the connections. For example, if a door were to close separating two of the rooms shown in Figure 6.4, the structure of the graph does not have to change. The connections in contact with the door could instead be updated with occlusion values representing the door's ability to block sound. Moving objects such as vehicles can also be modelled dynamically. A bus for example, is large enough that it should have a noticeable effect on sound propagation. For collision purposes we may treat the bus as a 3D rectangular box. It is possible to calculate which nodes are inside the box and add an occlusion value to each of their connections. By

storing a list of the affected nodes, it is possible to reset the occlusion from those nodes after the bus has moved on and they are no longer in contact with the bus. The advantage of a grid or voxel representation is that every node can be mapped to a location, and every location can be mapped back to its nearest node. With manually plotted graphs and optimised graphs, there is no direct mapping of locations to their nearest nodes forcing the framework to perform a search to find the nearest node, and this is less efficient.

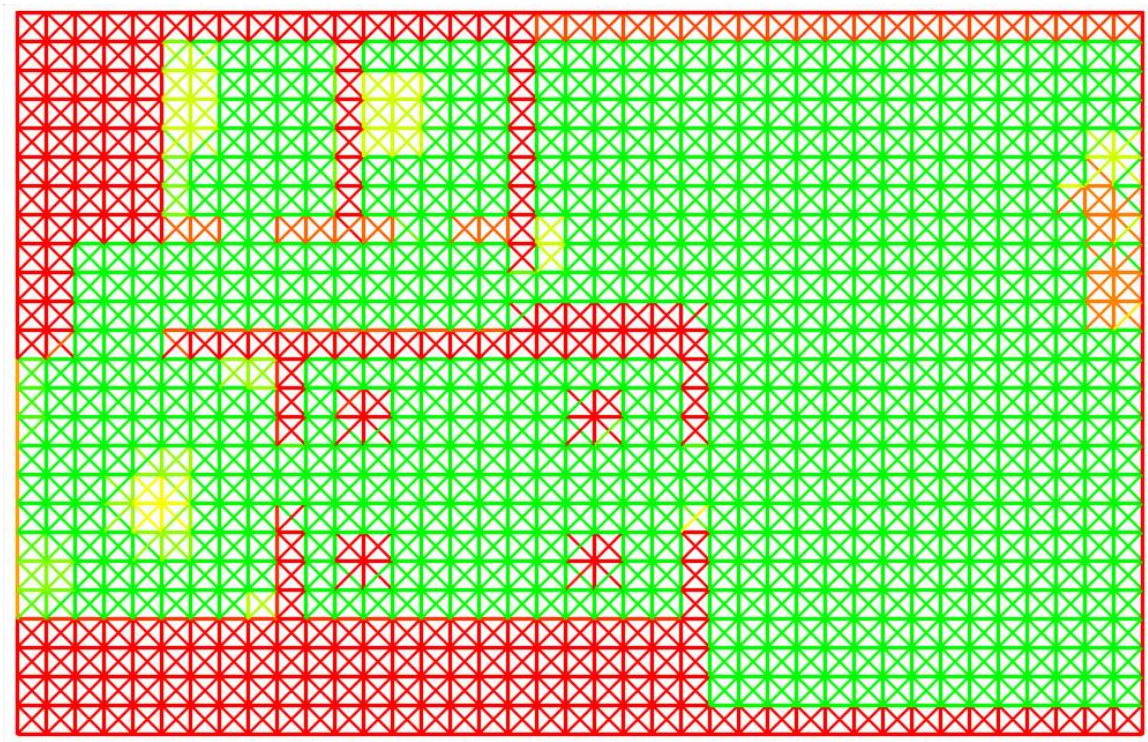


Figure 6.5: The framework models sound propagation based on a graph because graphs are far more efficient to process than images or 3D models.

A 3D graph organised as a grid can be generated from any type of 3D geometry provided that the user (sound designer) is able to provide the framework with a function that calculates the amount of occlusion between any two points. This can be accomplished by ray marching, ray tracing, or by using the occlusion method

described in Chapter 4. The user must specify the resolution or the number of nodes per meter as well as the area to be graphed. The area is specified by providing two 3D points which are opposite corners of an axis aligned 3D rectangular box. Based on this information, the framework can automatically create a grid of nodes and connect them. The occlusion function loops through every connection of every node and assigns an occlusion value. This can be done while the virtual environment (game) is loading or as an offline process.

6.3.3 Graph Optimization

Graphs can be automatically optimised by the framework if the grid representation is not needed. This is accomplished by first removing connections with a very high occlusion value. Following this, every node without any connections is deleted. The number of nodes and connections is further reduced by merging nodes that are close together until no further nodes can be merged. Two nodes can be merged only if they meet the following criteria:

- The merger will not result is any connections longer than the maximum length.
- The merger will not result in any connections having an occlusion value higher than some pre-defined maximum value.

The optimisation algorithm loops through all of the connections belonging to each node and merges nodes that share a connection if they meet the criteria above. Figure 6.6 shows the steps taken to merge two neighboring nodes.

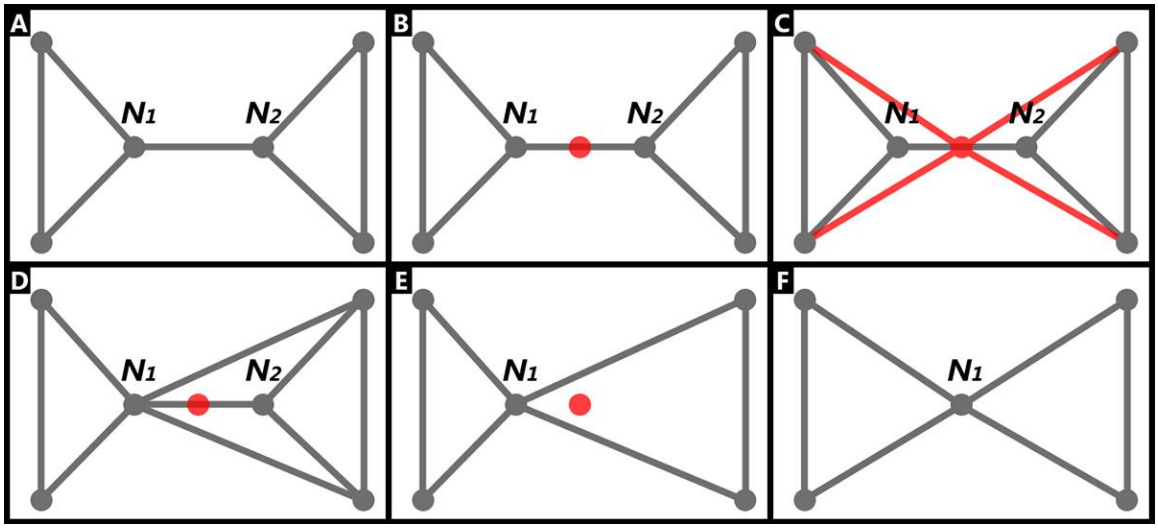


Figure 6.6: The step-by-step process of merging two points in a graph. **(A)** The graph is attempting to merge nodes N_1 and N_2. **(B)** A merge point shown in red has been calculated and placed between the nodes. **(C)** The new connections are tested for length and occlusion. **(D)** Node N_2's connections have been added to N_1. **(E)** Node N_1 and all of its connections have been deleted. **(F)** The position of N_1 has been updated.

Figure 6.6a shows part of a graph to be optimised. The optimisation algorithm is attempting to merge the nodes labeled ***N₁*** and ***N₂***. The first step in the process is to compute a merge point between the two nodes (the red dot shown in Figure 6.6b). If the merger is successful, one of the nodes will shift to this new location and the other will be deleted. The next step is to test whether all of the nodes connected to the nodes being merged are able to connect to the merge point. Nodes ***N₁*** and ***N₂*** will be merged only if all of the new connections are less than the maximum length and less than the maximum occlusion (Figure 6.6c). All of ***N₂***'s connections are added to ***N₁*** (Figure 6.6d) and then ***N₂*** and all of its connections are deleted (Figure 6.6e). Finally, ***N₁*** is moved to the merge point (Figure 6.6f).

During the optimization process each node keeps track of the number of times it has been merged. This can be thought of as a weight whereby nodes gain weight as

they absorb other nodes. If two nodes have an equal weight, they will merge to a point midway between them. When the node weights are different, the two nodes merge to a point that is closer to the heavier node. The merge point can be calculated using Equation 6.1 below.

$$P_m = P_1 + (P_2 - P_1) \times \frac{w_1}{w_1 + w_2}, \qquad (6.1)$$

where, $\boldsymbol{P_m}$ is the 3D point where the two nodes will merge if the conditions are met. $\boldsymbol{P_1}$ and $\boldsymbol{P_2}$ are the positions of nodes $\boldsymbol{N_1}$ and $\boldsymbol{N_2}$ respectively, and $\boldsymbol{w_1}$ and $\boldsymbol{w_2}$ are the weights of the two nodes. Without the weight system, too many nodes will be merged in large open areas leaving them without graph representation. Figure 6.7 shows an optimized graph that began as a 2D grid.

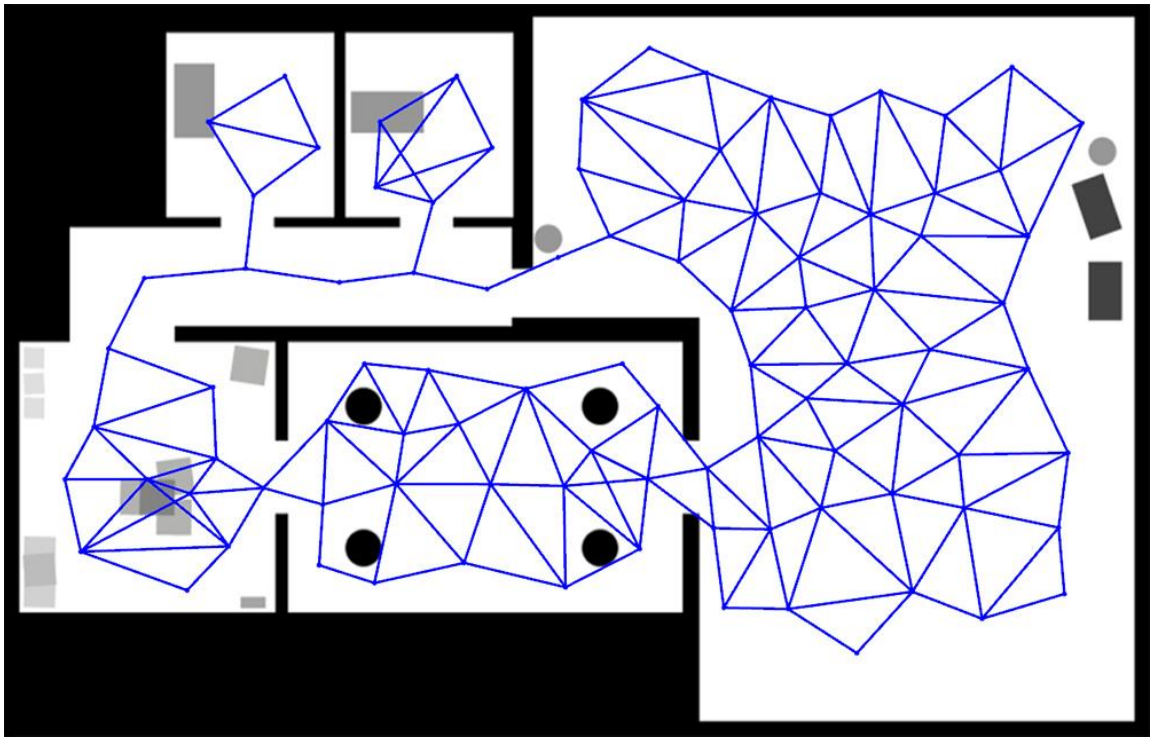


Figure 6.7: This graph was automatically generated as a grid and then optimized by iteratively merging nodes.

Graph optimisation for large graphs with tens of thousands of nodes can be a slow process lasting a few seconds (processing time depends heavily on the occlusion function supplied by the user, the number of nodes, and the system hardware). Graph optimisation is computed based on the static geometry only, and is generally an offline process. The resulting graph can be saved and loaded efficiently by the framework (a proprietary binary file type with a .dat file extension is used).

6.3.4 Finding Paths

Path finding is a processor intensive task that usually involves finding the most efficient route between any two nodes in the graph. In a virtual environment and in video games in particular, the most efficient route is not necessarily the shortest one. Other factors such as the roughness, or type of terrain may be taken into account. Likewise, road navigation software used to find the best route between any two points on a road map usually factors in the speed limit of each road in order to optimise the travel time [Xiao et al. 2012]. Sound is primarily attenuated by occlusion and distance. To simulate spatial sound using a graph, the amount of sound energy that is able to reach the listener as well as the direction that it travelled from must be calculated. However, virtual environments may contain dozens of sounds playing simultaneously. Some of those sounds are likely to be dynamic, attached to objects or characters in the virtual world. To perform a pathfinding operation for each sound is wasteful and inefficient. In most cases, there are a small number of human listeners compared to the number of sounds. VR headsets are generally limited to one headset per system. Multiplayer VR is generally handled by networking computers so that each headset is connected to a

separate computer. Each computer is responsible for running the simulation or game and rendering the graphics and sound for only one user. Rather than finding a path or paths by starting at each sound source and tracing the path to the listener, it is more efficient to work backwards starting at the listener. A variant of Dijkstra's algorithm is used to find the amount of sound energy reaching every node in the graph. Dijkstra's algorithm generally stops searching once the destination node is discovered. This variant of the algorithm does not have a destination, instead it continues searching until every node in the graph has been discovered. Once the search is complete, every node will store the estimated amount of sound energy able to reach that location and the direction.

Rather than store the path length and occlusion separately for each node, the amount of attenuation caused by occlusion is factored in by increasing the distance. For example, two nodes may be 10 m apart physically but due to the occlusion factor, more than 10 m of distance may be added to the path length calculation. Exaggerated path lengths are returned for routes that are heavily occluded. The amount of attenuation between two nodes is stored as a distance which is determined by the following Equation 6.2:

$$D_a = D \times \left(1 + \frac{{O_c}^{1.5}}{4}\right), \quad (6.2)$$

where, $\boldsymbol{D_a}$ represents the amount of attenuation and is the exaggerated distance between two nodes due to occlusion. $\boldsymbol{D}$ is the physical distance between the nodes, and $\boldsymbol{O_c}$ is the occlusion measure stored in the connection. $\boldsymbol{O_c}$ is stored as a single byte ranging in value between 0 and 255. When the occlusion is 0, the distance is

equal to the distance between the two nodes. Adding occlusion extends the distance up to 1,018 times the physical distance between the two nodes, making it very unlikely that any sound will pass through the connection.

Calculating many paths to a single listener leads to redundancy as many of the paths will overlap. By starting at the listener's position, we are effectively treating the listener as a sound source and measuring the sound energy at each node. In this simulation, the amount of sound energy lost along the path is not direction dependent meaning that the same amount of energy will be lost if sound travelled from point A to point B as would be lost travelling from point B to point A. This may not always hold true in the real world, but it is a consequence of using graphs to model (approximate) sound propagation. The direction information stored in each node represents the direction that the listener would perceive sound to be coming from, if the sound source were located at this node.

Each time the "Update" function is called, the listener is connected to the nearest node in the graph (Figure 6.8). If the graph is formatted as a grid, the nearest node can be calculated directly from the listener's position. The first step to finding the closest node to a 3D point is to convert the point from world space to grid space. The coordinate system used for a grid-based graph is integer based and always begins at the origin (0, 0, 0). All nodes reside on the positive $\boldsymbol{x}$, $\boldsymbol{y}$, and $\boldsymbol{z}$ axes and are spaced exactly 1 unit apart (0, 1, 2, 3, …n). Equation 6.3 is used to convert a world space vector to grid space.

$$P_g = \frac{P_w - O}{S_p}, \quad (6.3)$$

where, $\boldsymbol{P_w}$ is the point in world space and $\boldsymbol{P_g}$ is the point in grid space. $\boldsymbol{O}$ is a vector storing the offset or location of the origin in world space, and $\boldsymbol{S_p}$ is a vector storing the distance between each node in world space (node spacing). The grid space coordinate can then be converted to an identification number (ID) using Equation 6.4 which provides direct access to the node and all of its properties.

$$N_{ID} = z + y \times S_z + x \times S_y \times S_z, \quad (6.4)$$

where, $\boldsymbol{N_{ID}}$ is the ID number of the closest node. $\boldsymbol{x}$, $\boldsymbol{y}$, and $\boldsymbol{z}$ are the Cartesian coordinates of grid point $\boldsymbol{P_g}$,, and $\boldsymbol{S}$ a vector specifying the size of the 3D array of nodes. Alternatively, if the graph has been manually created or has been optimised, then an A* search is performed starting at the previous nearest node to the listener.

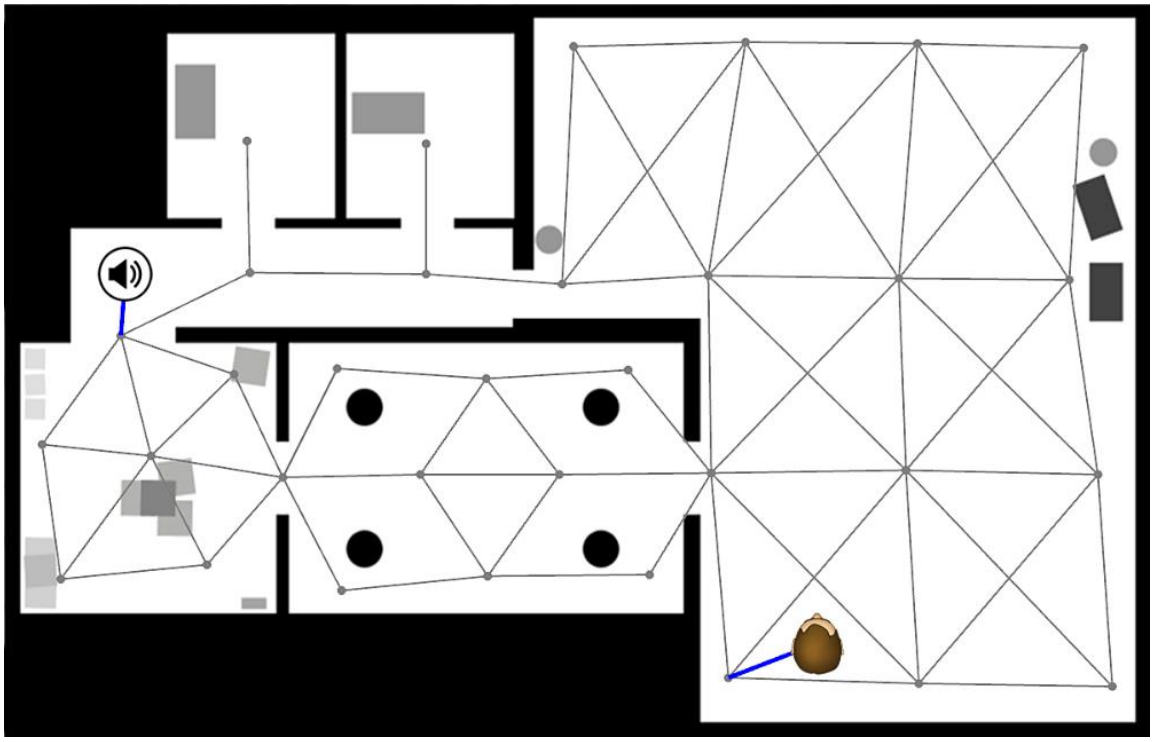


Figure 6.8: The environment has been described by the creation of a manually plotted graph. The listener (depicted by a top-down view of a person's head) is connected to the closest node in the graph. A sound source is also shown connected to its closest node.

Once the nearest node to the listener is found, the listener is connected to every node that the nearest node is connected to (Figure 6.9).

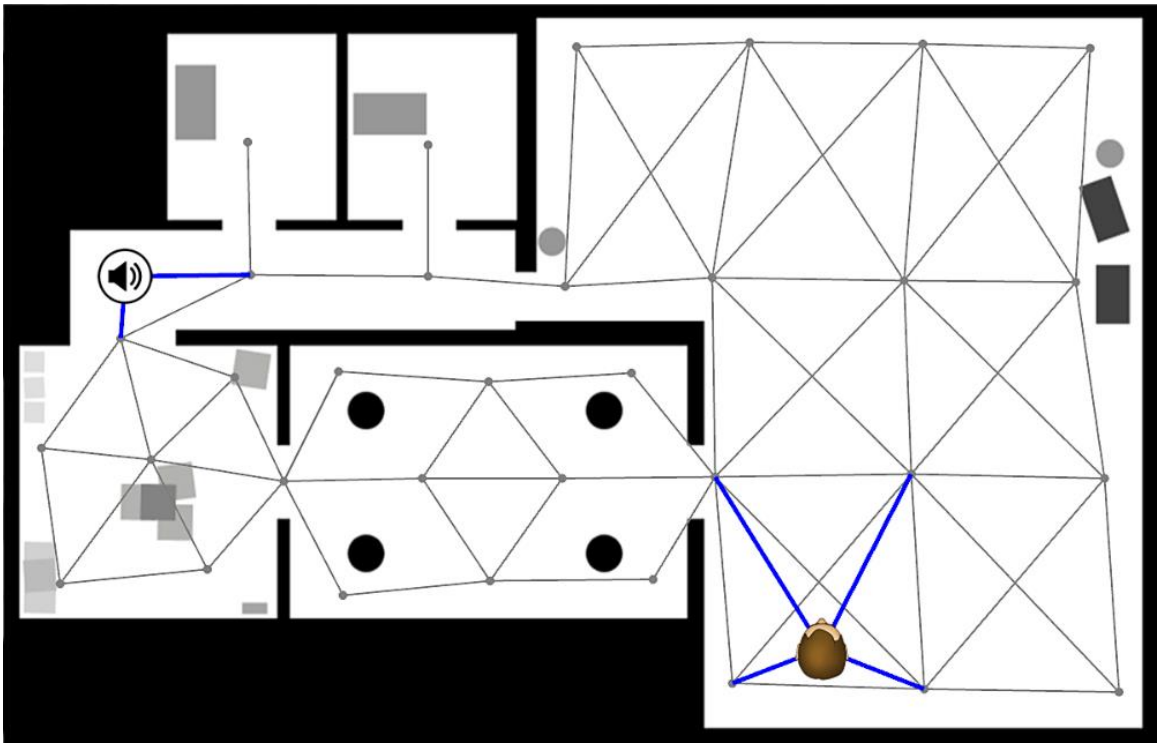


Figure 6.9: The listener is now directly connected to all of the nodes that the closest node was connected to. Each sound source will only connect to neighboring nodes if that connection will result in a shorter path to the listener.

These nodes are considered to have a direct connection to the listener. The attenuation assigned to these directly connected nodes will be based on the direct distance between the node and the listener (Equation 6.5)

$$N_d = \|N_p - L_p\|, \qquad (6.5)$$

where, $\boldsymbol{N_d}$ is the node's distance from the listener and $\boldsymbol{N_p}$ is the node's position, and $\boldsymbol{L_p}$ is the listener's current position. The direction stored in each node represents the "best path" to the listener from this point. The best path is the shortest least occluded path. The direction vector for the directly connected nodes is a normalised vector starting at the listener's position and pointing toward the node (Equation 6.6).

$$\widehat{V_n} = \frac{N_p - L_p}{N_d}, \quad (6.6)$$

where, $\boldsymbol{V_n}$ is a unit vector pointing in the direction of the connected node. This method of calculating the best path direction only applies to nodes that have a direct connection to the listener.

Distance and direction are calculated differently for nodes without a direct connection. The distance and direction for the remaining nodes are calculated in parallel using the GPU with a CUDA function. For every node in the graph that is not connected to the listener, each connection is tested to determine the shortest path to the listener (path length). This path length takes into account occlusion encountered along the way which may lead to the length of the path being exaggerated by occlusion. Figure 6.10 shows a manually plotted graph with a listener and a sound source. The nodes and connections are colour-coded to display the path length at each point in the graph. Pure yellow represents a path length of 0 m, and the colour shifts from warm to cooler colours as the length increases (yellow → orange → red → purple → blue → black).

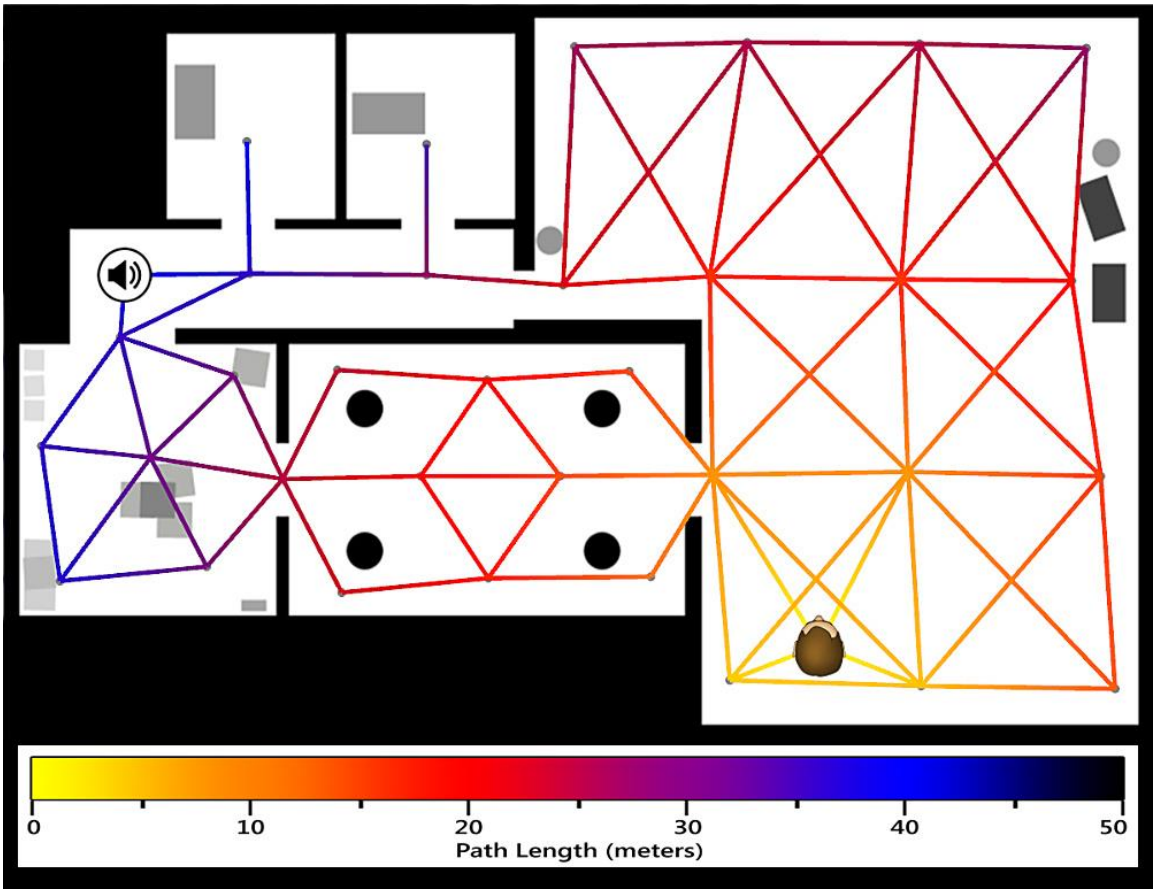


Figure 6.10: The path length of each node in the graph displayed as colours where warm colours represent short distances (yellow, orange, red) and cool represent long distance (purple, blue, black).

The perceived direction was calculated for the nodes close to the listener using Equation 6.6. This is the direction that the listener would perceive the sound to be coming from if the sound were located at one of these nodes. For the nodes not directly connected to the listener, each node checks each of its connections to determine the shortest or best path to the listener. The direction is taken from the connection with the shortest path to the listener. For example, if the listener is in a room with two openings, one to the left and one to the right, it doesn't matter where the sound is located, it will enter the room through one of the openings. The listener will hear the sound coming from their left or right depending which opening

provided the shortest path length (least resistance to sound propagation). Note that it is also possible that sound could enter the room through both openings leading to ambiguity.

Figure 6.11 shows the same setup as Figure 6.10, however, the colour-coding represents the direction instead of the distance. The colour displayed in the graph represents a 2D normalised vector where red represents the positive x-axis and green represents the positive y-axis.

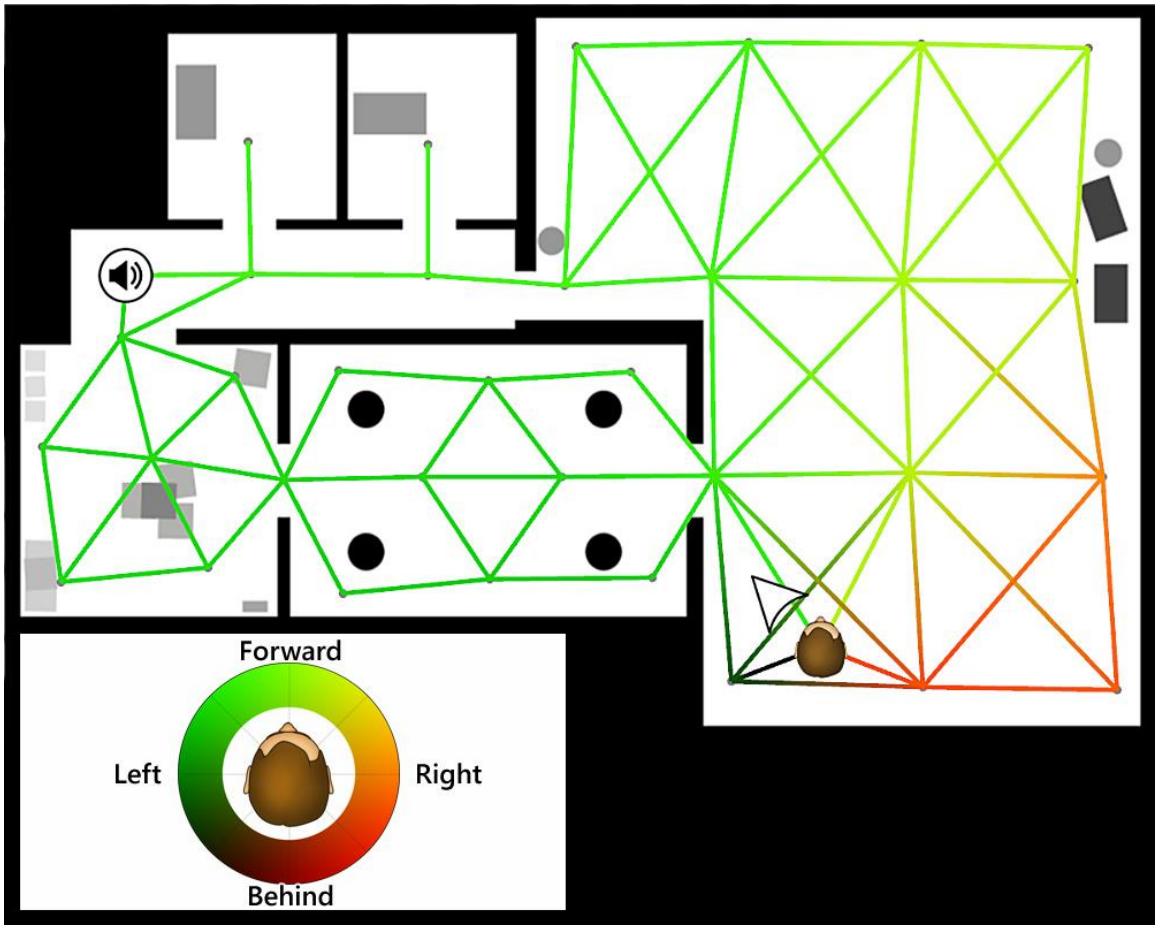


Figure 6.11: The perceived direction stored as a normalised vector where red represents the x axis and green represents the y axis. The white arrow next to the listener displays the direction that the listener would perceive the sound to be coming from.

Each node can calculate its path distance and direction based on the neighboring node with the shortest path length as described previously. However, an alternative method can be used for nodes close to the listener to improve the realism by rendering ambiguity. The distance and direction are calculated by taking a weighted average of the neighboring nodes rather than simply basing the calculation on the neighbor with the shortest path length. The distance at which the less accurate method takes over is set by the designer. Reducing the distance reduces ambiguity and therefore realism. However, removing the ambiguity may improve the player's ability to navigate efficiently as they will never be unsure of the direction that the sound is coming from.

Recalculating paths at 60 fps can also lead to redundancy as the paths wouldn't change much in 1/60th of a second. So instead of calculating every path from end to end each frame, only one hop (the length of one connection or distance between neighboring nodes) is calculated each frame for every node in the graph. Each time the algorithm's "Update" function is called (generally once per frame or 60 fps), each node in the graph surveys all of its neighbors to find the average path distance. For example, assume that the graph is a densely packed grid with the nodes spaced only one meter apart. Any change to the listener's position will immediately affect the directly connected nodes and their connections. Each time the Update function is called, the information about the change in listener position is spread by the length of one connection (1 m in this example). Assuming that the game is running at 60 fps, then information about the listener's position would travel 60 m/s through the graph. If the listener stepped behind an occluding object, the added

occlusion will immediately affect nearby sounds. However, the added occlusion will not affect a sound source located 60 m away until one second has passed (based on the example above with nodes spaced 1 m apart and the graph processed at 60 fps). It is of course possible in most types of environments to space the nodes farther apart. It is also possible to call the update function more frequently than once per frame, or simply allow the game to run at a higher frame rate (above the standard monitor refresh rate of 60 fps). In general, distant sounds will have less intensity than sounds that are close to the listener. The delay caused by the time required to spread the listener information throughout the graph is likely to go unnoticed by the player.

Since every node in the graph contains current (with a small delay based on distance from the listener), path length and perceived direction information, the graph serves as a lookup table for the sound sources. The amount of processing time required to update the graph depends only on the number of nodes in the graph and not the number of active sound sources. Each sound finds its nearest node in the graph and then surveys that node's connections to find the neighbor with the shortest path length. The distance and direction information are taken directly from the node and used to update the placement of the virtual sound source as shown in Figure 6.12.

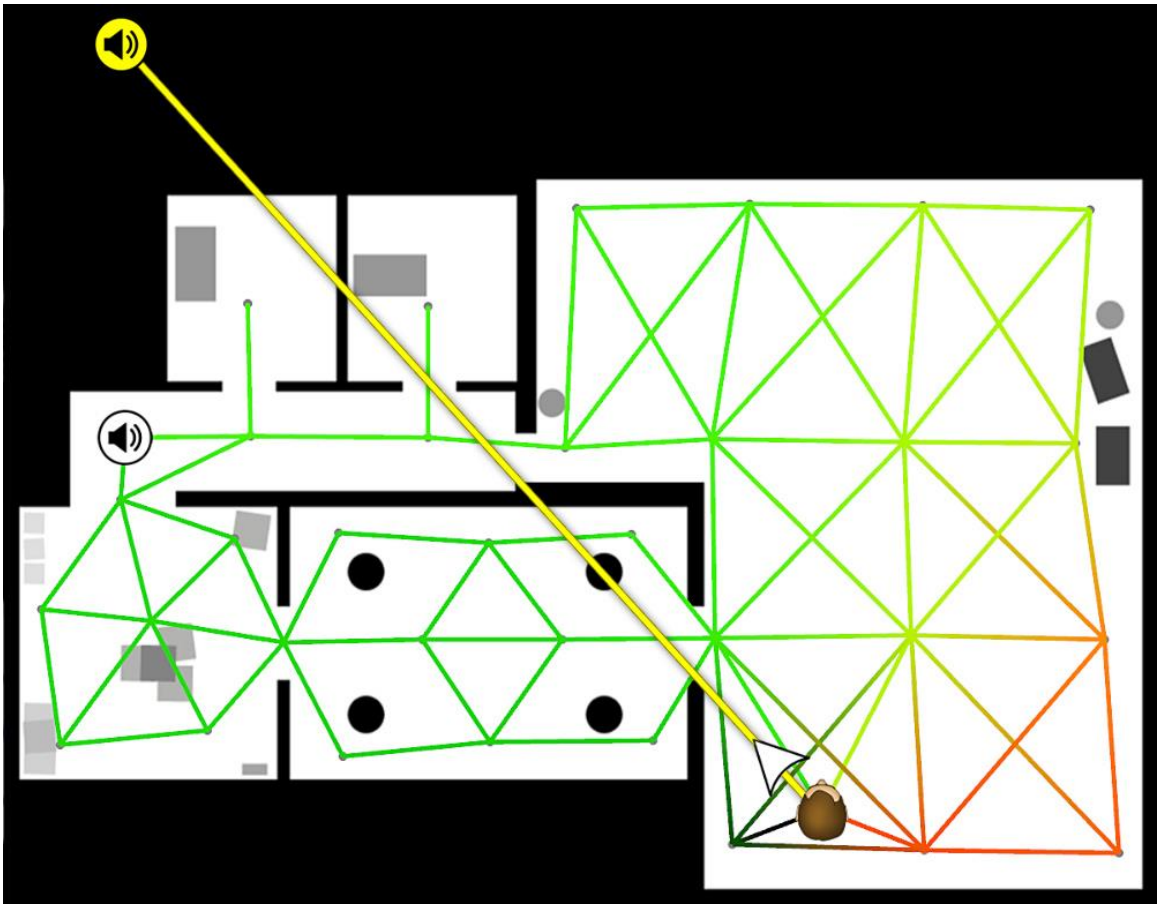


Figure 6.12: The sound source uses the graph as a lookup table to find the path length and perceived direction in order to update the placement of the virtual sound source shown in yellow.

6.3.5 Simulating Occlusion and Diffraction

The sound is played using the Fmod sound engine. Fmod is given the virtual location which it uses to calculate the attenuation, the ILD, ITD, and HRTF filtering. Attenuation is based on the path length returned by the graph. This path length can represent the actual distance that sound has travelled in order to reach the listener, or it can be lengthened in response to occlusion between the nodes making up the path. Occlusion is simulated by comparing the path length to the straight line distance between the sound source and listener. Consider the three diagrams shown in Figure 6.13. The diagram on the left shows a sound source and

listener with a clear unoccluded path between them. The path length and the straight line distance are equal which means that no occlusion is present.

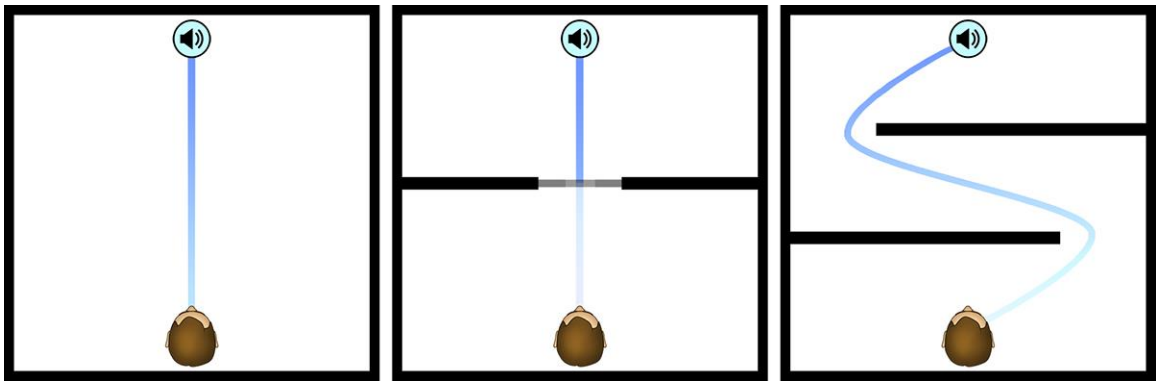


Figure 6.13: The diagram on the left shows a listener and a sound source located in the same room with no occluding objects between them. The diagram in the middle shows the effect of sound passing through an occluding object. The diagram on the right shows sound bending around walls before reaching the listener.

The diagram in the center (Figure 6.13 center) shows an occluding object such as a closed door blocking a portion of the sound from reaching the listener. To accomplish this, the connection that intersects the door would be given an occlusion value based on the door's material. The distance along this connection will be greatly exaggerated causing the path length to be significantly longer than the straight line distance. The diagram on the right shows that sound can be occluded without passing directly through an occluding object. The direct path is blocked, forcing the sound to take an indirect path in order to reach the listener. Sound is able to move around the barriers by a combination of diffraction and reflection [Serafin et al. 2018]. In both cases where the path length is longer than the straight line distance (Figure 6.13 center and right diagrams), the path will be

calculated to be occluded. The amount of occlusion is estimated using the formula below (Equation 6.7).

$$O = 1 - \left(\frac{D}{P}\right)^2, \qquad (6.7)$$

where, $\boldsymbol{O}$ represents the amount of occlusion between the sound source and listener (ranges in value between 0 and 1), $\boldsymbol{D}$ represents the direct distance between the sound source and listener ignoring all obstacles, and $\boldsymbol{P}$ represents the path length calculated by the graph. Higher occlusion values will result in more occlusion being applied to the sound using Fmod DSP effects.

When sound passes through an occluding object such as a door, the higher frequencies are generally attenuated more than the lower frequencies making up the sound. Fmod and other sound engines do implement an occlusion DSP effect which muffles sound. However, the effect is very subtle. To make occlusion more noticeable, a low-pass filter may be used in addition, or in place of the occlusion DSP. Diffraction is usually simulated using a low-pass filter as well. Since the effect of occlusion and diffraction will be simulated with the same type of filtering, it didn't make sense to calculate them separately, and will likely not improve the simulation noticeably, but it would increase the amount of data stored in each node, and the amount of data communicated between the CPU and GPU. Each node in the graph would need to store two additional floating point numbers to represent path occlusion, and diffraction. However, by computing occlusion and diffraction together, the path occlusion can be combined with the path distance negating the need for additional variables. The amount of memory used to store

the graph is negligible. The problem is that communication between the GPU and CPU represents a performance bottleneck. It often requires more time to transfer the data to the GPU and back, than it takes to perform the actual processing. Combining the effects of occlusion and diffraction was done to reduce the amount of data communicated between the CPU and GPU. Therefore, the situations depicted in Figure 6.13 middle and right diagrams, would have the same effect on the rendered sound. In both cases the occlusion and low-pass DSP filters would be applied.

6.3.6 Rendering Ambiguity

The nearest node to the listener (player) and its neighboring (connected) nodes calculate the direction vector simply by subtracting the listener's position from the node's position and normalising the result. As the direction vector propagates through the graph, each node calculates the direction at that point based on a weighted average, causing the direction vector to become un-normalized due to the linear interpolation of vectors. The length of the direction vector is reduced where sound is able to branch out in multiple directions. Therefore, at any node in the graph, the ambiguity can be measured based on the length of the direction vector. A direction vector with a length of 1 indicates that sound can only reach this point using a single path (no ambiguity). A direction vector with a length that is less than one indicates that there were multiple propagation paths leading to this point in the graph. The graph returns an ambiguity variable that ranges from between 0 and 1 which is simply one minus the length of the direction vector stored in the node that is closest to the sound source. Fmod has the ability to render sounds with

ambiguity. The direction information such as the ILD, ITD, and HRTF filtering are reduced for ambiguous sounds. An ambiguity factor of 1 causes the sound to be rendered with a clear direction, while sounds with an ambiguity close to 0 are directionless. The framework allows ambiguity to be turned down or turned off completely if ambiguity is not desirable.

6.3.7 Reverberation

A method was introduced in Chapter 5 to approximate reverberation by sampling the environment around the source of a sound. The method involved rendering the six views of a cube map with the camera positioned at the sound source [Cowan & Kapralos 2011c]. The actual polygonal geometry used to render the environment graphically, was rendered using a shader that measured the distance and reflectivity around the sound source. Reflectivity was based on the surface angle and the acoustic material assigned to each model in the scene. The resulting cube map texture was then parsed to determine the average reflectivity of the scene and the average distance traveled (room size) by the sound before being reflected. Both of these indexes were then used to drive the reverberation effects of the Fmod audio API.

The graph-based framework introduced here is also built on top of the Fmod audio API and requires the room size and reflectivity in order to simulate reverberation. Room size and reflectivity are stored in each node in the graph. The values are interpolated as the sound source moves to provide smooth transitions. By default, each node in the graph is assigned a reflectivity of zero meaning that reverberation

is turned off. The sound designer simply needs to change the variables stored in each node to enable the reverberation. In a manually generated graph, the designer can drag room properties onto each node representing different types of spaces (living room, hallway, auditorium, cave, etc.). The method introduced in Chapter 5 can be used to automatically set the reverberation parameters for each node, provided that the environment consists of polygonal meshes and the designer is able to assign acoustical properties to every object in the scene. Alternatively, the designer is free to write their own sampling function based on ray tracing or ray marching to room size and reflectivity measurements

When multiple DSP effects are applied to a sound in Fmod, the order that they are applied has an effect on the rendered sound. Each subsequent effect takes the output from the previous effect as its input. Therefore, each effect must be applied in order based on the propagation of sound starting with early influences such as the immediate environment around the sound source, then considering the path taken by the sound, and finally, listener specific effects such as HRTF filtering. The following list shows the ordering of the effects applied to each active 3D sound.

1. **Echo**: The echo DSP is applied first based on the room size and reflectivity at the sound source's location.
2. **Occlusion** (optional): The occlusion DSP effect is applied based on the amount of occlusion estimated by the graph.
3. **Low-pass**: The occlusion DSP has a minimal effect on the sound signal. A low-pass filter is applied to simulate the effects of occlusion and diffraction.

4. **Echo**: The echo DSP is applied a second time in order to simulate the many reflections that have taken place as the sound moved through the environment. This echo has a fixed room size provided by the designer, and the reflectivity is based on the path length returned by the graph (longer paths result in the reverberation becoming more pronounced).
5. **Attenuation**: The attenuation is based on the path length determined by the graph (attenuation is not a DSP effect).
6. **HRTF**: ILD, ITD, are applied as well as HRTF filtering if it is enabled. If HRTF filtering is not supported by the sound card, Fmod applies a direction dependant low-pass filter as a fall-back procedure.

Additional DSP effects may be applied to a sound by the sound designer. For example, it is common to randomly adjust the pitch of a sound each time it is played in order to add variation. Additional DSP effects must be applied first, before any of the effects relating to spatial sound listed above.

6.4 Providing AI with a Sense of Hearing

The game industry acknowledges that good artificial intelligence (AI) is a "necessary ingredient to make games more entertaining and challenging" [Bowling et al. 2006]. Yet, video game players (gamers) are generally dissatisfied with the quality of game AI, and prefer human-controlled opponents [Arrabales et al. 2013; Schaeffer 2001]. While it is generally true that visuals/graphics sells games, poor AI is something gamers routinely complain about and this in turn can negatively affect sales. The most common complaint about game AI is that it fails to behave

in a believable manner. NPCs often do not behave in realistic manner because they do not perceive the world in a realistic manner. In order for NPCs to behave realistically, their knowledge should be limited to what they could perceive by way of their five senses (hearing, sight, touch, smell, and taste). This includes prior knowledge, current sensory input, and information received through some form of communication (e.g., from other NPCs, video surveillance, and alarm ringing, amongst others). However, typically only the visual sense is simulated and this is typically limited to checking for line-of-sight between the NPC and the player's avatar using simple ray-casting approaches. The NPC's sense of hearing is often ignored, or simply distance-based without accounting for the environment. To simulate the NPC's sense of sight, it is important to test for visual occlusion (blocking caused by objects in the environment). Similarly, acoustical occlusion/diffraction effects must also be approximated in order to simulate the NPC's sense of hearing.

6.4.1 Perceiving Direction

The graph-based spatial sound framework introduced here is capable of estimating occlusion and path length attenuation for a human listener. However, this same system can also be used to estimate how well NPCs are able to hear sounds originating at the player's location (e.g., sounds made by the player such as footsteps or firing a weapon). As previously stated, the amount of sound energy lost along the path is direction independent, meaning that the same amount of energy would be lost if sound travelled from point A to point B as would be lost travelling from point B to point A. Therefore, we can use the NPC's position to determine how

well the player can hear sounds made by the NPC, and we can reuse that same path length information to determine whether or not the NPC is capable of hearing sounds made by the player. No additional processing is required by the GPU; all of the data required by the NPCs was created as a by-product of the graph-based method.

It is also possible to calculate the perceived direction to the listener (listener direction) from every point in the environment. The graph stores the direction that the listener would perceive the sound to be coming from for each node. However, the perceived direction is based on the last-leg of the journey, or the last obstacle that the sound had to pass before reaching the listener. This direction cannot simply be reversed to find the direction that an NPC would perceive sound originating at the listener's location. Instead, the NPC must find the nearest node in the graph, and use the path length of each of the neighbors to estimate the direction as shown in equation 6.8.

$$\hat{V} = \left\| \sum_{i=0}^{c-1} (P_n - P_i) \times L_i \right\|, \quad 6.8$$

where, $\boldsymbol{V}$ in a normalised vector pointing in the perceived direction of a sound originating at the listener's position. $\boldsymbol{P_n}$ represents the position of the nearest node to the NPC, and $\boldsymbol{P_i}$ represents the position of the $\boldsymbol{i^{th}}$ neighboring node. $\boldsymbol{L_i}$ is the path length stored in the $\boldsymbol{i^{th}}$ neighboring node.

In this way, the method is able to simulate humanlike perception of sounds originating at the listener's (player's) location for any number of NPCs. By following the perceived direction of sounds made by the player (originating at the player's location) NPCs are effectively performing pathfinding (locating the shortest route between two points), assuming that there are no obstacles to travel that are not obstacles to sound. The method is essentially a GPU-based pathfinding technique capable of calculating the path (taken by the sound) from every point in the environment to the player, and from the player to every point in the environment simultaneously and in real-time [Cowan & Kapralos 2015].

6.4.2 Environmental Factors

Knowing the direction that the NPC perceives the sound to be coming from and the estimated path length, may be sufficient to simulate rudimentary AI. The NPC will simply move in the opposite direction of sound propagation for a short distance before switching back to a wandering behaviour. The player would have to repeatedly make noise in order to guide the NPC to their location. That might be an acceptable method to simulate animal intelligence, but it is insufficient to simulate the thought process of a human opponent. Upon hearing a sound made by the listener, enemy units (NPCs) should estimate the location where the sound originated from, travel to that location, and then begin an organised search of the surrounding area. A NPC's ability to estimate the origin of a sound depends on many factors including, the type of environment, the characteristics of the sound, the amount of background noise present, and the individual characteristics of the listener (NPC).

Reverberation can have a negative effect on the listener's ability to locate a sound source [Zannini et al. 2011]. Reflections with a short delay and amplitudes similar to the direct signal interfere with our ability to accurately localize sound sources [Ribeiro et al. 2010]. However, the ratio of the direct to reverberant energy provides an important distance cue for sound sources that are greater than one meter from the listener [Shinn-Cunningham 2000]. In addition, reverberation may provide additional information about the environment where the sound originated, such as the size of the room and the reflectivity of the walls and other surfaces. Sounds are altered by their environment and these alterations can provide information regarding the sound's origin. A listener in a familiar environment may know which room a sound originated from based on the acoustics of the room, even if the path between the listener and sound source is indirect.

Having additional information about the environment such as the average reflectivity of the materials present, or the "room size", allows the system to estimate how useful reverberation is for sound localization. Room size is a term used by sound APIs such as Fmod. It is a value that ranges between 0 and 1, and it is used to describe how open or enclosed an environment is. For example, a room size of 1 represents an open space, while a small room size represents a small enclosed space such as a hallway or a tunnel. Fmod uses a room size estimation to simulate dynamic reverberation. A GPU-based algorithm that estimates both reflectivity and room size in real-time for use with Fmod and other sound engines is described in Chapter 5. Other environmental factors such as background noise

and acoustical occlusion can reduce the listener's ability to detect a sound, in addition to interfering with the their ability to accurately locate a sound source. All of the factors discussed above were taken into account when designing this system to approximate human hearing for artificial characters in a virtual environment.

6.4.3 Audibility

An important consideration is to determine whether the NPC is able to hear the sound at all. A NPC that does not hear a sound will not react in any way. First, the amplitude of the sound is decreased following the inverse square law (Equation 6.9).

$$A_1 = A_0 + [1 - (1 - R_s) \times R_r] \times 20 \times \log_{10} \frac{1}{D_p \times 10^{-3}},, \qquad (6.9)$$

where, $\boldsymbol{A_1}$ is the amplitude attenuated by distance, $\boldsymbol{A_o}$ is the source amplitude, $\boldsymbol{R_s}$ is the room size, $\boldsymbol{R_r}$ is the reflectivity of the environment and $\boldsymbol{D_p}$ is the distance of the shortest path between the sound source and listener. Note that the path distance ($\boldsymbol{D_p}$) is multiplied by 10^{-3} to convert from meters output by the graph, to kilometres. Using this formula, sound attenuation generally follows the inverse square law except when the environment is very enclosed such as a tunnel ($\boldsymbol{R_s}$ is close to 0) with highly reflective surfaces ($\boldsymbol{R_r}$ is close to 1). The room size $\boldsymbol{R_s}$ and reflectivity $\boldsymbol{R_r}$ are based on the room size and reflectivity properties of the nearest node to the player (the player is the sound source that NPC's are listening for).

Next, the attenuation caused by the medium (M), in this case air, must be accounted for (Equation 6.10). Here we assume M to be equal to 1 dB/km (10^{-3}

dB/m) for sounds with a frequency of 200 Hz (roughly the frequency of human voices) at sea level with a temperature of 21 degrees Celsius and 50% humidity [ISO 9613-1, 1993].

$$A_2 = A_1 - M \times D_p, \quad (6.10)$$

where, $\boldsymbol{A_2}$ is the amplitude remaining after attenuation due to distance and the medium ($\boldsymbol{M}$) has been applied. $\boldsymbol{A_1}$ is the output from equation 6.9. $\boldsymbol{Dp}$ is the path distance returned by the graph in meters.

The amplitude is reduced further by the amount of acoustical occlusion present along the path taken by the sound. The occlusion is calculated by the graph (described in section 6.3.4 Finding Paths). An occlusion value of 1 implies that the source is unoccluded, while a value of 0 implies that no sound is able to reach the listener. Equation 6.11 takes the output from Equation 6.10 and further attenuates the amplitude based on the path occlusion output by the graph.

$$A_3 = A_2 \times \left[1 - \left(1 - O_p\right) \times (1 - R_r)\right], \quad (6.11)$$

where, $\boldsymbol{A_3}$ is the resulting amplitude after factoring in the path occlusion $\boldsymbol{O_p}$. $\boldsymbol{A_2}$ is the output from Equation 6.10 and represents the amplitude remaining after attenuation due to distance and the medium have been applied. Note that the occlusion is reduced for reflective environments. Even if there is no direct sound, much of the sound energy would be added back in the form of reverberation.

Beginning with the source amplitude ($\boldsymbol{A_o}$), the three formulae above (Equations 6.9 – 6.11), reduce the amount of sound energy that is able to reach the listener. Next, the listener's ability to hear the resulting sound based on the amount of background noise as well as the hearing ability of the individual listener (NPC) is accounted for. When the effect of background noise is considered, it is important to note that the presence of a sound does not prevent a listener from hearing another sound, even if that sound has a lower intensity. However, the presence of background noise does dull our sense of hearing making other sounds appear less intense [MacPherson & Middlebrooks 2000]. Equation 6.12 is a useful approximation used to reduce the amplitude further based on the background noise level.

$$A_4 = A_3 \times \left(\frac{A_3}{\sqrt{A_3^2 + L^2}} \right), \qquad (6.12)$$

where, $\boldsymbol{A_4}$ is the perceived amplitude of the sound due to the presence of background noise level in decibels ($\boldsymbol{L}$). $\boldsymbol{A_3}$ is the attenuated and occluded amplitude from the previous steps. This formula can be used to simulate locations where a consistently load background noise is generated by something in the environment (e.g. a waterfall, music, machinery) that would impair the NPC's ability to hear. If the perceived amplitude is greater than the listener's individually assigned threshold of hearing, then the listener has heard the sound and will react to it (most likely by moving to the perceived location).

6.4.4 Localization Error

The next step is to estimate the localization error based on the intensity at the listener's position and the direction that the listener is facing. Localization error is decreased as the sound intensity increases up to approximately 70 dB [Davis & Stephens 1974]. Very loud sounds may also be difficult to localize due to the reverberant energy they generate. Reverberation causes an increased localization error due to the reflected sound waves carrying conflicting directional information [Giguere & Abel 1993]. To approximate the effect intensity has on sound localization, Equation 6.13 incorporates a parabola with an axis of symmetry located at 70 dB.

$$E_1 = (0.1 \times A_4 - 7)^2, \qquad (6.13)$$

where, E_1 is the amount of localization error in degrees, and A_4 is the amplitude of the sound at the listener's location in dB.

Source localization error ranges between 2.75° for sounds that are in front of the listener, and up to 20° for sounds that are behind and above or below the listener under ideal conditions (young listeners seated in an anechoic chamber) [Makous & Middlebrooks 1990]. Equations 6.14 and 6.15 provide an efficient approximation of localization error based on the direction the NPC is facing relative to the direction of the perceived sound.

$$w = \frac{\widehat{V_l} \cdot \widehat{V_s}}{2} + \frac{1}{2}, \qquad (6.14)$$

$$E_2 = E_1 + w \times E_f + (1 - w) \times E_b, \qquad (6.15)$$

where, w is a weight factor used to linearly interpolate between the forward facing error (E_f) and the error factor for sounds located behind the listener (E_b). Note that E_f is equal to 2.75°, and E_b is equal to 20° for ideal listeners. V_l is a normalized direction vector pointing in the direction that the listener is currently facing. V_s is the normalized direction that sound appears to be coming from based on the path(s) taken by the sound. E_1 is the output localisation error in degrees given by Equation 6.13 which is based on the perceived intensity of the sound. E_2 is the combined localisation error due to the sound intensity and the direction that the listener is facing relative to the sound.

If the path between the sound source and listener is unoccluded, then the path is a straight line. A path contains occluding geometry if the path taken by the sound has a greater length than the straight line distance. Occluded paths contain twists and turns that further obscure the origin of the sound. The localization error caused by occlusion can be approximated by comparing the length of the path to the straight line distance between the listener and the sound source. If we assume that the worst case for sound source localisation occurs when the path contains one sharp turn, then we can assume that the localization error caused by occlusion is between zero and our worst case depicted in Figure 6.14.

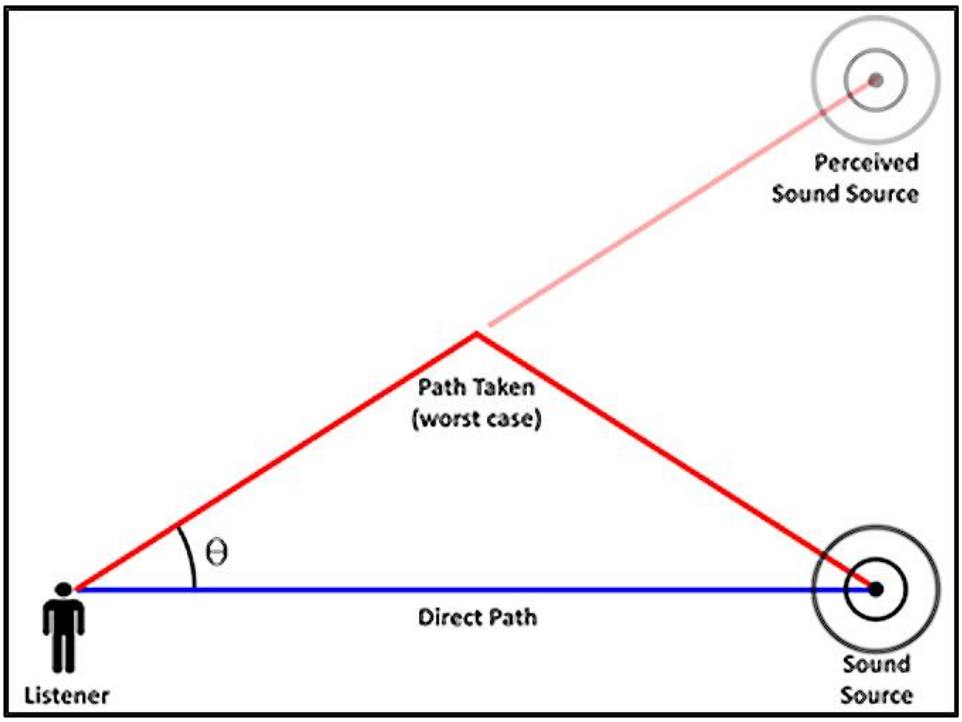


Figure 6.14: The worst case example, all of the occlusion is due to one sharp turn midway between the listener and the sound source. The listener may perceive the sound source as originating in a different direction due to PBSL.

The angle represents the maximum amount of localization error caused by path occlusion. The average error due to path occlusion is assumed to be half of the maximum or worst case error. This error level is added to the two previous error calculations based on the amplitude and the relative direction (Equation 6.16).

$$E_3 = E_2 + \tan^{-1}\left(\frac{\sqrt{D_P^2 - D_A^2}}{D_A}\right) \times \frac{180}{2\pi}, \quad (6.16)$$

where, D_P is the path length or distance travelled by the sound returned by the graph, and D_A is the actual straight line distance between the listener (NPC) and the sound source (player). E_2 is the amount of error caused by the direction of the sound combined with the error introduced by background noise. The total amount of localization error (E_3) can be used to determine the size of the area that the NPC

would need to search while looking for the source of the sound. A smaller search area implies that the NPC is more confident about their estimate (Figure 6.15).

The location of the sound source is estimated for each NPC based the amount of localization error. The worst estimate will place the sound source in the perceived direction ($\boldsymbol{V_p}$) with a distance equal to the path length ($\boldsymbol{D_p}$). The best estimate will place the sound source in the correct location which is the straight line direction to the sound source ($\boldsymbol{V_a}$) with a distance equal to the actual straight line distance ($\boldsymbol{D_a}$) between the sound source and the listener.

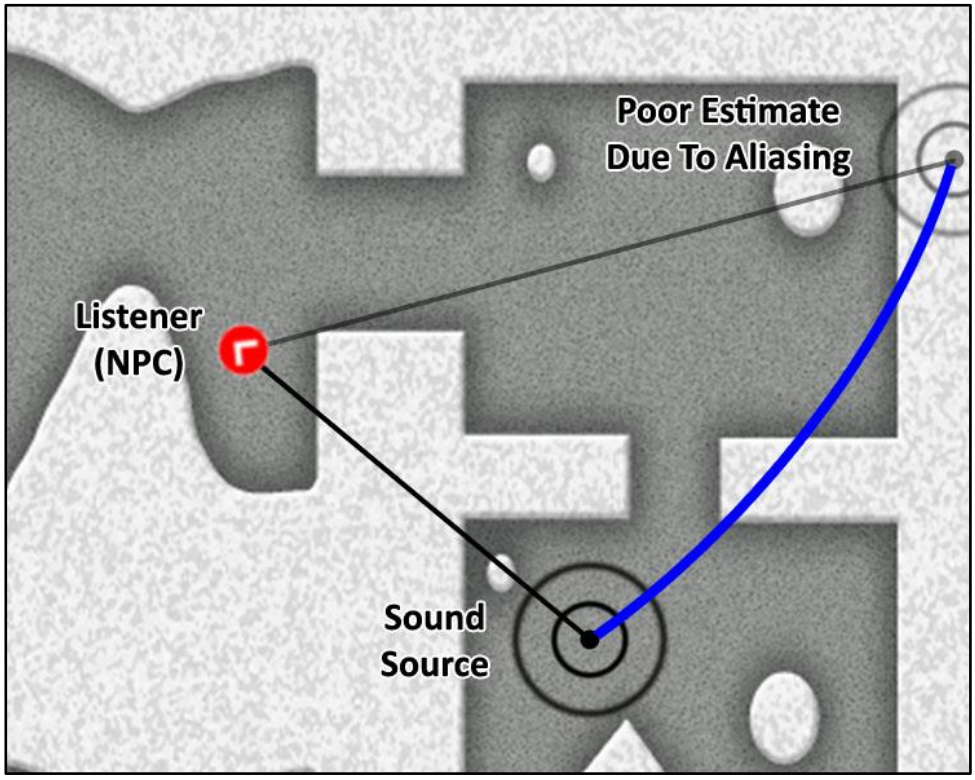


Figure 6.15: The diagram above contains a listener (red circle), a sound source, and poor estimate of the sound source's location caused by occluding geometry. The NPC pictured here will estimate the sound source to be located at some point on the curved blue line based on the accuracy calculated above.

$$w = \max(1 - \frac{E_3}{90}, 0), \quad (6.17)$$

$$\widehat{V_e} = \widehat{V_a} \times w + \widehat{V_p} \times (1 - w), \quad (6.18)$$

$$D_e = D_a \times w + D_p \times (1 - w), \quad (6.19)$$

$$L_e = L_l + \widehat{V_e} \times D_e, \quad (6.20)$$

where, $\boldsymbol{w}$ is a weight factor used to linearly interpolate between two normalized vectors; the direction that the sound is perceived to be arriving from ($\boldsymbol{V_p}$), and the actual direction vector pointing directly to the sound source ($\boldsymbol{V_a}$). $\boldsymbol{V_e}$ is a normalized direction vector pointing toward the location where the NPC estimates the sound source to be, and $\boldsymbol{D_e}$ is the estimated distance between the listener and the sound source. $\boldsymbol{D_p}$ is the length of the path taken by the sound in order to reach the listener, and Da is the actual straight-line distance between the listener and sound source. $\boldsymbol{L_l}$ is the location of the listener (NPC), and $\boldsymbol{L_e}$ is the position where the listener estimates the sound source (player) to be located.

The size of the area to be searched depends of the localisation error (the NPC's confidence), and the resolve of the NPC to find the source of the sound. For example, a lazy guard may only check the location where they estimate that the sound originated from before returning to relaxed state, whereas a more tenacious guard will continue to search a wide area around the estimated source location. Equation 6.21 can be used to decide on the size of the area to be search based on the localisation error, and the tenacity of the NPC that heard the sound. Note that the Equation 6.21 was designed to provide a convenient circular search area. This formula is not based on science, but was arrived at through informal testing, and is

provided for ease of implementation. Some programmers may find it difficult to construct a search behaviour for irregularly shaped areas.

$$r = \frac{E_3}{90} \times T \times D_p, \qquad (6.21)$$

where, $\boldsymbol{r}$ is the radius to be searched in meters, $\boldsymbol{E_3}$ is the estimation error from Equation 6.16, $\boldsymbol{T}$ is the tenacity or resolve of the NPC to find the player, and $\boldsymbol{D_p}$ is the path length provided by the graph. If $\boldsymbol{T}$ is set to 0, the NPC would only check the estimated location, if $\boldsymbol{T}$ is set to 1, the NPC would search a wide area around the estimated location before giving up.

The characteristics of an individual listener may also be incorporated into the model. 10 Db can be used as the threshold of hearing (amplitude of normal human breathing recorded from a distance of 1m) for ideal listeners (NPCs with excellent hearing) [NIDCD 2010]. However, higher values can be substituted to simulate NPCs with hearing loss. The threshold of hearing may also be lowered to simulate the superior sense of hearing in some animals such as dogs. In addition, localization error may be increased for NPCs wearing helmets or headphones. Additional background noise may be added for a short period of time after a loud explosion is heard by a NPC to simulate temporary hearing loss. Background noise may also be dynamic based on the NPC's proximity to sound sources in the environment. For example, it should be far easier for a player to sneak up on a NPC who is standing near loud machinery, or who is discharging a weapon.

NPCs can use the output from this algorithm to search for the source of the sound in a more realistic way. In a graph system, the environment is simplified for the NPCs by representing it as a series on interconnected nodes which can be searched efficiently using a path finding algorithm such as A*. The output from the method described here provides an area to be searched. Nodes located close to the center of the search area have a higher probability of containing the sound source (the player). Nodes outside of the search area would be given a probability of 0% or a probability close to 0%. The NPC could use this information to prioritize unsearched nodes based on the probability of the sound originating at that location, and their distance from the node. Nodes are considered searched if the NPC has an unobstructed line-of-sight with the node which results in the probability at that node being set to zero (there is 0% chance that the player is present there). The NPC might return to a more relaxed patrolling state once all of the nodes have been searched.

6.4.5 Limitations

Frequency dependent effects have not been considered. Where frequency is required for calculations, a frequency of 1,000 Hz has been assumed. Constructing an algorithm whereby frequency is not required as an input was done to improve ease of implementation as many game sounds contain a variety of frequencies or change frequency during playback. The frequency of the sound effects may be taken into account by suppling an average frequency of a similar sound taken from a look-up-table. In the future, sound files could be parsed in order to calculate their average frequency.

In addition, some of the values used by the formulae were arrived at through informal listener testing. Using the formulas presented here will provide a believable sense of hearing for NPCs. However, extensive user testing is required to fine tune the formulas and constants before this method could be used to predict audibility and localization accuracy for real-world environments. The algorithm presented here simulates human behaviour for NPCs by providing them with a realistic and customisable sense of hearing, it cannot be used to predict actual human behaviour.

6.5 Convolution Applications

In Chapter 3, the real-time convolution of sound samples with HRTF filters was explored using a variety of GPU and CPU implementations. Performance tests revealed that the CUDA and OpenCL implementations were efficient enough to allow for real-time HRTF convolution [Cowan & Kapralos 2013b]. However, the results were published in 2013, and since that time, HRTF and DSP effect processing has become a common feature of high-end sound cards designed for gaming applications. In this version of the framework, HRTF filtering is left to the sound card when hardware processing is available, and crudely approximated by the Fmod audio API when not supported in hardware.

That being said, convolution can be used for applications beyond HRTF filtering. For example, sounds could be edited to create alternate versions while the game is loading. Convolution filterers can be used to alter human voices to sound alien or robotic. Convolving a sound with a Gaussian filter will have a muffling effect that

can be used to simulate sound that has passed through a solid object such as a closed door or thin interior wall. Creating a separate muffled version of each sound can improve the occlusion effect over the standard implementation based on Fmod DSP effects. To implement this in the framework, the muffled and unedited sound are both played at the same location. The volume of each sound ranges from 0 to 1 in Fmod, likewise, the occlusion estimation returned by the graph has the same range. The occlusion value can be used to linearly interpolate the volume of the two sounds so that when occlusion is low, the unaltered sound is heard. As the occlusion increases, the unaltered version of the sound fades out as the muffled version is faded in.

6.6 GPU Acceleration

The efficiency of CUDA and OpenCL were compared by implementing HRTF convolution in both languages and measuring the processing time while varying the number of samples processed [Cowan & Kapralos 2013b]. The results indicated that CUDA out performed OpenCL when the sample size was less than 65,000. OpenCL out performed CUDA for sample sizes 65,000 and up. The convolution operation is not directly comparable to the graph-based sound propagation function. However, the results do indicate that the performance of CUDA and OpenCL are similar. CUDA was selected for the framework because of the language's similarity to the C and C++ programing languages that were used in the rest of the framework (the portion of the framework created to run on the CPU). In addition, Nvidia provides a free CUDA debugger in the form of a Visual Studio plug-in called Nsight which simplifies the process of adding CUDA functions to

C++ programs. It should be noted that the framework does not make use of any CUDA functionality that is exclusive to the language. The compute shader used by the framework can easily be adapted to other shading languages such as OpenCL or GLSL if CUDA is not desirable due to hardware restrictions.

6.7 Discussion

The spatial sound framework introduced here can provide realistic spatial sound for any type of virtual environment, including those constructed from polygonal geometry, voxelised geometry, or 2D sprites. The propagation of sound is modelled in real-time by first converting the virtual environment to a graph structure, and then modelling the propagation of sound efficiently by taking advantage of the inherent parallelism of the GPU architecture.

In addition to providing realistic spatial sound for human listeners, the framework can be used to provide a sense of hearing for AI controlled non-player characters (NPCs). The threshold of hearing and the localisation accuracy of each NPC can be customised to suite the type of character they are meant to portray.

The framework presented here is a proof of concept only. To prepare this framework for commercial use, the CUDA compute shader would likely have to be replaced with a GLSL shader for wider hardware support. Much of the code in the current version of the framework has not been abstracted away to simplify the coding interface. To use the framework in its current state, the user would have to be well versed in OpenGL, shaders, and the Fmod audio API. The interface of the

framework could be simplified to the point that it is suitable for novice users who do not have a strong background in audio. However, the creation of a user friendly and commercial-ready spatial sound API is beyond the scope of this thesis.

Chapter 7 User Study

Many gamers are highly competitive and may value winning over realism and immersion [Moeller, Esplin, & Conway 2009]. If the added realism afforded by spatial sound rendering does not result in improved task performance, competitive gamers may reject the technology in favour of traditional panning methods (binaural sound cues) which allow players to hear through walls and other obstacles providing accurate distance and direction information. If players find that spatial sound has a detrimental effect on their in-game performance, they may prefer basic amplitude panning for the tactical advantage. PBSL should allow players to find the source of a sound efficiently. However, the addition of spatial sound may add ambiguity, and more specifically, the player might not always know which direction the sound is coming from, and the attenuation caused by occlusion can easily be confused with attenuation due to distance. This ambiguity exists in the real world, however, players may not be accustom to experiencing ambiguity in video games as spatial sound is relatively new to real-time applications and is often unrealistic. A user study was performed to test whether player performance (navigation) could be improved with the inclusion of spatial sound over traditional panning (binaural sound cues only) and ray-cast occlusion (binaural sound cues with occlusion calculated by casting a single ray from the sound source to the listener) in a 3D first-person video game. Non-gamers or novice players may expect the sound in virtual environments to closely mimic real-world phenomena. Novice players may become confused by sound cues that seem to violate real-world acoustics (e.g., sound propagating completely through solid obstacles). Bălan et al.

[2014] demonstrated that the inclusion of spatial sound may improve the player's ability to navigate virtual environments, but does this hold for all players? Expert players may instinctively disregard spatial sound having become accustom to spatial sound cues being inaccurate or non-existent in the past. Therefore, the main question this study aims to answer is "how does the inclusion of realistic spatial sound affect performance (completion time) for novice and expert players?"

The secondary goal was to measure the efficiency of the spatial sound framework in a voxelised environment. Informal listening tests (conducted by the designers of the framework) have shown that the algorithm performs well with manually created graphs and 3D graphs that were automatically generated based on polygonal structures. However, a voxel-based environment requires a large number of nodes and many connections per node thus increasing computational complexity.

In order to test the effectiveness of the framework, I developed a simple 3D game and conducted a user study whereby each participant was tasked with navigating through a simple 3D environment using the standard first-person controls (mouse rotates the camera, and the keyboard is used to move the player). The participants were asked to find and collect the sound sources in the 3D environment. Sound sources which appeared as floating pulsing spheres were collected when the player moved close to them. Each time a sound source was collected, it disappeared and reappeared at another pre-selected location within the 3D environment.

7.1 Game Description

The game's graphical style was inspired by the extremely popular game Minecraft. Developer Mojang (owned by Microsoft), announced via Twitter on February 27, 2017, that 122 million copies of Minecraft have been sold so far, and it is estimated that 55 million people play Minecraft each month [Sarkar 2017]. In addition to game sales, Microsoft earns royalties from Minecraft merchandise which includes toys, clothing, and books [Minecraft 2018].

By basing the graphics, and to some degree, the game-play on Minecraft, it was anticipated that many of the participants would find the game world familiar and less intimidating. In addition, a simplified environment constructed completely from cubes should be easier to navigate than a visually complex setting constructed from irregular shapes. Since all of the cubes were the same size, the player was able to judge whether or not they can fit through an opening or jump over an obstacle. In addition, less 3D modeling skill is required to create and alter the environment which saved a considerable amount of time and effort.

7.1.1 The Cube Craft Editor

Rendering voxels directly is certainly possible, but it is far more efficient to render the scene as a polygonal mesh. Therefore, the world is represented as both a 3D array of voxel data and a 1D array of vertex and surface normal data. Before implementing the sound system, it was first necessary to build a world editing program capable of adding and deleting blocks in real-time with a "what you see is what you get" (WYSIWYG) first person graphical user interface (GUI).

The Cube Craft world editor allows the user to fly around and through objects. The editor uses the standard first-person controls, WASD keys move the camera, and moving the mouse rotates the camera. A single ray is projected from the center of the camera to the first intersecting cube. A semi-transparent white cube appears at the intersection point to act as a cursor. The cursor cube shows the user where a new cube would be added if the left mouse button is clicked. The cursor cube also has one side shown in red. The red side highlights the surface of an existing cube. This is the cube that will be deleted if the user presses the right mouse button (see Figure 7.1). The up and down arrow keys are used to select the material that is applied to newly created cubes. The name of the currently selected material is displayed as the window's title (title bar). The editor saves the environment as a 512 KB binary file that is simple to parse. In addition, Cube Craft is able to export the environment as an OBJ file which is an open standard 3D model format that can be used with most 3D modeling and sculpting software.

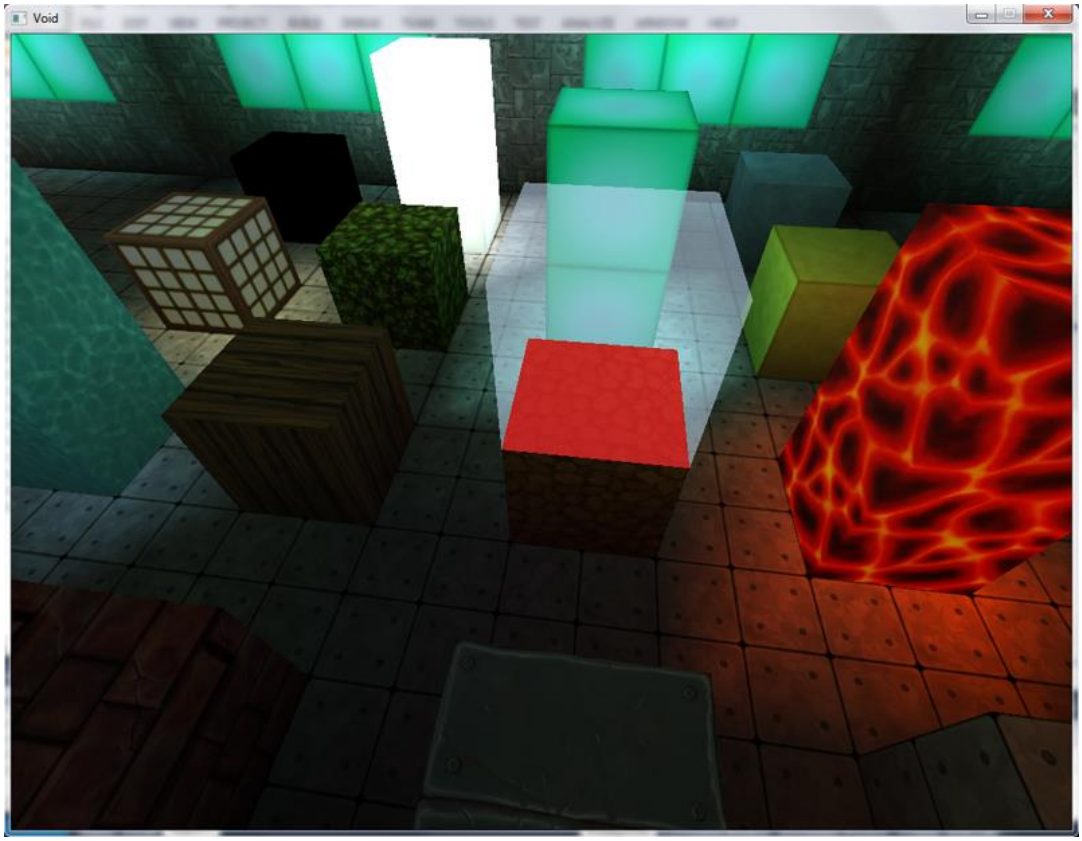


Figure 7.1: A screen capture of the Cube Craft editor interface.

The Cube Craft editor was instrumental in testing the spatial sound rendering framework since it allows changes to be made to the environment in real-time. As each new block is added or removed from the scene, the audio graph is updated as well. The number of sound nodes remains constant and their connections to neighboring nodes are not changed. Instead, the occlusion values assigned to each connection are updated when blocks are added or removed between sound nodes. For example, building a wall between two sound nodes increases the occlusion value of their connection, implying that less sound will be able to pass between the two nodes. The sound would instead travel around the occluding obstacle by passing through additional nodes and connections, changing the path. This leads

to additional attenuation and may change the direction that the listener perceives the sound to be coming from (PBSL).

7.1.2 Building a Graph based on Voxels

The resolution of the voxel array is 128 by 128 wide and 32 voxels high. This represents the maximum volume of the playable area. The environment used for the study is a fairly irregular shape and takes up about half of the available volume. The graph used to calculate the spatial sound fills the entire volume, including areas that the player is unable to travel to. Walls and other obstacles prevent the player from traveling too far off course. Some of the areas outside of the playable space are populated with objects such as trees and buildings. These objects are used as scenery, suggesting to the player that there is more to this world beyond the confines of the playable space.

The resolution of the graph used to calculate the spatial sound is independent from the rendered scene. For convenience, the sound nodes are spaced exactly two blocks apart leading to a resolution of 64 by 64 wide and 16 nodes high. The occlusion value for each connection is calculated by sampling the occlusion value of the 8 blocks that lie directly between neighboring sound nodes. Every node in the graph is processed each time the system is updated which happens once per rendered frame or 60 fps (frames per second).

7.1.3 The Game World

The environment used in the study was designed to represent a variety of genres. It contains both indoor and outdoor areas, as well as natural and manmade spaces such as buildings. Different time periods are also represented. The game begins in a medieval courtyard with a castle visible as scenery. There is a small bridge, a fountain, and several trees (see Figure 7.2a). There are also two small houses, one of which has a staircase leading to a second floor (see Figure 7.2b), while the other house is a ruin. Sound nodes zero and one are located in this area. Nodes two, three, and four are located in a cave environment with irregular structures and winding tunnels (see Figure 7.2c & 7.2d). Once the player has cleared an area of sound nodes, there is never a need to backtrack through the previously explored area (a commonly used approach in commercial video games intended to prevent players from becoming bored or lost). Nodes five and six are found in a futuristic building with cement floors and metal or concrete walls (see Figure 7.2e), and there are two staircases which lead to the final area. Nodes seven, eight, and nine are found on the second floor which features a contemporary style including carpeted floors, and a striped wallpaper texture (see Figure 7.2f).

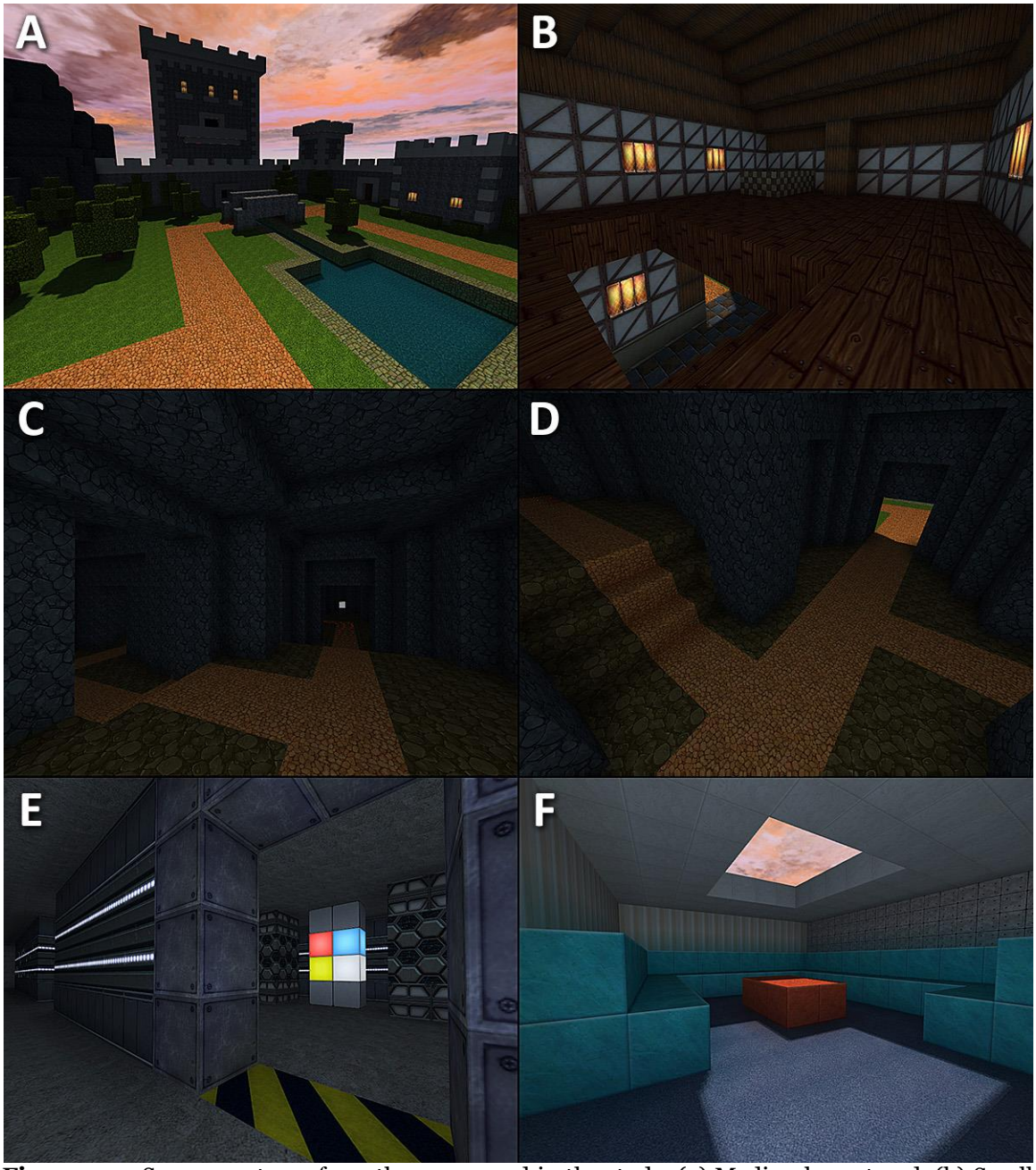


Figure 7.2: Screen captures from the game used in the study. (a) Medieval courtyard, (b) Small medieval house, (c) & (d) Cave, (e) Futuristic lower floor, (f) Modern style upper floor.

Figure 7.3 illustrates a top-down map of the environment showing the areas that the player can reach only. The map was made by rendering the environment orthographically from the top-down. Yellow arrows show where staircases connect the upper floor to the ground level. The location of the sound nodes are marked by

white circles and are numbered zero to nine. The white circle without a number is the starting point. The player begins the game at this location facing the small bridge where node zero is located.

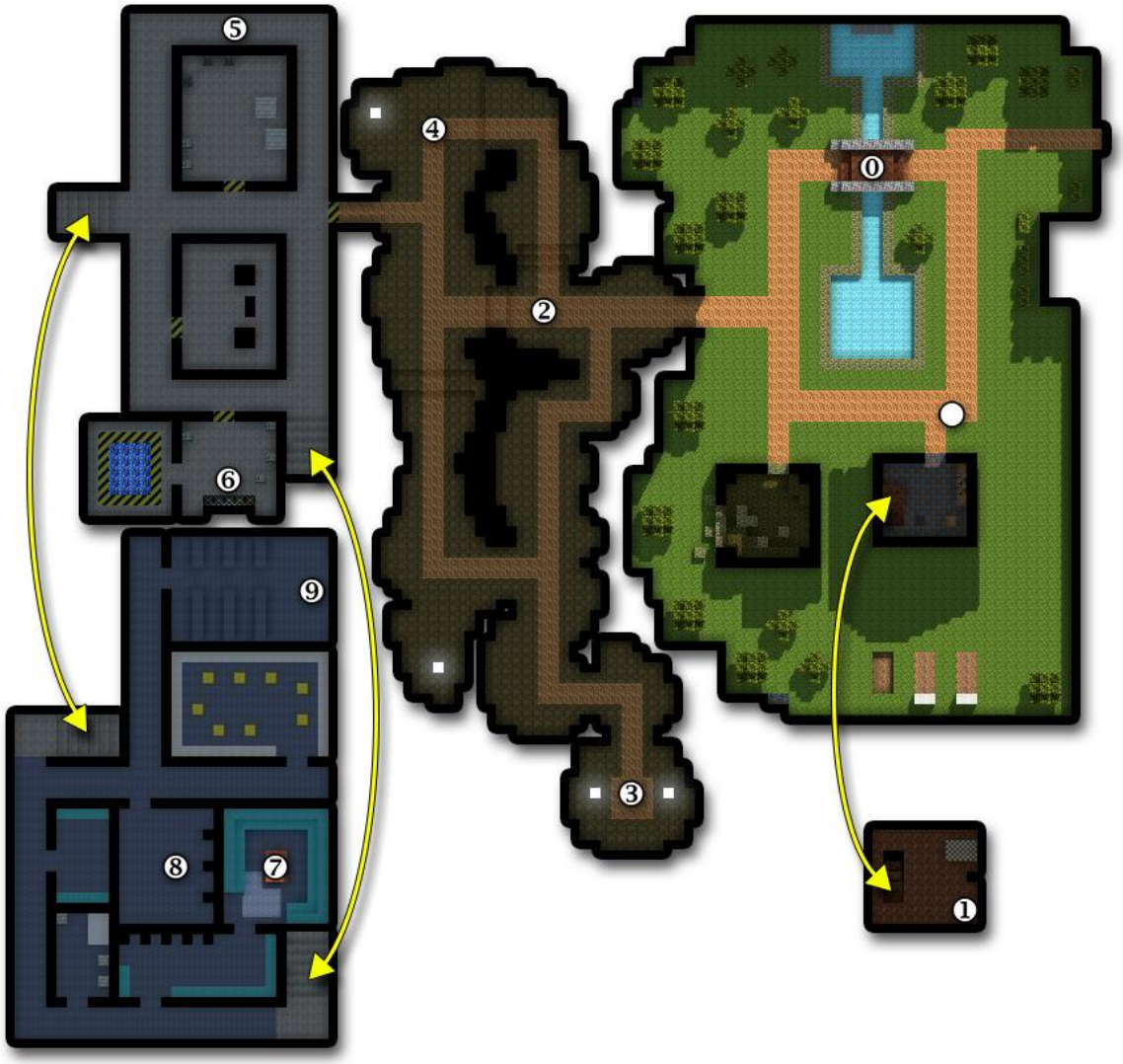


Figure 7.3: A top-down map of the game environment with the starting point and nodes marked by white circles. Yellow arrows show the points where the lower and upper floors are connected with staircases.

The time taken for the participant to travel from the starting point to node zero is not relevant to the study. Maneuvering from the starting point to node zero allows the player to become familiar with the controls (keyboard and mouse) and physics (a small amount of momentum is applied to lateral movement and simulated

gravity provide a downward acceleration when jumping or falling). Participants were encouraged to familiarize themselves with the game and adjust the volume if necessary before touching the first node. Participants were advised that they will be timed and that they should "go as fast as they can" after collecting the first sound node. Researchers stopped talking to the participant once the first node was collected but remained nearby to answer any questions the participant might have.

Three different algorithms were used to render the 3D sound: i) amplitude panning, ii) amplitude panning with Fmod's ray-cast occlusion, and iii) our GPU spatial sound method. The ordering of the three algorithms was randomly selected when the game started. Three of the sound nodes rendered the sound using amplitude panning, three of them will used panning with occlusion, and three used our method. The algorithm used to render the sound for node zero was chosen randomly too, although the time taken to reach node zero was not relevant to the study. Only one node at a time was visible and emitting sound to prevent interference from other nearby nodes and prevents the player from collecting them in the wrong order. When the last sound node was collected, a "thank you" message appeared on the screen for a few seconds as the program saved the data that it recorded before closing.

7.2 The Participants

The user study was conducted between the 5th and 13th of December, 2018. There were 23 participants in total, 19 males and 4 females. The majority of participants were students recruited from the Faculty of Business and Information Technology

at the Ontario Tech University located in Oshawa, Ontario, Canada. Participants were asked to sign a consent form and answer the three demographic questions below (figure 7.4).

What is your gender?

- ○ Male
- ○ Female

How old are you?

- ○ Under 20
- ○ 20 – 29
- ○ 30 – 39
- ○ 40 – 49
- ○ 50 – 59
- ○ Over 60

How much time do you spend playing video games in an average week (this includes games you played on a mobile devices, games played in a web browser, as well as traditional computer and console games)?

- ○ 0 hours a week (I do not play video games)
- ○ Less than 2 hours a week
- ○ 2-5 hours a week
- ○ 5-10 hours a week
- ○ 10-20 hours a week
- ○ More than 20 hours a week

Figure 7.4: Survey questions

The participants where not specifically asked about hearing disabilities in the questionnaire. However, none of the participants wore a hearing aid or had difficulty understanding verbal instructions. In addition, the volume of the game was set to a comfortable level, and participants were advised that they were permitted to adjust the volume at any time. The majority of participants used the default volume level set by the investigators.

Fifteen of the nineteen male participants were under the age of 30 (79%), while all four female participants belonged to the twenty to twenty-nine age bracket. Five to ten hours per week was the most common answer selected when asked about the average time spent playing video games per week (see Figure 7.5). This answer may have been skewed lower due to the timing of the study. December is the end of the fall semester and many of the participants were students who were very busy completing final assignments and studying for exams.

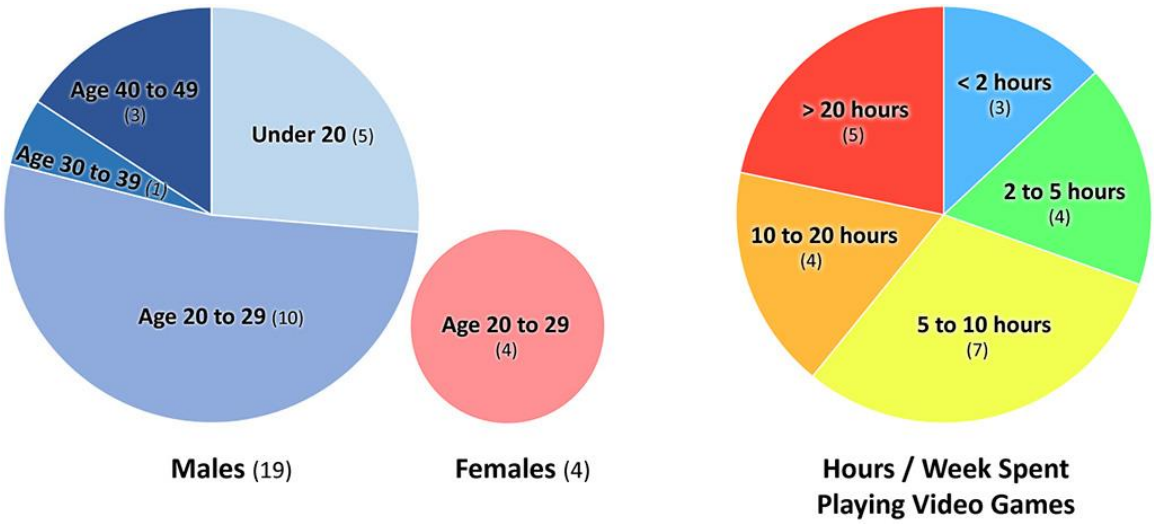


Figure 7.5: Participant age, gender, and hours per week spent playing video games.

In addition to the demographic questions, the game itself recorded game play data as it was played. The spatial sound algorithm used to render the sound for each node was recorded along with the completion time. In addition, the exact position of the camera (player) was recorded twice per second (every 0.5 seconds). The orientation of the camera was also recorded twice per second and stored as a highly compressed binary file. The recorded play-though data is accurate enough to allow the investigators to replay any of the participant's play-throughs, seeing and hearing exactly what the participant experienced during the trial. This play-

through data may also be viewed in third-person mode which allows the investigators to view the player as a 3D avatar roaming around the environment. Having access to the recorded play-through data allowed the investigators to closely examine the behaviour of the participants without the need to look over their shoulder during the trial which might have made them feel uncomfortable or influence the way they played. The play-through data was especially valuable for examining the decisions of participants who went way off course or became stuck temporarily unable to find the next sound node.

7.3 The Results

Completion times differed between novice and experienced players. Participants were not asked about their gaming experience directly, instead, they were asked how many hours they spent playing video games per week on average. The answer is quantifiable and therefore more objective than asking them to rate their own playing ability. Participants who estimated that they spend less than five hours/week playing video games (seven participants) required on average eight minutes and 38 seconds (8:38) to collect nodes 1 through 9. It should be noted that completion times varied greatly for this group ranging between 2:42 and 18:23. Several participants who claimed to spend less than five hours/week were clearly very experienced gamers and likely made their selection based on recent weeks rather than an average week. In contrast, participants who indicated that they spent more than 10 hours/week playing video games (nine participants) had an average completion time of 3:22. There was less variation in the completion times for this group which ranged between 2:30 and 4:23 (see Figure 7.6).

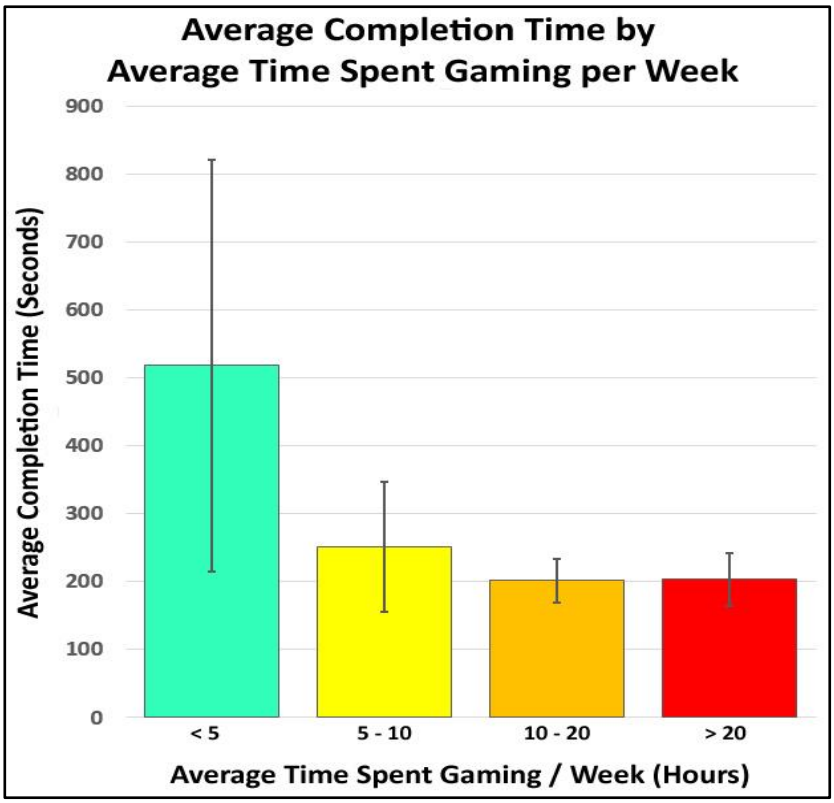


Figure 7.6: Completion time (performance) by hours per week spent playing video games (experience).

Figure 7.7 shows the average completion times for each node with participants grouped by the number of hours spend playing video games per week. Participants who spend less than five hrs/w playing video games took longer to travel between nodes. Novice players were more likely to get lost (travel in the wrong direction and then have to backtrack) and some of them had difficulty coordinating the mouse and keyboard in order to move around obstacles efficiently. Traveling between node 6 and arriving at node seven required the most amount of time even through these nodes were not far apart. The difficulty stems from the fact that node 6 was on the first floor and node seven was on the second floor. This may not have been obvious depending on the random method selected to render the sound for this leg of the journey. Both novice and experienced players found themselves confused,

but the experienced players generally performed a more efficient search and quickly concluded that the sound source must be on the other floor.

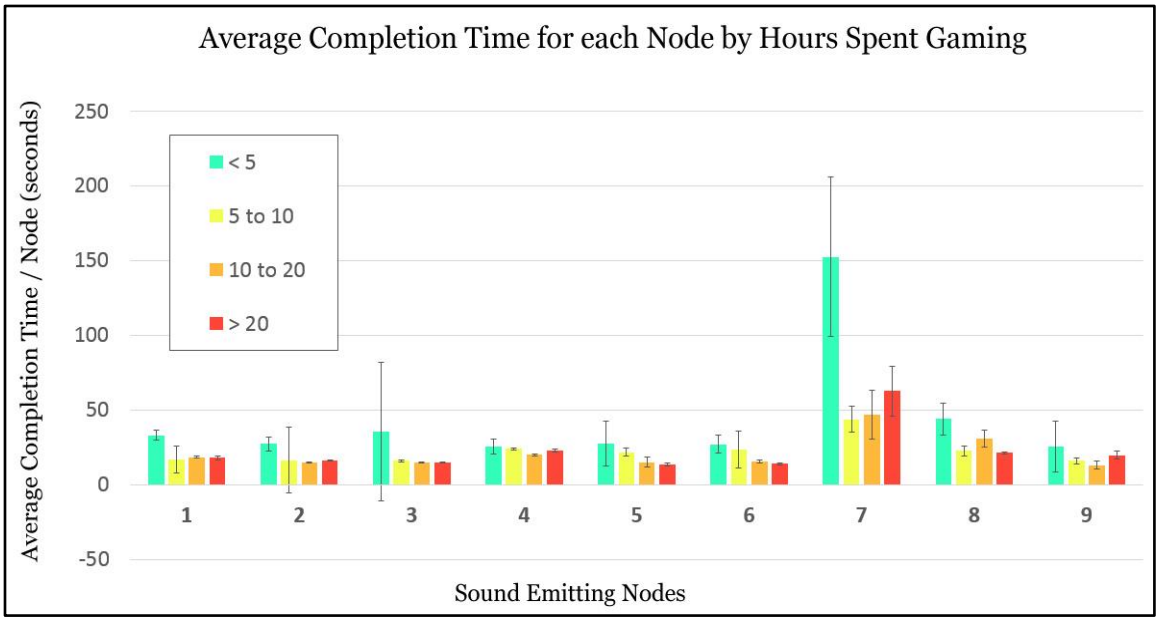


Figure 7.7: Completion time per node separated by experience (hours / week spent playing video games)

Figure 7.8 shows the average completion time based on the algorithm used to render the sound. Since each participant experienced all three of the sound rendering methods, we calculated the completion time for each method by first calculating the average completion time for each node and method. The total completion time is then the sum of the average node completion times by algorithm.

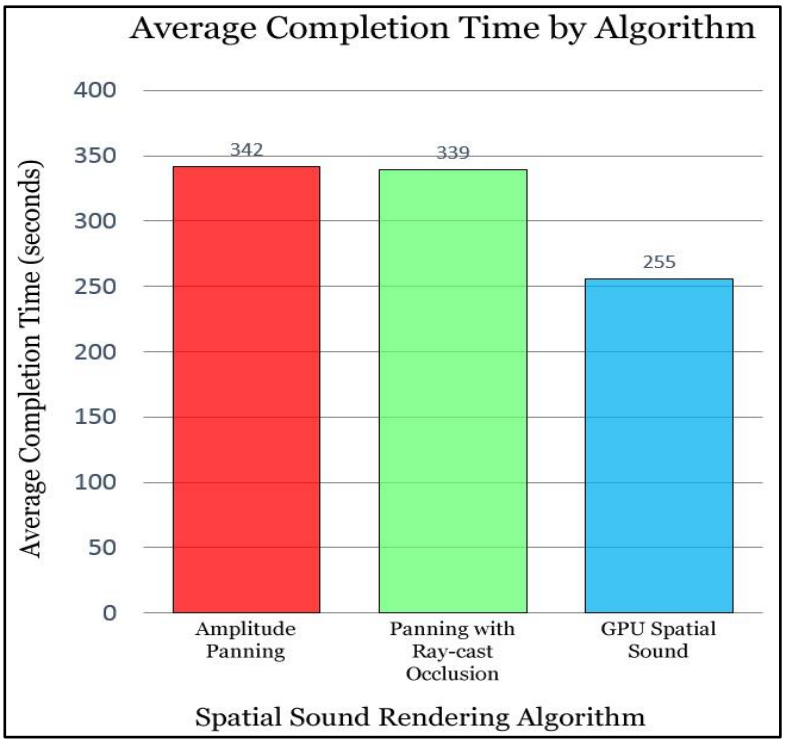


Figure 7.8: Total median completion time by sound rendering algorithm.

The total completion time with amplitude panning was 5:42. When the ray cast occlusion effect is applied to the sound, the completion time decreased slightly to 5:39. This simple occlusion method should have increased the realism of the rendered sound, but did not lead to a significant decrease in the completion time. Our GPU based spatial sound algorithm had the shortest completion time at 4:15. Figure 7.9 shows the completion time per node by algorithm.

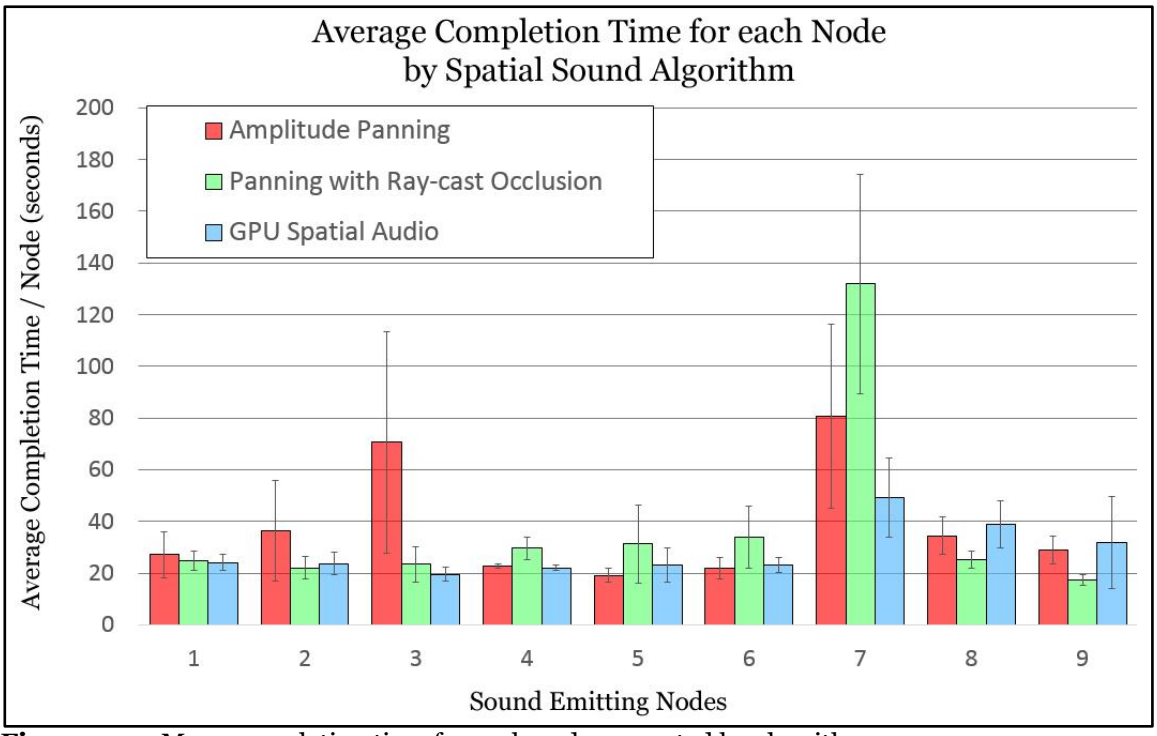


Figure 7.9: Mean completion time for each node separated by algorithm.

For most of the nodes in the game, the completion time was unaffected by the algorithm used to render the sound. The greatest difference in the completion time between the algorithms occurred when the players attempted to navigate from node 6 to node 7. None of the sound nodes were hidden, and in most cases, a vague idea about the direction of the sound combined with a quick visual search was all that was needed to find the next sound source. Node 7 was the most difficult node for players to find. Traveling from node 6 to node 7 required the longest amount of time despite the fact that the distance between them is relatively short (compared to the distance between some of the other sets of nodes). The problem was that node 6 is on the lower level and node 7 is on the second floor. Both amplitude panning methods allowed sound to pass through the floor which resulted in players believing that the sound was on the same level as node 6 (the last one collected). Experienced gamers know not to trust the sound cues provided by video games.

They typically performed a quick visual sweep of the area before moving to the second floor. Inexperienced players could not accept that they were hearing a sound that had passed through the floor above. Instead they thoroughly searched the first floor, often rechecking the same room two or three times. One participant exclaimed “it’s right here, why can’t I see it!”, while standing directly under the sound located on the floor above. It was only when they realized that they have found the source along the x and z plain, that they realized that the sound might be above them. The addition of ray cast occlusion only exacerbated the problem. The occlusion made the sound source seem farther away and it was more difficult for some players to pinpoint the location in the xz plane. In some cases, they exhaustively searched the first floor before trying one of the two stair cases leading up to the second floor. The completion time for node 7 with spatial sound was also higher than it should have been based on the path length between nodes 6 and 7. After reviewing the playback data, it is clear that the extra time was due to players distrusting the sound cues. The spatial sound was in fact guiding the players toward the staircase, however, most players still peeked into rooms along the way just in case the sound cues they were hearing were inaccurate. This suggests that the difference in completion times between amplitude panning and spatial sound would increase if players became accustomed to spatial sound and were able to trust the directional information it provides. Figure 7.10 shows the playback data for panning (red) and ray-cast occlusion (green) overlaid onto the map of the environment.

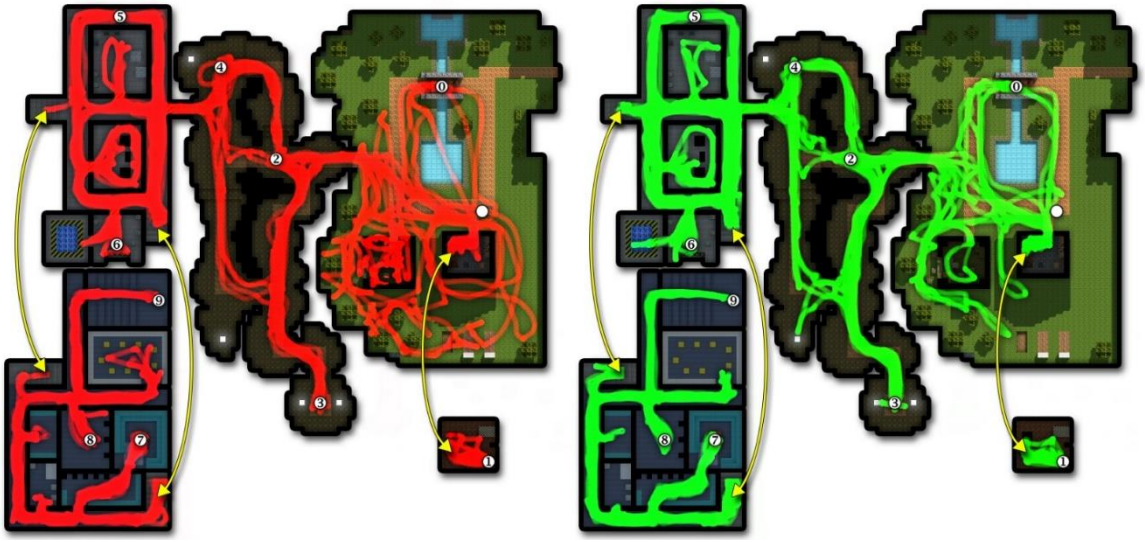


Figure 7.10: Paths made by participants. The left image shows paths in red made by participants hearing sound rendered with amplitude panning. On the right, paths shown in green were made by participants hearing amplitude with the addition of ray-cast occlusion.

Participants had a tendency to wander off course while experiencing amplitude panning or panning with ray-cast occlusion. In the medieval portion of the map, we see paths generated from the play through data that show participants exploring areas where there are no sound nodes present. Figure 7.11 below shows the paths created by participants guided by the graph-based GPU spatial sound. Notice that the participants were less likely to wonder of course or become lost. In addition, they did not fully search rooms that did not contain any sound nodes. In most cases, the participants noticed that the volume dropped slightly when they entered the wrong room and the PBSL provided by the graph, directed them back out into the hallway.

Figure 7.11: Paths created by participants using the graph-based GPU spatial sound (shown in blue).

Spatial sound also decreased the completion time for Node 9. While traveling from node 8 toward node 9, participants arrived at point where they had to pick one of two hallways. Based on panning alone, it was impossible to know which of the hall would lead to the sound source. Most of the participants chose the correct hallway when the sound was rendered with our graph based spatial sound. Participants who experienced either of the panning methods had about a fifty percent chance of guessing the wrong hall and then having to double back.

Figure 7.12 shows the average completion time by the average number of hours spent gaming per week. The three methods are graphed separately to show the effect of each method on the completion time.

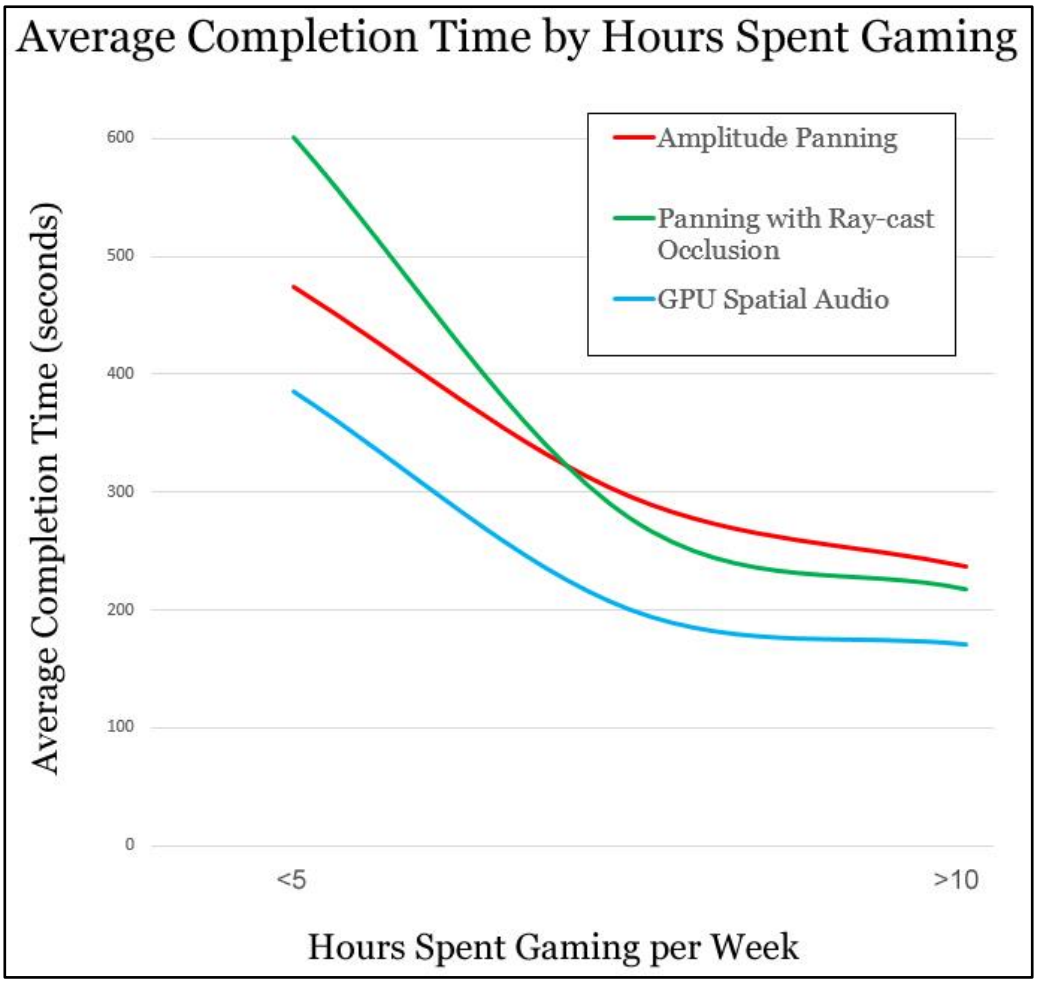


Figure 7.12: The average completion time by the hours spent gaming per week. The three methods (amplitude panning, panning with ray-cast occlusion, and GPU spatial sound) are plotted separately to show how they affect the task completion time of players at different skill levels.

Participants who indicated that they spend more than 10 hours per week playing video games, were least affected by the method used to render the sound. This suggests that experienced gamers have learned not to trust the sound cues provided by video games. Experienced gamers tended to conduct a well-practiced visual sweep of the area, moving sideways as they passed open doorways to visually scan each room without entering. Novice players expected sound to behave as it does in the real world which resulted in them occasionally getting stuck because the panning methods allow sound to pass through solid objects such as walls, which

resulted in the player being led to dead ends. The completion time differed between novice and experienced players with spatial sound rendering as well, but to a lesser degree. This difference in task completion time is likely due to the experienced players having familiarity with the standard first-person controls as well as the ability to perform efficient searches visually.

7.4 Processing Time Requirements

The graph is arranged as a 3D grid of nodes 64 by 64 nodes wide and 16 nodes high. Each node can be connected with up to 18 neighboring nodes. In total the graph contains 65,536 nodes and 559,392 by directional connections. However, in order to implement the system efficiently on the GPU, each connection is one directional. A by directional connection is implemented by providing both neighboring nodes with a one directional connection to the other node. A total of 1,118,784 one directional connections are processed each time the system is updated. As a measure of running-time requirements, the graph was updated 1,000 times in a loop without rendering the scene graphically. The listener was placed at the starting point of the game (the exact placement of the listener and sounds does not have a noticeable effect on the processing time because every node in the graph is processed once per update regardless of the listener position).

The average processing time required to connect the listener to the nearest node, update the neighboring (connected) nodes, send the node data to the GPU, process each node using CUDA, and read back the node data to the CPU was 1.1154 ms. The tests were performed using an Alienware Auroa R6 desktop computer with Intel

Core i7-7700 (4 cores, 3.6 GHz) Processor with 32 GB of RAM and an NVIDIA GeForce GTX 1080 Ti GPU with 11 GB GDDR5X standard memory and 3584 CUDA cores.

7.5 Discussion

Spatial sound increases the realism and the player's sense of immersion which is very desirable for video game and other virtual environments. However, simple amplitude panning provides the most accurate distance and direction information to the listener by ignoring all occluding objects, providing the player the ability to hear through walls. Therefore the adoption of spatial sound in video games might have encountered resistance from competitive gamers looking for any advantage over the competition. For certain tasks, amplitude panning may provide some tactical advantages. However, the results of this study indicate that the inclusion of spatial sound improved player performance by reducing navigation time for both novice and experienced players.

The spatial sound for this study was rendered using the spatial sound framework introduced here. However, the result suggests that any spatial sound method capable of providing realistic PBSL, should result in improved player navigation for players at any skill level. In addition, with spatial sound cues guiding the player through the environment, visual navigation cues such as mini-maps and arrows may become unnecessary. The inclusion of spatial sound in video games and virtual environments will improve the realism of the simulation, improve the player's

ability to navigate, and may help to reduce the visual clutter of heads-up displays (HUDs).

The run-time test showed that graph-based sound propagation can be processed efficiently enough for real-time applications by utilizing the GPU. This study also proves that the spatial sound framework presented here is capable of supporting fully dynamic 3D voxel or grid type environments.

Chapter 8 Discussion, Conclusions, and Future Work

Given the importance of sound, and our ability to localize sound sources in three-dimensions in the real world, virtual environments (including virtual simulations, serious games, and video games in general), where often times the goal is to replicate the real world, should include realistic spatial sound cues, However, the generation of spatial sound, that is, modeling the propagation of sound within a virtual environment while accounting for the human listener, is difficult, technically challenging, and computationally expensive [Li et al. 2006].

8.1 Developing the Spatial Sound Framework

The use of nav-mesh graphs for spatial sound has been used commercially in Rainbow Six as well as other games by Ubisoft [Guay 2012], despite the limitations. A fully 3D graph system could remove the limitations inherent in the 2.5D nav-mesh. Furthermore, reverberation, and occlusion effects could be applied using parametric methods in order to remain computationally efficient.

8.1.1 Graph-Based Sound Propagation

Motivated by the success of this earlier work, graph-based sound propagation modelling was explored culminating in the creation of the spatial sound framework introduced here. Early works including GPU convolution, occlusion rendering, and reverberation sampling are now components in this unified framework. The occlusion work provides an approximation of the amount of acoustical occlusion

between two points in space. The occlusion work was intended for situations where the sound source and listener are located in the same room, or in separate rooms that share an opening. The algorithm assumes that sound is generally moving away from the sound source and towards the listener. It cannot accurately simulate occlusion for environments that contain winding paths whereby sound has to change direction at various points in order to propagate through the structure.

A GPU based spatial sound algorithm was developed that efficiently operates in real-time to provide realistic spatial sound for video games and virtual environments. Sound propagation is modeled using a graph of interconnected nodes. By replacing complex geometry with a graph structure, individual interactions with each surface can be ignored. Instead, graph-based sound propagation models the aggregate movement of sound. It is also possible to remove the geometry completely from the computational process by converting the geometrical description of the environment to a structured graph whereby nodes are placed in open areas (areas devoid of occluding structures), throughout the environment. Nodes are connected to their neighbours only if sound is able to pass between them. An occlusion value representing the amount of sound energy that is able to pass between the nodes is stored for each connection (the occlusion work detailed in Chapter 4 can be used to calculated the occlusion value stored in each connection). In addition, each node is capable of storing information regarding its surroundings such as the average reflectivity and the size of the space (the acoustical reflectivity work detailed in Chapter 5 can be used to provide the reflectivity and room size values for each node). Graphs have the advantage of

being efficient to parse and can potentially be processed in parallel to take advantage of the GPU architecture.

The shape of the graph and the properties assigned to each node and connection may be calculated based on the geometry, whether the environment is voxel based or constructed from polygons. One of the features that makes this framework unique is its geometry independence. The graph can be made simply by plotting points and connections manually. This framework can be used to render believable spatial sound for 2D environments or for games with hand drawn graphics.

8.1.2 Providing a Sense of Hearing for NPCs

NPCs often do not behave in realistic manner because they do not perceive the world in a realistic manner. The ability to model sound propagation for dozens of NPCs was not possible in the past. In many video games, the NPCs behave as if they are deaf, unable to hear loud noises such as gun fire in the very next room. This type of NPC only becomes aware of the player when a direct line-of-sight is established. Other games provide the NPCs with a rudimentary sense of hearing that is often completely distance based (the straight line distance between the sound and the listener). The NPC becomes aware of the player if the player makes a sound within a specified radius of the NPC. Without propagation modeling, occluding structures such as walls, ceilings, and floors are often ignored which can, at times lead to NPCs detecting the player under impossible circumstances.

The graph based spatial sound framework introduced here, produces a graph that is completely filled with simulated acoustical occlusion data. While calculating the occlusion between the listener (player) and each node in the graph for the purpose of rendering spatial sound for a human listener, the algorithm has provided enough information to simulate a sense of hearing for NPCs. This information is a by-product of the graph based sound propagation method. Repurposing this data for artificially intelligence requires no extra processing on the GPU, and very little extra processing on the CPU. The method outlined in Chapter 6 is capable of providing a realistic sense of hearing for any number of NPCs in real time which can result in them behaving in a more realistic manner.

8.2 Study Results and Implications

Traditional panning methods (binaural sound cues) do not model sound propagation. Sound travels directly from the sound source to the listener, ignoring all obstacles in between. The listener always perceives the direction of the sound source accurately and the attenuation is based on the direct distance between the sound source and the listener, not the distance traveled (path length). A player following the sound in order to locate the source can be lead to dead ends because the sound is able to pass through walls while the player cannot. The addition of ray-cast occlusion improves the realism by rendering the sound with muffling applied to it when the direct line of sight between the listener and the sound source is blocked. However, the direction of the sound is still perceived to point directly at the sound source even when the source is occluded. Although unrealistic, panning

methods do provide accurate distance and direction information to the listener allowing them to hear through walls and other obstacles.

Spatial sound including propagation modelling, provides different information about the sound source location. Instead of attenuation based on the direct distance, the attenuation is based on the path(s) taken by the sound as well as the occluding objects encountered along the path. The listener is unable to accurately judge their distance from the sound source. For example, a sound could be physically located on the other side of a wall only a few meters away, but it can seem distant to the listener based on the path that sound travelled in order to reach the listener. The perceived direction of the sound source is also inaccurate due to occluding structures. The direction that the listener perceives the sound to be coming from is based on the path taken by the sound. In addition, the directional information can be somewhat ambiguous in situations where sound is able to reach the listener from two or more opposing directions. However, the spatial sound framework models the way that sound behaves in the real world. Although amplitude panning methods provide accurate distance and direction information, they are noticeably unrealistic. Experienced gamers, accustomed to experiencing amplitude panning in many of their favorite games, may take advantage of the information provided. However, novice players may expect the sound in the game to behave as it does in the real world and become confused by the unrealistic sound cues.

A study was conducted to measure the task performance of both novice and experienced players. There were 23 participants in total (19 males and 4 females), (83% of participants were under the age of 30). In addition to age and gender, participants were asked how many hours they spend playing video games in an average week. 61% of participants indicated that they played games for 10 hours or less per week. The participants were asked to wear headphones and play a first-person 3D video game. The game's environment was constructed completely from cubes with a similar graphical style to the popular game Minecraft. This simple graphical style was primarily selected for ease of navigation. The world consisted of both indoor and outdoor spaces, with four different graphical themes (medieval, cave, contemporary, and futuristic). The goal of the game was to collect 10 sound emitting orbs as fast as possible. Only one orb exists at a time. Orbs were collected when the participant moved close to them. As each orb was collected, it disappeared and reappeared at the next location which was in most cases hidden by walls and other obstructions. The participants were forced to use sound cues to track down each orb which emitted a constant drum beat that could be heard from a considerable distance. Each participant's play-though was recorded so that the investigators could replay any of the participant's play-throughs, seeing and hearing exactly what the participant experienced during the trial. This allowed investigators to better understand the strategies used by the participants.

Three algorithms were used to render the 3D sound: i) amplitude panning (binaural sound cues only), ii) amplitude panning with Fmod's ray-cast occlusion, and iii) the graph-based GPU spatial sound method. The ordering of the three

algorithms was randomly selected when the game started. The spatial sound rendering included ambiguity. It was not always clear which of the available paths was the most direct in situations where sound was able to reach the listener through multiple paths. The ambiguity did not seem to noticeably impact the participant's ability to navigate. In situations where the sound's direction became ambiguous, participants behaved much as they would in a real-world scenario, they guessed at the direction, travelled a short distance, and then listened to see if the sound intensity was increasing or decreasing.

The amplitude panning methods allowed sound to pass through solid geometry such as walls. The ray-cast occlusion rendered the sounds with a low-pass filter when the direct line-of-sight was blocked by walls. However, the direction of the sound still pointed directly through solid structures. This led participants into dead-ends which were confusing for novice players who expected that the sound in the game world would behave the same as sound in the real world. Both amplitude panning methods allowed sound to pass through the floor which resulted in players believing that the sound was on the same level they were. Experienced gamers knew to ignore the spatial sound cues, and instead focused on performing a visual search of the area using the sound only as a rough guide. Inexperienced players could not accept that they were hearing a sound that had passed through from the floor above. Instead they thoroughly searched the first floor, often rechecking the same room two or three times. Both novice and experienced players performed better with spatial sound.

Aside from its use in entertainment-based games, spatial sound can play an important role in serious games and simulations. For example, the inclusion of spatial sound may reduce the learning curve for non-gamers, which will allow them to focus on the training that the simulation is meant to provide. In addition, spatial sound including PBSL, can remove the need for artificial visual guidance systems or heads-up-displays (HUD) which negatively affect emersion [Zhou et al. 2007]. HUDs generally consist of 2D elements rendered over the 3D scene. One of the most common HUD elements is a mini-map which is used to guide the player to their next mission objective, inform them about nearby points of interest, or alert them about approaching enemy units. With spatial sound, the mission objective, points of interest, and enemies could emit sounds which guide the player to them (or perhaps away, in the case of enemies).

The game world used in the study was made entirely from textured blocks, 128 by 128 blocks wide and 32 blocks high. Each block measured one half of a meter cubed. The resolution of the graph used to provide spatial sound measured 64 by 64 nodes across, and 16 nodes high. The nodes were spaced exactly one meter apart. This tightly packed grid of nodes represents the worst case scenario for the algorithm. An optimised graph produced by removing nodes that are located within solid structures, and merging nodes that are close together, can be used to represent the same environment, while substantially reducing the number of nodes and the processing requirements.

As stated in Chapter 7, average processing time was 1.12 ms (tests performed using an Alienware Auroa R6 desktop computer with Intel Core i7-7700 (4 cores, 3.6 GHz) Processor with 32 GB of RAM and an NVIDIA GeForce GTX 1080 Ti GPU with 11 GB GDDR5X standard memory and 3584 CUDA cores). The processing time could have been reduced through graph optimisation. However, the 3D grid format does have the advantage of being easy to update making it suitable for dynamic environments. When the graph is in the grid format, the ID number of the nearest node can be calculated from a 3D location. The occlusion value of each connection can be updated as the environments changes. The processing time for the graph is independent from the number of active sound sources. The data in the graph is used as a lookup table for the individual sounds, instructing them where to place the virtual sound source and which settings to use (occlusion, reverberation, attenuation). The complexity of the graph is not based on the visual detail (polygon count) of the environment. The shape of the graph is instead based on the shape of the occluding objects in the environment and models paths that are open to sound propagation.

8.3 The Spatial Sound Framework as a Research Tool

The rising popularity of video games has seen a recent push towards the application of video game-based technologies to teaching and learning. A serious game can be defined as an interactive computer application, with or without a significant hardware component, that i) has a challenging goal, ii) is fun to play and/or engaging, iii) incorporates some concept of scoring, and iv) imparts to the user a skill, knowledge, or attitude that can be applied to the real world [Bergeron 2006].

Serious games "leverage the power of computer games to captivate and engage players/learners for a specific purpose such as to develop new knowledge or skills" [Corti 2006] and with respect to students, strong engagement has been associated with academic achievement [Shute et al. 2009]. In addition to promoting learning via interaction and engagement, serious games allow users to experience situations that are difficult (even impossible) to achieve in reality due to factors such as cost, time, and safety concerns [Squire & Jenkins 2003]. The spatial sound framework introduced here was originally designed for use in simulations and serious games. In addition to spatial sound rendering, the framework was designed to streamline the game development process by providing a simplified interface or "wrapper" over the low level Fmod API. An early version of this framework was used to provide the audio for a serious game designed to train orthopaedic residents to perform the total knee arthroplasty (TKA) procedure (Figure 8.1) [Cowan et al. 2010].

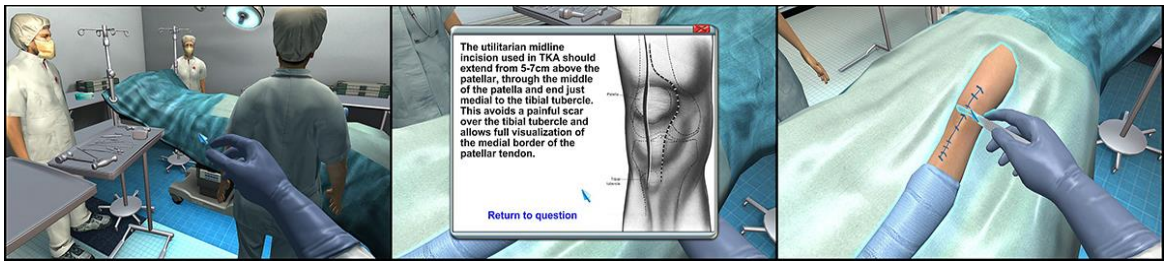


Figure 8.1: Screen captures taken of the total knee arthroplasty serious game.

The TKA serious game was intended to serve as a memory aid to students learning the procedure and to be used in conjunction to other "traditional" learning materials including books and videos. The goal was for the user/trainee to

successfully complete the TKA procedure focusing on the procedure itself (i.e., focusing on the ordering in which steps are performed and on the tools required to perform each step as opposed to the technical aspects of the procedure) while minimizing the time to complete the procedure and maximizing the score (points are either added or taken away based on the trainee's actions).

Sound for the TKA serious game was added simply by including a header file in C++ and adding a folder containing the framework and Fmod. The total implementation time for incorporating spatial sound in the TKA serious game using the framework was approximately 1.5 hours. The sound implementation would likely have added a few days to the development time had the framework not been available.

8.3.1 Multimodal Interaction

In the real world, our senses are flooded with information. We are constantly receiving information from multiple senses (visual, auditory, vestibular, olfactory, proprioceptic), continuously and are able to integrate/process this information and respond immediately. It is currently beyond our capability to faithfully account for all of the human sense within a virtual simulation/serious game, and although the emphasis of (virtual) simulations in general (including serious games) is on the visuals/graphics [Eriksson 2017; Carlile 1996], visuals within such environments are rarely presented in silence but rather, include sound of some type. In the real world, visuals and auditory stimuli do influence one another. Various studies have examined the perceptual aspects of audio-visual cue interaction, and it has been

shown that sound can potentially attract part of the user's attention away from the visual stimuli and lead to a reduced cognitive processing of the visual cues [Mastoropoulou et al. 2005].

An early version of the spatial sound framework was used in studies designed to test the effect that sound has on the perception of visual fidelity. Rojas et al. [2014] conducted a user study to examine the effect of contextual ambient sound on visual fidelity perception. Participants were shown a series of static images depicting a surgeon holding a surgical drill. The surgeon and drill were rendered as a 3D textured mesh. The participants were repeatedly shown the image of the surgeon for a few seconds followed by a screen asking them to rate the image quality on a scale from 1 to 7. Each time the image was shown, one of six texture resolutions were selected at random and applied to the mesh. In addition, randomly selected ambient sound was played while the image was displayed. The six auditory conditions consisted of i) no sound at all, ii) white noise, iii) classical music (Mozart), iv) heavy metal (Megadeth). The three contextual auditory conditions consisted of i) operating room ambiance which included machines beeping, doctors and nurses talking, ii) drill sound, and iii) hospital operating room ambiance coupled with the drill sound. The last three auditory conditions listed above are contextual, meaning that they are relatable to the visual element, the other conditions did not match the context presented visually. When the image of the surgeon was displayed with non-contextual auditory cues (white noise or heavy metal music), a resulting decrease in visual fidelity perception observed. The

operating room ambience resulted in an increase in the perceived visual fidelity, while the inclusion of the drill sound had mixed results.

A follow-up study was conducted to determine the effect that ambient sound has on task performance [Cowan et al. 2015]. The operating room taken from the TKA serious game shown in Figure 8.1 provided a 3D setting. The participants were required to walk around the surgical bed and stationary operating room staff to the tool tray on the other side. Once in range, the participant had to point and click on the tray to compete the task. Upon completion, a screen appeared asking them to rate the visual fidelity of the game on a scale of 1 to 7. Each participant was required to complete this simple task many times. One of four background sound tracks was played each time the participant repeated the task: (1) no sound, (2) white noise, (3) operating room ambiance which included machines beeping, doctors and nurses talking, and (4) hospital operating room ambiance coupled (mixed) with a drill sound. In addition, 1 of 6 blur filters were randomly applied to the scene for each attempt. The results indicated that white noise led to a significant decrease in performance while the sounds that were related to the visual scene improved player performance (reduced the time required to complete the task).

8.4 Contributions

The spatial sound framework introduced here, is capable of providing believable spatial sound for both geometric (polygonal models or voxels) and non-geometric environments (2D, hand painted scenes). The virtual environment can be automatically converted into a 3D graph where geometry is available, or manually

plotted for non-geometric environments. Replacing complex virtual environments with a 3D graph consisting of interconnected nodes, greatly reduces the computational requirements. In addition, each node in the graph can be processed in parallel making use of modern GPU architectures. Some spatial sound APIs such as Resonance Audio approximate reverberation and occlusion based on the listener's immediate surroundings and do not trace the path of sound propagation [Resonance Audio 2019]. Without the simulation of sound propagation, it is not possible to simulate realistic localisation error or occlusion. Graph-based methods using nav-meshes, typically trace a single path representing the shortest route between the listener and the sound source [Beig 2019; Guay 2012]. However, basing the sound source localisation of the shortest path can cause a jarring effect as the shortest path changes due to a moving sound source or listener. These methods are also not capable of providing sound source ambiguity. The spatial sound framework introduced here is capable of providing PBSL with ambiguity if desired. To accomplish this, every node in the graph is populated with both distance and direction data. The result is that every possible path connecting the listener to the sound source is explored, and the perceived direction is based on a weighted average of all available paths. Ray tracing and beam tracing methods can model sound propagation with multiple paths as well. However, the efficiency is negatively affected by geometric complexity, and the computation time increases linearly with each added sound source [Tsingos et al. 2007; Jedrzejewski 2004; Röber et al. 2007; Nvidia 2019a]. The spatial sound framework abstracts away the geometric complexity (polygon count) by replacing the detailed environment with a graph. The graph is fully populated with data and serves as look-up table for

sound sources which allows for an unlimited number of sounds with no additional processing required by the GPU.

The run-time testing detailed in Chapter 7 has demonstrated that a large-scale graph (65,536 nodes and 559,392 by directional connections) can be processed in real-time (1.12 ms per frame) to provide spatial sound (reverberation, occlusion/diffraction, and PBSL) for an unlimited number of sound sources. In addition, the results of the user study detailed in Chapter 7 demonstrated that spatial sound improved the player's ability to navigate the 3D virtual environment over amplitude panning (binaural sound cues only) and ray-cast occlusion (a single ray is cast from the sound source to the listener to determine whether or not the sound is occluded). If realistic spatial sound can be calculated efficiently without the need for specialised hardware, and the rendering of spatial sound can be shown to improve player navigation, then there is no reason to delay the implementation of spatial sound in video games and virtual environments. Game developers do not need to wait for GPU hardware supported ray tracing to become ubiquitous. Spatial sound including propagation modeling can be performed efficiently with current hardware using GPU processed graphs as outlined in Chapter 6.

The spatial sound framework was originally designed for the creation of serious games and simulations. In addition to providing spatial sound, the framework provides all of the standard functionality you would expect from an audio API since it was built on top of the full-featured Fmod sound engine. The framework provides a simplified interface or wrapper that hides the low-level Fmod code in favour of a

set of intuitive C++ classes. This interface was designed to greatly decrease the amount of code a programmer is required to type, reducing the production time and therefore cost. An early version of this framework as already been used to reduce the development time of the TKA serious game. The framework has also been instrumental for conducting research studies relating to multimodal interaction and the effect of contextual sound (related to the scene/task) vs non-contextual sound (not related to the scene/task) on task performance.

The primary focus of this framework has always been the rendering of spatial sound for human listeners. However, the framework is also capable of providing a sense of hearing for any number of NPCs without any additional computation required by the GPU. Using this technology, NPCs could react to sound in a far more realistic manner, even becoming confused and being misled by PBSL. Players would need to take into account the acoustics of the environment when attempting to sneak up on enemy NPCs.

8.5 Future work

Real-time acoustic modeling that is faithful to real-world phenomena is not possible with today's hardware. While accurate acoustic modeling methods do exist, they are computationally intensive and generally not suitable for not real-time applications [Serafin et al. 2018]. A comprehensive list of acoustical modeling techniques is provided by Savioja and Svensson [2015]. Chapter 2 examined some of the most promising real-time acoustic modeling solutions to date, all of which are crude approximations. Likewise, the graph based spatial sound framework

introduced here is an approximation based on replacing the detailed environment that is rendered graphically, with a graph that can be processed efficiently and in parallel on the GPU.

The framework employs several "useful hacks" or shortcuts designed to approximate (sound like) real-world phenomena, while reducing the computational complexity of the system. Many of the formulas presented in Chapter 6 were arrived at through informal listener testing and could be adjusted to improve the realism. Direct comparisons with accurate offline acoustical modeling software packages could be used to calibrate the formulas and variables used by the framework. This calibration could increase the accuracy of the framework without necessarily increasing the computation time.

In addition, an effort should be made to bring real-time spatial sound rendering including propagation modeling to portable devices such as tablets and smart phones, which lack the processing power to perform beam tracing or ray tracing for multiple dynamic sounds. The current version of framework could easily be adapted for portable devices by converting the shader to the GLSL shading language, as well as offering a multi-threaded CPU only version for devices without programmable GPUs.

Despite being an approximation, our sense of hearing is imperfect and subjective. Therefore, the goal of real-time acoustic modeling is not perfection which is unattainable, rather, the goal is to render sound that is perceptually valid. More

extensive psychoacoustic testing must be performed with human listeners to determine whether or not perceptual validity has been reached.

Quantitative testing of system could be accomplished by conducting an additional user study comparing the graph-based spatial sound system introduced here with other spatial sound rendering techniques that also include some form of sound propagation modelling. Instead of testing the system by having the participants play a game, a non-interactive VR walkthrough could be created to allow the participants to focus on the audio without being influenced by their own playing ability or play style. The environment should be based on a commonly encountered real-world setting such as a school or an office building. It should contain hallways and interconnected rooms of different sizes, with and without furnishings. Participants would experience the same walkthrough multiple times with the sound rendered by each of the methods being compared (presented in random order). After each walkthrough, the participants would be asked to rate the quality of the sound.

Appendix A

In this section, the GLSL vertex and fragment shader source code in addition to the source code to implement the scaling filters is provided (the scaling filters are generated using the CPU once during initialization; the scaling filtering is performed on the CPU).

A.1 Occlusion Vertex Shader

GLSL Vertex Shader Code

```
/*
Brent Cowan & Bill Kapralos, June 2009

This shader samples many points between the sound source and listener gathering
information much like a person taking a survey. The data collected is rendered as an
image to be read back from the frame buffer. The image is then filtered (scaled) on the
CPU in order to estimate the occlusion, diffraction, and reverberation present in the
sampled space.
*/

//The distance between the sound source and listener could be calculated on the CPU
more
//efficiently.
//These variables have been provided more for future use.
uniform vec3  soundPos;   //Sound source position xyz
uniform vec3  listenPos;  //Listener position xyz

varying vec3  normal;     //Stores the interpolated surface normal
varying vec3  position;   //position of the object's surface currently being rendered
varying float distance;   //distance between the sound and listener
varying vec3  testVector; //a normalized vector pointing from the listener to the point
on
                          //the object currently being rendered

void main ()
{
        gl_Position = ftransform();
        normal = normalize( gl_NormalMatrix * gl_Normal );
        position = vec3( gl_ModelViewMatrix * gl_Vertex );
        distance = length( listenPos.xyz - soundPos.xyz );

        testVector = normalize(position - vec3( gl_ModelViewMatrix * vec4( listenPos,
1.0 ) ) );
}
```

Algorithm A.1: The reverberation vertex shader code (GLSL).

A.2 Occlusion Fragment Shader

GLSL Fragment Shader Code

```
/*
Brent Cowan & Bill Kapralos, June 2009

This shader samples many points between the sound source and listener gathering
information much like a person taking a survey. The data collected is rendered as an
image to be read back
from the frame buffer. The image is then filtered (scaled) on the CPU in order to
estimate the occlusion, diffraction, and reverberation present in the sampled space.
*/

//The attribute vector contains properties of the object being rendered
//attribute.x = reflective property (currently only used to provide a crude estimate of
//reverberation)
//attribute.y = the objects ability to block sound as opposed to allowing sound to pass
//through it
//attribute.z = not currently used
uniform vec3  attribute;
//objects closer than this distance could block the camera, the occlusion properties of
the
//object are reduced if the distance is below the blockDist variable set by the
application.
uniform float blockDist;

varying vec3  normal;     //Stores the interpolated surface normal
varying vec3  position;   //position of the object's surface currently being rendered
varying float distance;   //distance between the sound and listener
varying vec3  testVector; //a normalized vector pointing from the listener to the point
on
                          //the object currently being rendered


void main ()
{
        //The distance between the sound source and the point on the object currently
being
         //rendered
        float objectDist = length(position);

        //Discard fragment if point is farther than the listener. This effectively
causes the
       //view frustum to have a rounded bottom
        if(objectDist < distance )
        {
                vec3  norm    = normalize(normal); //Object's surface normal

                //We know that the sound source is in front of the surface because the
scene
                //is being rendered from the sound source's position. If the surface
normal
                //where facing away from the sound source, than the surface would not be
                //rendered due to back face culling. If the listener is also in front of
the
                //surface than both the sound and listener are on the same side of the
surface
                //meaning the surface is not blocking the sound. The dot product of the
surface
                //normal and the test vector would return a negative number when the
listener
                //is in front of the surface. This negative number would be replaced by
0 after
                //the max function has been called indicating that this surface is not
                //occluding.
                float block = max(dot(norm, normalize(testVector)), 0.0);

                //block is reduced for objects too close to the source.
```

```
                block = block * min(objectDist / blockDist, 1.0);

                //muffle is a measure of the amount of sound allowed to pass through the
object
                float muffle  = block;
                block         = block * attribute.y;
                float reflect = attribute.x;
                muffle        = muffle - block;

                //The scene begins as a white canvas. Each object rendered has its
colors
                //subtracted so that overlapping objects darken the scene.
                gl_FragColor  = vec4( reflect, block, muffle, 1.0 );
        }
        else
        {
                gl_FragColor = vec4( 0.0, 0.0, 0.0, 1.0 ); //Do not darken the scene
        }
}
```

Algorithm A.2: The reverberation fragment shader code (GLSL).

A.3 Receiver Scaling Filters

C++ Code, Receiver Scaling Filters

```
//Create the scaling filters
int width = 128;
int height = 64;
float *lowFilter  = new float[width*height];
float *highFilter = new float[width*height];
Vector3D v1(float(width)*0.5f, float(height), 0.0f);
Vector3D v2;
for(GLuint w=0; w<width; w++)
{
        for(GLuint h=0; h<height; h++)
        {
                v2.x = float(w); testV.y = float(h*2);
                v2 = v2-v1;
                float_A = v2.GetLength() / (float(width)*0.7071068f);

                //Create the Low Freq. Sin scaling filter
                lowFilter[w + h*width] = cos(float_A*90.0f*DTR);

                //Create the High Freq. Sin scaling filter
                highFilter[w + h*width] = pow(cos(float_A*90.0f*DTR),10.0f);
        }
}
```

Algorithm A.3: The Receiver Scaling Filters (C++ code).

References

Akkas, A. (2019). *Efficient memory and GPU operations for Tiramisu compiler* (Doctoral dissertation, Massachusetts Institute of Technology).

Algazi, V. R., Duda, R. O., Thompson, D. M., & Avendano, C. (2001, October). The CIPIC HRTF database. In *Proceedings of the 2001 IEEE Workshop on the Applications of Signal Processing to Audio and Acoustics (Cat. No. 01TH8575)* (pp. 99-102). IEEE.

Allen, J. B., & Berkley, D. A. (1979). Image method for efficiently simulating small-room acoustics. *The Journal of the Acoustical Society of America*, *65*(4), 943-950.

Angel, E. (2009). *Interactive Computer Graphics* (5th ed.). Boston, MA: Pearson Education Inc.

Antani, L., Chandak, A., Savioja, L., & Manocha, D. (2012). Interactive sound propagation using compact acoustic transfer operators. *ACM Transactions on Graphics (TOG)*, *31*(1), 7.

Antani, L., Chandak, A., Taylor, M., & Manocha, D. (2010). Fast geometric sound propagation with finite edge diffraction. *Technical Report TR10-011, University of North Carolina at Chapel Hill*.

Arrabales, R., Muñoz, J., Ledezma, A., Gutierrez, G., & Sanchis, A. (2013). A machine consciousness approach to the design of human-like bots. In *Believable bots* (pp. 171-191). Springer, Berlin, Heidelberg.

Bălan, O., Moldoveanu, A., Moldoveanu, F., & Dascălu, M. I. (2014, October). Navigational 3D audio-based game-training towards rich auditory spatial representation of the environment. In *2014 18th International Conference on System Theory, Control and Computing (ICSTCC)* (pp. 682-687). IEEE.

Bass, H. E., Sutherland, L. C., & Zuckerwar, A. J. (1990). Atmospheric absorption of sound: Update. *The Journal of the Acoustical Society of America*, 88(4), 2019-2021.

Beig, M., Kapralos, B., Collins, K., & Mirza-Babaei, P. (2019). G-SpAR: GPU-based voxel graph pathfinding for spatial audio rendering in games and VR. In *Proceedings of the IEEE conference on games*, August 20-23, 2019, London, UK.

Bergeron, B. (2006). Developing Serious Games, Hingham, MA. USA: *Thompson Delmar Learning*.

Biot, M. A., & Tolstoy, I. (1957). Formulation of wave propagation in infinite media by normal coordinates with an application to diffraction. The Journal of the Acoustical Society of America, 29(3), 381-391.

Bowling, M., Fürnkranz, J., Graepel, T., & Musick, R. (2006). Machine learning and games. *Machine learning*, *63*(3), 211-215.

Bowman, D. A., & McMahan, R. P. (2007). Virtual reality: how much immersion is enough?. *Computer*, 40(7), 36-43.

Brown, A. D., Stecker, G. C., & Tollin, D. J. (2015). The precedence effect in sound localization. *Journal of the Association for Research in Otolaryngology*, 16(1), 1-28.

Buck I, Foley T, Horn D, Sugerman J, Fatahalian K, Houston M, Hanrahan P (2004) Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics* 23(3):777–786

Calamia, P. T., Svensson, U. P., & Funkhouser, T. A. (2005, August). Integration of edge-diffraction calculations and geometrical-acoustics modeling. In *Proceedings of forum acusticum* (Vol. 2005).

Cambridge English Dictionary. Model [Dictionary definition]. Retrieved July 28, 2019, from https://dictionary.cambridge.org/dictionary/english/model

Candusso, D. (2017, July). Designing Spatial Sound: Adapting Contemporary Screen Sound Design Practices for Virtual Reality. In *SMPTE17: Embracing Connective Media* (pp. 1-10). SMPTE.

Carlile, S. (1996). *Virtual Auditory Space: Generation and Application*. Austin, TX. USA: R. G. Landes Company.

Chandak, A. (2011). Effcient geometric sound propagation using visibility culling. PhD Thesis, Department of Computer Science, University of North Carolina at Chapel Hill, NC, USA.

Computer Hope. What is a Plugin? (2018, November 13). Retrieved September 12, 2019, from https://www.computerhope.com/jargon/p/plugin.htm

Corti, K. (2006). Games-based Learning; a serious business application. PIXE-Learning, 34(6), 1-20.

Cowan, B., & Kapralos, B. (2008). Spatial sound for video games and virtual environments utilizing real-time GPU-based convolution. In *Proceedings of the ACM FuturePlay 2008 International Conference on the Future of Game Design and Technology*. Toronto, Ontario, Canada, November 3-5, 2008.

Cowan, B., & Kapralos, B. (2009a). Real-time GPU-based convolution: A follow-up. In *Proceedings of the ACM FuturePlay 2009 International Conference on the Future of Game Design and Technology*. Vancouver, British Columbia, Canada, May 12-13, 2009.

Cowan, B., & Kapralos, B. (2009b). Real-time acoustical diffraction and first order specular reflection modeling using the GPU. In *Proceedings of the 10th Western Pacific Acoustics Conference*. Beijing, China, September 21-23, 2009.

Cowan, B., & Kapralos, B. (2010). GPU-based real-time acoustical occlusion modeling. *Virtual reality*, *14*(3), 183-196.

Cowan, B., & Kapralos, B. (2011a, February). GPU-based acoustical diffraction modeling for complex virtual reality and gaming environments. In *Audio Engineering Society Conference: 41st International Conference: Audio for Games*. Audio Engineering Society. London, UK.

Cowan, B., & Kapralos, B. (2011b, September). GPU-based acoustical occlusion modeling with acoustical texture maps. In *Proceedings of the 6th Audio Mostly Conference: A Conference on Interaction with Sound* (pp. 55-61). ACM.

Cowan, B., & Kapralos, B. (2011c). A GPU-based method to approximate acoustical reflectivity. *Journal of Graphics, GPU, and Game Tools*, *15*(4), 210-215.

Cowan, B., & Kapralos, B. (2013a, September). GPU-based acoustical occlusion modeling for virtual environments and games. In *2013 IEEE International Games Innovation Conference (IGIC)* (pp. 48-50). IEEE.

Cowan, B., & Kapralos, B. (2013b, July). Spatial sound rendering for dynamic virtual environments. In *2013 18th International Conference on Digital Signal Processing (DSP)* (pp. 1-6). IEEE.

Cowan, B., & Kapralos, B. (2015, July). Interactive rate acoustical occlusion/diffraction modeling for 2D virtual environments & games. In *2015 6th International Conference on Information, Intelligence, Systems and Applications (IISA)* (pp. 1-6). IEEE.

Cowan, B., Rojas, D., Kapralos, B., Moussa, F., & Dubrowski, A. (2015). Effects of sound on visual realism perception and task performance. *The Visual Computer*, *31*(9), 1207-1216.

Cowan, B., Sabri, H., Kapralos, B., Porte, M., Backstein, D., Cristancho, S., & Dubrowski, A. (2010). A serious game for total knee arthroplasty procedure, education and training. *Journal of CyberTherapy & Rehabilitation (JCR)*, *3*(3), 285-298.

Cremer, L., & Müller, H. A. (1978). *Die wissenschaftlichen grundlagen der raumakustik*. Stuttgart: Hirzel.

Davies, A. C. (2015). Stereo Sound Recording and Reproduction? Remembering the History. *IEEE Signal Processing Magazine*, 32(5), 14-16.

Davis, R. J., & Stephens, S. D. G. (1974). The effect of intensity on the localization of different acoustical stimuli in the vertical plane. *Journal of Sound and Vibration*, 35(2), 223-229.

Deo, N. (2017). *Graph theory with applications to engineering and computer science*. Courier Dover Publications.

Doelle, L. L. (1972). *Environmental acoustics*. McGraw-Hill.

Ebersole, M. (Sep 10, 2012). What is CUDA? Available from: http://blogs.nvidia.com/2012/09/what-is-cuda-2/ Retrieved: April 6, 2013.

Ekman, M., Warg, F., & Nilsson, J. (1994). An in-depth look at computer performance growth, *Computer Architecture News* 33(1), 144–147.

Eriksson, P. R. (2017). A Comparison Of Two Commercially Available Alternatives For Spatializing Audio Over Headphones In A Game Setting.

Fang, J., Varbanescu, A. L., & Sips, H. (2011, September). A comprehensive performance comparison of CUDA and OpenCL. In *2011 International Conference on Parallel Processing* (pp. 216-225). IEEE.

Fialka, O., & Cadik, M. (2006, July). FFT and convolution performance in image filtering on GPU. In *Tenth International Conference on Information Visualisation (IV'06)* (pp. 609-614). IEEE.

Fmod. Audio API Documentation. Retrieved October 10, 2019, from https://www.fmod.com/resources/documentation-api?version=1.10&page=content/generated/overview/3dsound.html

Foley, J. D., Van Dam, A., Feiner, S. K., Hughes, J. F., & Phillips, R. L. (1994). *Introduction to computer graphics* (Vol. 55). Reading: Addison-Wesley.

Funkhouser, T., Carlbom, I., Elko, G., Pingali, G., Sondhi, M., & West, J. (1998, July). A beam tracing approach to acoustic modeling for interactive virtual environments. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques* (pp. 21-32). ACM.

Funkhouser, T., Min, P., & Carlbom, I. (1999, July). Real-time acoustic modeling for distributed virtual environments. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques* (pp. 365-374). ACM Press/Addison-Wesley Publishing Co.

Funkhouser, T., Tsingos, N., Carlbom, I., Elko, G., Sondhi, M., & West, J. (2002, September). Modeling sound reflection and diffraction in architectural environments with beam tracing. Forum Acusticum, Seville, Spain. pp. 8, The 3rd EAA European Congress on Acoustics. <inria-00606724>

Funkhouser T, Tsingos N, Carlbom I, Elko G, Sondhi M, West JE, Pingali G, Min P, Ngan A (2004) A beam tracing method for interactive architectural acoustics. J Acoust Soc Am 115(2):739–756

Gallos, E., & Tsingos, N. (2003). Efficient 3D audio processing with the GPU. In *Proceedings of the ACM International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 2003)*, San Diego, CA. USA, July 27–31 2003, pp. 896–907.

Gardner, W. G. (1995). Efficient convolution without input-output delay. *Journal of the Audio Engineering Society* 43(3), 127-136.

Geer, D. (2005). Taking the graphics processor beyond graphics, *IEEE Computer* 39(9), 14–16.

Giguere, C., Abel, S.M. (1993). Sound localization: effects of reverberation time, speaker array, stimulus frequency, and stimulus rise/decay. *Journal of the Acoustical Society of America*, 94 (2) , pp. 769-776.

Guay, J. F. (2012, March). Real-time sound propagation in video games. In *Game Developers Conference (GDC)| GDC Vault* (pp. 5-9).

Hakala, M. (2019). Synthesis of Spatially Extended Sources in Virtual Reality Audio. *Aalto University*.

Hamidi, F., & Kapralos, B. (2009). A review of spatial sound for virtual environments and games with graphics processing units. *The Open Virtual Reality Journal*. 1(1), 8-17.

Haraldsen, S. (2010). Sound propagation: Bringing audio to a higher level. *Game Developer Magazine*, pp. 32-35, 2010.

Hecht, E. (2002). *Optics*, fourth ed., Pearson Education Inc., San Francisco, CA. USA.

Heckbert, P. S. (1986). Survey of texture mapping. *IEEE computer graphics and applications*, *6*(11), 56-67.

Hillesland, K., & Lastra, A. (2004, August) GPU floating-point paranoia. In: Proceedings of ACM workshop on general purpose computing on graphics processors 318, Los Angeles, California, USA.

Hu, L., Che, X., & Zheng, S. Q. (2016). A closer look at gpgpu. *ACM Computing Surveys (CSUR)*, 48(4), 60.

IEEE (1987, February) IEEE standard for binary floating-point arithmetic. ACMSIGPLAN Notices 22, 2 (Feb.), 9–25

Intel (2018, December). 6th Gen Intel® Core™ X-Series Processor Family Datasheet, Vol. 1. Available At: https://www.intel.ca/content/www/ca/en/products/processors/core/6th-gen-x-series-datasheet-vol-1.html

International Organization for Standardization (ISO) (1993). ISO 9613-1: 1993, Acoustics—Attenuation of sound during propagation outdoors—Part 1: Calculation of the Absorption of Sound by the Atmosphere.

James, P. (2016, May 09). Nvidia's VRWorks Audio Brings Physically Based 3D GPU Accelerated Sound – Road to VR. Retrieved from https://www.roadtovr.com/nvidias-vrworks-audio-brings-physically-based-3d-gpu-accelerated-sound/

Jedrzejewski, M. (2004). *Computation of room acoustics on programmable video hardware*, Master's Thesis, Polish-Japanese Institute of Information Technology, Poland.

Jensen, HW (2001) Realistic image synthesis using photon mapping. A. K. Peters, Ltd., Natick

Kapralos, B., Jenkin, M. R., & Milios, E. (2003). Auditory perception and spatial (3d) auditory systems. *Department of Computer Science, York University, Tech. Rep. CS-2003-07.*

Kapralos, B., Jenkin, M., & Milios, E. (2008) Sonel mapping: a probabilistic acoustical modeling method. Building Acoust 15(4), 289–313

Kapralos, B., & Mekuz, N. (2007, June). Application of dimensionality reduction techniques to HRTFs for interactive virtual environments. In *Proceedings of the ACM Advancements in Computer Entertainment (ACE 2007)*, Salzburg, Austria.

Kastbauer, D. (2010, January 28) The next big steps in game sound design. In Gamasutra, http://www.gamasutra.com/. [Accessed on: July 20, 2019].

Kay, R. (2018, November 28). The future of 2D gaming. Retrieved October 25, 2019, from https://www.gamesindustry.biz/articles/2018-11-27-the-future-of-2d-gaming.

Keller, J. B. (1962). Geometrical theory of diffraction. *Journal of the Optical Society of America*, 52(2), 116-130.

Kelly, I., Gorzel, M., & Güngörmüsler, A. (2017, March). Efficient externalized audio reverberation with smooth transitioning. Technical Disclosure Commons.

Kistler, D. J., & Wightman, F. L. (1992). A model of head-related transfer functions based on principle components analysis and minimum phase reconstruction. Journal *of the Acoustical Society of America*, 91(3), 1637-1647.

Kleiner, M., Dalenbäck, B., & Svensson, P. (1993). Auralization – an overview. Journal of *the Audio Engineering Society*, 41(11), 861-875.

Krüger, J., & Westermann, R. (2003, July). Linear algebra operators for GPU implementation of numerical algorithms. In *ACM Transactions on Graphics (TOG)* (Vol. 22, No. 3, pp. 908-916). ACM.

Kuttrff, H. (2000) Room acoustics, 4th edn. Spon Press, London.

Kuzminski, A. (August 2016) VR Audio: Trends and Challenges of Pioneering a New Field. Retrieved January 11, 2020, from http://designingsound.org/2016/08/vr-audio-trends-and-challenges-of-pioneering-a-new-field/

Lakka, E., Malamos, A. G., Pavlakis, K. G., & Ware, J. A. (2018). Spatial Sound Rendering–A Survey. *International Journal of Interactive Multimedia and Artificial Intelligence*, 5(3), 33-45.

Laukkanen, S. (2018). Post-processing In video games. Turku University of Applied Sciences.

Li, Y., Driessen, P. F., Tzanetakis, G., & Bellamy, S. (2006, August). Spatial sound rendering using measured room impulse responses. In *2006 IEEE International Symposium on Signal Processing and Information Technology* (pp. 432-437). IEEE.

Lokki, T., Svensson, P., & Savioja, L. (2002, June). An efficient auralization of edge diffraction. In *Audio Engineering Society Conference: 21st International Conference: Architectural Acoustics and Sound Reinforcement. Audio Engineering Society*.

Luebke, D., & Humphreys, G. (2007). How GPUs work, *IEEE Computer* 40(2), 96–100.

MacPherson, E. A., & Middlebrooks, J. C. (2000). Localization of brief sounds: effects of level and background noise. *The Journal of the Acoustical Society of America*, 108(4), 1834-1849.

Makino, H., Ishii, I., & Nakashizuka, M. (1996, October). Development of navigation system for the blind using GPS and mobile phone combination. In *Proceedings of 18th annual International Conference of the IEEE Engineering in Medicine and Biology society* (Vol. 2, pp. 506-507). IEEE.

Makous, J. C., & Middlebrooks, J. C. (1990). Two-dimensional sound localization by human listeners. *The journal of the Acoustical Society of America*, 87(5), 2188-2200.

Mansbridge, S., Finn, S., & Reiss, J. D. (2012, October). An autonomous system for multitrack stereo pan positioning. In *Audio Engineering Society Convention 133*. Audio Engineering Society.

Mark, W. R., Glanville, P. S., Akeley, K., & Kilgard, M. J. (2003, July). Cg: a system for programming graphics hardware in a C-like language, In *Proceedings of the ACM International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 2003)*, San Diego, CA. USA, (pp. 896–907).

Mastoropoulou, G., Debattista, K., Chalmers, A., & Troscianco, T. (2005, August). The influence of sound effects on the perceived smoothness of rendered animations. In *Proceedings of the 2nd Symposium on Applied Perception in Graphics and Visualization*, La Coruña, Spain, (pp. 9–15).

Matsuyama, K., Fujimoto, T., & Chiba, N. (2007, October). Real-time sound generation of spark discharge. In *15th Pacific Conference on Computer Graphics and Applications (PG'07)* (pp. 423-426). IEEE.

Mehra, R., Rungta, A., Golas, A., Lin, M., & Manocha, D. (2015). Wave: Interactive wave-based sound propagation for virtual environments. *IEEE transactions on visualization and computer graphics*, 21(4), 434-442.

Mehta, M., Johnson, J., & Rocafort, J. (1999) Architectural acoustics principles and design. Prentice Hall Inc., Upper Saddle River

Menshikov (2003) Modern audio technologies in games. Retrieved November 28, 2018 from http://ixbtlabs.com/articles2/sound-technology/index.html

Middlebrooks, J. C., & Green, D. M. (1991). Sound localization by human listeners. *Annual review of psychology, 42*(1), 135-159.

Minecraft, official website. Retrieved December 12, 2018, from https://minecraft.net/en-us/

Moeller, R. M., Esplin, B., & Conway, S. (2009). Cheesers, pullers, and glitchers: The rhetoric of sportsmanship and the discourse of online sports gamers. *Game Studies*, 9(2).

Murphy, D. T., & Beeson, M. J. (2003, June). Modelling spatial sound occlusion and diffraction effects using the digital waveguide mesh. In *Audio Engineering Society Conference: 24th International Conference: Multichannel Audio, The New Reality*. *Audio Engineering Society*.

Murphy, D., & Neff, F. (2011). Spatial sound for computer games and virtual reality. In Game sound technology and player interaction: *Concepts and developments* (pp. 287-312). IGI Global.

National Institute on Deafness and Other Communication Disorders NIDCD). (2010, June 7). I Love What I Hear! Common Sounds. Retrieved July 23, 2016, from https://www.nidcd.nih.gov/health/i-love-what-i-hear-common-sounds

Ni, T. (2009, October). Direct Compute: Bring GPU computing to the mainstream. *In GPU Technology Conference* (p. 23).

Nickolls, J., Buck, I., & Garland, M. (2008, August). Scalable parallel programming. In *2008 IEEE Hot Chips 20 Symposium (HCS)* (pp. 40-53). IEEE.

Nordahl, R., Turchet, L., & Serafin, S. (2011). Sound synthesis and evaluation of interactive footsteps and environmental sounds rendering for virtual reality applications. *IEEE transactions on visualization and computer graphics*, *17*(9), 1234-1244.

Nvidia Corporation: Developer Resources. Retrieved July 15, 2019, from https://developer.nvidia.com/vrworks/vrworks-audio

Nvidia Corporation: NVIDIA CUDA Programming Guide, version 1.1. Technical report (November 2007)

Nvidia (2019, March 26). Retrieved April 5, 2019, from https://developer.nvidia.com/cuda-zone

Nvidia, (2008). CUDA Programming Guide Version 2.0, July 2008.

OpenCL Official website. Retrieved April 6, 2013, from http://www.khronos.org/opencl/

Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., & Purcell, T. J. (2007, March). A survey of general-purpose computation on graphics hardware. In *Computer graphics forum* (Vol. 26, No. 1, pp. 80-113). Oxford, UK: Blackwell Publishing Ltd.

Parfit, D. (2005). *Interactive Sound, Environment, and Music Design for a 3D Immersive Video Game* (Doctoral dissertation, New York University).

Pausch, R., Proffitt, D., & Williams, G. (1997). Quantifying immersion in virtual reality.

Plack, C. J. (2018). The Sense of Hearing (3rd edition), Routledge, London, UK.

Pulkki, V., Delikaris-Manias, S., & Politis, A. (Eds.). (2018). *Parametric time-frequency domain spatial audio*. John Wiley & Sons, Incorporated.

Qian, Y. Y., & Teather, R. J. (2017, October). The eyes don't have it: an empirical comparison of head-based and eye-based selection in virtual reality. In *Proceedings of the 5th Symposium on Spatial User Interaction* (pp. 91-98). ACM.

Rayleigh, L. (1907). XII. On our perception of sound direction. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, *13*(74), 214-232.

Reference for Business. Retrieved October 6, 2019, from https://www.referenceforbusiness.com/history2/64/NVIDIA-Corporation.html

Resonance Audio. Discover Resonance Audio. Retrieved July 25, 2019, from https://resonance-audio.github.io/resonance-audio/discover/overview

Ribeiro, F., Ba, D., Zhang, C., & Florêncio, D. (2010, July). Turning enemies into friends: Using reflections to improve sound source localization. In *International Conference on Multimedia and Expo* IEEE (ICME) (pp. 731-736).

Röber, N., Kaminski, U., & Masuch, M. (2007, September). Ray acoustics using computer graphics technology. In *Proceedings of the 10th International Conference on Digital Audio Effects*. Bordeaux, France.

Röber, N., Spindler, M., & Masuch, M. (2006, November). Waveguide-based room acoustics through graphics hardware. In *Proceedings of the International Computer Music Conference 2006*. New Orleans, LA. USA.

Rojas, D., Cowan, B., Kapralos, B., Collins, K., & Dubrowski, A. (2014, February). The effect of contextual sound cues on visual fidelity perception. In *Medicine Meets Virtual Reality* (pp. 346-352).

Rost, R. (2006). OpenGL Shading Language, second ed., Addison-Wesley Professional, Boston, MA. USA.

Sarkar, S. (February 27, 2017). Minecraft sales hit 122M copies. Polygon. Retrieved September 20, 2018, from https://www.polygon.com/2017/2/27/14755644/minecraft-sales-122m-copies

Savioja, L., & Svensson, U. P. (2015). Overview of geometrical room acoustic modeling techniques. *The Journal of the Acoustical Society of America*, *138*(2), 708-730.

Schaeffer, J. (2001). A gamut of games. *AI Magazine*, *22*(3), 29-29.

Serafin, S., Geronazzo, M., Erkut, C., Nilsson, N. C., & Nordahl, R. (2018). Sonic interactions in virtual reality: state of the art, current challenges, and future directions. *IEEE computer graphics and applications*, 38(2), 31-43.

Sherrod, A. (2008). Game graphics programming, Course Technology, Cengage Learning, Boston, MA USA, 2008.

Shinn-Cunningham, B. G. (2000, December). Distance cues for virtual auditory space. In *Proceedings of the IEEE-PCM* (Vol. 2000, pp. 227-230).

Shtrepi, L., Astolfi, A., Pelzer, S., Vitale, R., & Rychtáriková, M. (2015). Objective and perceptual assessment of the scattered sound field in a simulated concert hall. The *Journal of the Acoustical Society of America*, 138(3), 1485-1497.

Shute, V. J., Ventura, M., Bauer, M., & Zapata-Rivera, D. (2009). Melding the power of serious games and embedded assessment to monitor and foster learning. *Serious games: Mechanisms and effects*, *2*, 295-321.

Slattery III, W. H., & Middlebrooks, J. C. (1994). Monaural sound localization: acute versus chronic unilateral impairment. *Hearing research*, 75(1-2), 38-46.

Smith, S. W. (1997). The scientist and engineer's guide to digital signal processing, California Technical Publishing, San Diego, California.

Squire, K., & Jenkins, H. (2003). Harnessing the power of games in education. *Insight*, *3*(1), 5-33.

Svensson, U. P., Fred, R. I., & Vanderkooy, J. (1999). An analytic secondary source model of edge diffraction impulse responses. *The Journal of the Acoustical Society of America*, 106(5), 2331-2344.

Tatarchuk, N. (2009, August). Advances in real-time rendering in 3D graphics and games I. In *ACM SIGGRAPH 2009 Courses* (p. 4). ACM.

Techopedia. What is a Sprite? - Definition from Techopedia. Retrieved August 12, 2019, from https://www.techopedia.com/definition/2046/sprite-computer-graphics

Torres, R. R., Svensson, P., and Kleiner, M. Computation of edge diffraction for more accurate room acoustics auralization. *Journal of the Acoustical Society of America 109*, 2 (2001), 600–610.

Tsingos, N., Dachsbacher, C., Lefebvre, S., & Dellepiane, M. (2007, June). Instant sound scattering. In *Proceedings of the 18th Eurographics conference on Rendering Techniques* (pp. 111-120). Eurographics Association.

Tsingos, N., Funkhouser, T., Ngan, A., & Carlbom, I. (2001, August). Modeling acoustics in virtual environments using the uniform theory of diffraction. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (pp. 545-552). ACM.

Tsingos, N., & Gascuel, J. D. (1997, May). Soundtracks for computer animation: sound rendering in dynamic environments with occlusions. In *Proceedings of Graphics Interface '97* (Kelowna, BC. Canada), 9–16.

Tsingos, N., & Gascuel, J. D. (1998, May). Fast rendering of sound occlusion and diffraction effects for virtual acoustic environments. In 104th *Audio Engineering Society* (Amsterdam, The Netherlands), 1–14.

von Tycowicz, C., & Loviscach, J. (2008). A malleable drum. In *Proceedings of the 35th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 2008 Posters)*, Los Angeles, CA. USA, August 11-15 2008, Article No. 74.

Ubisoft Rainbow Six Siege Game Information Page. Retrieved October 6, 2019 from: https://rainbow6.ubisoft.com/siege/en-ca/game-info/index.aspx

Välimäki, V., Parker, J., Savioja, L., Smith, J. O., & Abel, J. (2016, January). More than 50 years of artificial reverberation. In *Audio Engineering Society Conference: 60th International Conference: DREAMS (Dereverberation and Reverberation of Audio, Music, and Speech)*.

Walton, J. (2019). AMD Radeon RX 5700 and RX 5700 XT review in progress. [online] Retrieved July 8, 2019 from https://www.pcgamer.com/amd-radeon-rx-5700-and-rx-5700-xt-review-in-progress/

Wang, N., & Pan, J. (2018, November). Time-domain modelling of boundary reflection using an acoustic wave propagator and image source method. In *Proceedings of ACOUSTICS* (Vol. 7, No. 9).

Whalen, S. (2005). Audio and the graphics processing unit. Author report, University of California, Davis, California, USA.

Wilde, M. D. (2004). *Audio Programming for Interactive Games: The Computer Music of Games*. Focal press.

Xiao, Y., Zhao, Q., Kaku, I., & Xu, Y. (2012). Development of a fuel consumption optimization model for the capacitated vehicle routing problem. *Computers & Operations Research*, 39(7), 1419-1431.

Zannini, C. M., Parisi, R., & Uncini, A. (2011, July). Binaural sound source localization in the presence of reverberation. In *the 17th International Conference on Digital Signal Processing (DSP)* (pp. 1-6). IEEE.

Zhang, W., Samarasinghe, P. N., Chen, H., & Abhayapala, T. D. (2017). Surround by sound: A review of spatial audio recording and reproduction. *Applied Sciences*, 7(5), 532.

Zhou, Z., Cheok, A. D., Qiu, Y., & Yang, X. (2007). The role of 3-D sound in human reaction and performance in augmented reality environments. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, *37*(2), 262-272.

Zurek, P. M. (1987). The precedence effect. In *Directional hearing* (pp. 85-105). Springer, New York, NY.