Individual Project Report

Matthew Reuben Thomas U1154964

Introduction

The goal of this project was to produce a piece of software which could efficiently synthesize high-quality impulse responses for virtual environments. The software had to run on commodity hardware, available to most musicians, while also producing high-quality output at a reasonable speed. In addition, the software had to be capable of rendering impulses suitable for basic auralisation, using head-related transfer-function (HRTF) based techniques.

Literature Review

Many different implementations of modelling reverbs exist. Some of these are commercially available, for example the Odeon family of room acoustics software (discussed in Rindel (2000)), and the Catt-Acoustic program (Dalenbäck (2010)). Many more exist only as research projects without commercial implementations - the programs described by Röber, Kaminski, & Masuch (2007), Savioja, Lokki, & Huopaniemi (2002), Schissler, Mehra, & Manocha (2014) and Taylor et al. (2012) (to name a few) do not appear to be available in any form to the general public.

Physically-based reverb algorithms can largely be divided into two categories, namely wave-based and geometrical algorithms, each of which can be divided into sub-categories.

Savioja et al. (2002) notes that wave-based methods include the *finite element method*, *boundary element method*, and the *finite-difference time-domain model*. These methods produce very accurate results at specific frequencies, however they are often very costly, especially at higher frequencies.

Geometric methods may be stochastic or deterministic, and are usually based on some kind of ray-casting. In these models, sound is supposed to act as a ray rather than a wave. This representation is well suited to high frequency sounds, but is unable to take the sound wavelength into account, often leading to overly accurate higher-order reflections (Rindel (2000)). These methods also often ignore wave effects such as interference or diffusion.

The most common stochastic method is ray tracing, which has two main benefits: Firstly, an implementation can take influences from the extensive body of research into graphical ray tracing methods. Secondly, ray tracing is an ‘embarrassingly parallel’ algorithm, meaning that it can easily be distributed across many processors simultaneously, as there is no need for signalling between algorithm instances, and there is only a single ‘final gather’ (Stephens (2013)).

A common deterministic method is the image-source method. This algorithm is conceptually very simple, and therefore fairly straightforward to implement. However, it quickly becomes very expensive, with complexity proportional to the number of primitives in the scene raised to the power of the number of reflections. For scenes of a reasonable size, a great number of calculations are required, making exclusively image-source based methods impractical for most purposes.

Many systems implement ‘hybrid’ algorithms, which may combine any of the above methods. Combining wave-based and geometric methods has an obvious benefit - the wave-based simulation can be used to generate a low-frequency response, while a geometric method can generate the higher-frequency portion. Similarly, combining ray tracing and image-source modelling allows for the use of accurate image-source simulation for just the early reflections, with faster stochastic modelling of later reflections.

The majority of the systems mentioned are real-time - the programs mentioned by Röber et al. (2007), Schröder (2011), Schissler et al. (2014), Savioja et al. (2002), and Taylor et al. (2012), along with the Catt-Acoustic software, are all capable of producing real-time auralisations. Off line renderers also exist. The Odeon software and the system described by B. Kapralos, Jenkin, & Milios (2004) are both examples of off line renderers.

Real-time systems play an important role in audio engines for interactive games or virtual-reality systems, where it is often necessary for virtual environments to provide realistic sonic cues to the interacting user. However, such systems must be capable of producing impulse responses very rapidly, at least tens of times per second. Savioja et al. (2002) and B. Kapralos et al. (2004) note that it is generally only possible to use geometric methods to simulate early reflections at real-time rates, as modelling later reflections becomes too computationally expensive. Schissler et al. (2014) mentions that in benchmarks for their real-time system, they traced 1000 rays from each source to an arbitrary depth of around 10 reflections. Cheng (2014) describes a real-time geometric system for modelling sound in 3D games, which is capable of rendering 4096 rays to a depth of 3 reflections in real-time.

In ray tracing, as with any other stochastic modelling method, better quality is achieved primarily through taking more samples. Clearly, the constraints that real-time processing places upon auralisation methods mean that impulses generated in this way will be inaccurate, and likely of insufficient quality for music production.

The final notable difference between implementations is to do with hardware rather than software. Many modern implementations of ray tracing algorithms take advantage of the computer's graphics processing unit (GPU), which provides a huge number of discrete processors that can run the same algorithm in parallel.

Though the processors are individually much less powerful than the typical central processing unit (CPU), great speed-ups are possible due to the sheer number of processes that can be run simultaneously. Ray tracing algorithms have the added benefit that most of the data supplied to the algorithm is 'static' - all the information about the scene, such as geometry, source position, microphone position, and materials can be reused for each ray that is fired. The only variant parameter is the initial ray direction. This means that the static data can be copied to graphics memory once, and then referred to by every ray tracer instance. Due to the popularity of ray tracing techniques in graphics, a variety of papers are available concerning the implementation of ray tracers on graphics hardware. Hradský (2011) describes general ray tracing techniques on the GPU, while Röber et al. (2007), Cheng (2014) and Taylor et al. (2012) all use the GPU specifically for ray tracing-based auralisation.

On the other hand, many ray tracing implementations just run on the CPU. It is still possible to achieve reasonable speed-ups over naive implementations by using single-instruction multiple-data (SIMD) instructions or multiple threads.

Advantages of this approach are mostly to do with tooling. GPU code must be written in a speciality language - 'general purpose' options include Cuda and OpenCL, though Cuda code will only run on Nvidia GPUs. Alternatively, a graphics shader language such as GLSL can be coerced into interacting with non-graphical data. This is certainly possible - Cowan & Kapralos (2008) uses the OpenGL Shading Language to implement a fast parallel one-dimensional convolution on the GPU. However, the shader program can only interact with data in 8-bit-channels, leading to both space and time inefficiencies. All GPU languages are speciality languages, and therefore often have very sparse programming environments. In general, much better tools are available for working with and debugging CPU code than GPU code. Additionally, every platform is guaranteed to have a CPU on which the program can run, but not every system will have a compatible GPU.

Disadvantages of using the CPU include the complexity of writing efficient threaded routines or SIMD-optimized assembly, and that SIMD code written for one processor family will generally not run on another. Schissler et al. (2014) uses SIMD-optimized code on the CPU to implement their sound propagation simulator, and Dammertz, Hanika, & Keller (2008) uses SIMD for fast graphical ray tracing.

Implementation

Basic Algorithm Choices

For this project, an off line ray tracing algorithm was chosen, which was later adapted into a hybrid ray tracing/image-source model. The costliest and 'most parallel' parts of the algorithm were kept on the GPU where possible, doing only mix-down and post-processing on the CPU.

The off line approach was taken in the hope that the lack of time constraints on processing would lead to higher quality outputs, of a suitable quality for musical applications.

A geometric approach was selected over a wave-based method, due to the relative simplicity of implementing geometric algorithms versus wave-based ones, especially in parallel.

Rather than trace the first few reflections of each ray, it was hoped that the entirety of each ray's path would be traced, influencing the choice of GPU over the CPU. If each ray takes a very short amount of time to trace, then the CPU is fast enough, relative to the GPU, that the time taken to trace short numbers of rays sequentially is comparable to the that taken to trace large numbers of rays in parallel. (Use of the GPU incurs an additional time penalty, as data must be copied to and from graphics memory.) However, if each ray takes longer to trace, the overhead of copying to graphics memory becomes negligible, and substantial speed-ups can be gained by running several traces slowly in parallel rather than quickly in sequence.

Algorithm Description

For a detailed description of the algorithm, check the parallel_raytrace user manual at https://github.com/reuk/parallel-reverb-raytracer/blob/master/cmd/parallel_raytrace.1.md.

The final algorithm is a hybrid of image-source and ray tracing. Figure 1 serves as a simple introduction to this algorithm. The standard way of checking for image source contributions is to reflect the microphone position in every possible combination of primitives up to a certain maximum number of reflections, and then to check for line-of-sight between the source and the microphone image. This quickly becomes very costly, requiring f^r line-of-sight tests when f is the number of surfaces in the model, and r is the maximum reflection depth. Additionally, the majority of the tested paths will not be valid in most scenes.

The hybrid algorithm merges the ray tracing and image-source steps, only checking for line-of-sight with combinations of primitives that have already been struck by rays. This restricts the image-source tests to paths that might plausibly provide an image-source contribution. If the path is valid for a set of perfect specular reflections, there's a much greater chance that the path will also be valid for image-source contributions.

Of course, because the initial ray directions are random, there is a chance that not all valid image-source paths will be found, but there's also a good chance that all the most meaningful/obvious image-source paths will be.

Some rays may start in similar (but not identical) directions, intersecting with the same surfaces in the same order, and leading to the same image-source contribution. To ensure that the same image-source path is not counted twice, each ray keeps track of the surfaces it has been reflected from, and a post-processing step checks and discounts duplicates.

Calculating image-source contributions in this way is still relatively costly, compared to basic ray tracing, so in this implementation only reflections up to the tenth order are tested, but this seems sufficient when combined with the diffuse contributions from the pure ray tracing.

For each contribution, diffuse or image-source, the total distance that the ray has travelled between the source and the microphone is stored, and used to calculate the time at which an impulse fired from the source would be received at the microphone. The volume of this impulse is calculated depending on the materials of the surfaces that the ray has struck, the attenuation due to air, and the angle at which the ray is reflected.

Technologies and Libraries

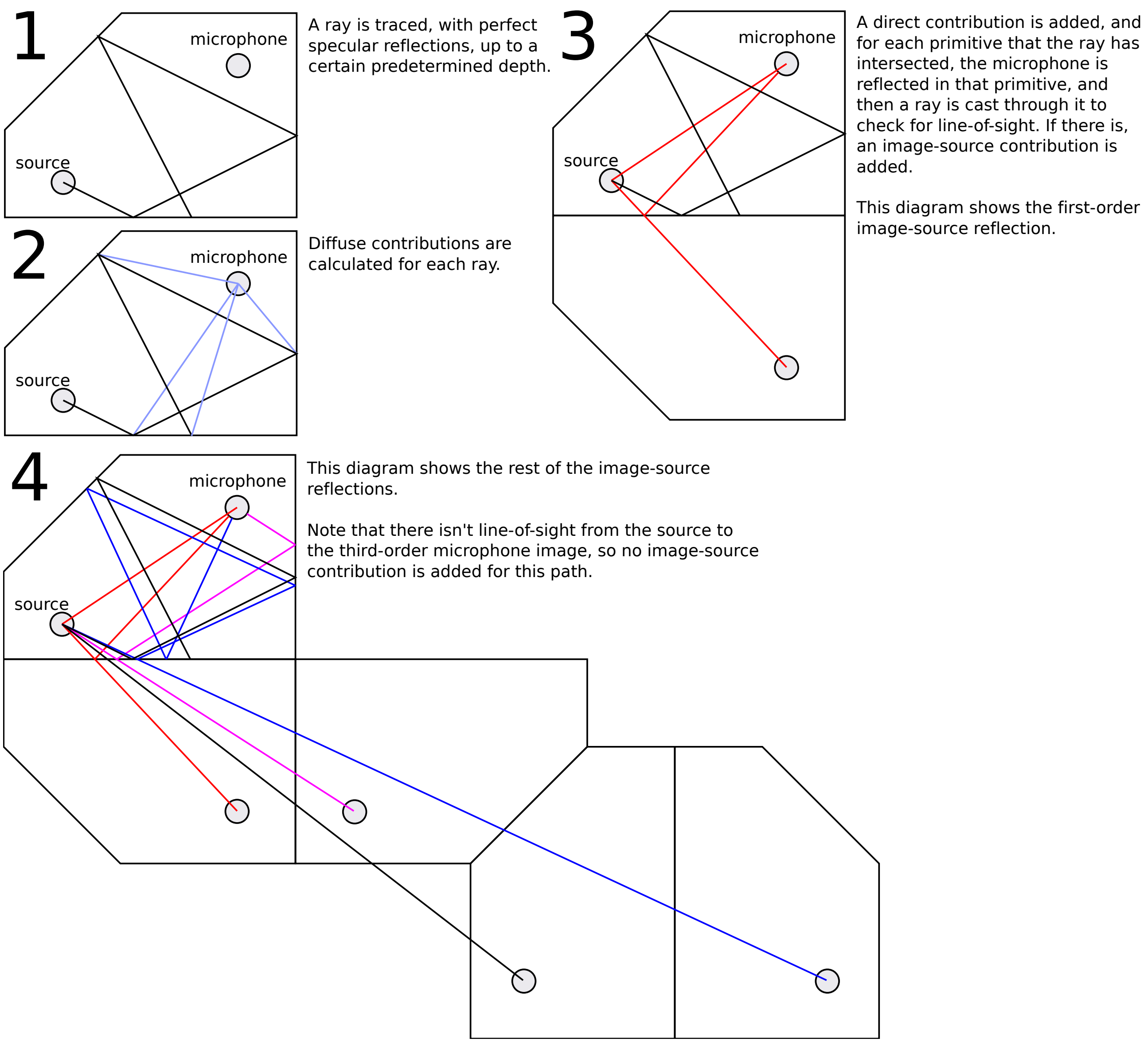
The code for the project was written mostly in C++, taking advantage of many modern C++ features. C++ was chosen as it marked an acceptable compromise between performance and ease of development. The majority of the project was written for the C++11 standard, which provides many features that make code simpler, faster to write, and in many cases more efficient too. Towards the end of the project, the project was moved to C++14, which is supported on newer versions of OS X. This standard additionally provides generic lambda expressions, which make it easier to write flexible, reusable code.

All GPU code was written for OpenCL 1.2, which was chosen for its increased portability over CUDA (which only runs on Nvidia hardware), and its 'general purpose-ness' - OpenCL is designed for general-purpose computation, whereas shader-language alternatives are designed specifically for graphics processing. The OpenCL interface is specified in C, but Khronos (the standards group responsible for OpenCL) also provide a C++ 'wrapper' API, which automates the manual memory management requirement of the C interface.

Only one deployment target was chosen: OS X. OS X provides recent C++ compilers and libraries by default, and in recent versions provides an OpenCL framework as a component of the operating system. In contrast, both Linux and Windows require additional libraries to be installed in order to run OpenCL code.

Several additional libraries were used to speed the development of the program:

The Assimp library was used for loading and processing of 3D model files. This library loads a variety of formats into a single universal data structure which can be queried in a consistent way. This allows the program to support a wide range of input formats.



A ray is traced, with perfect specular reflections, up to a certain predetermined depth.

Diffuse contributions are calculated for each ray.

A direct contribution is added, and for each primitive that the ray has intersected, the microphone is reflected in that primitive, and then a ray is cast through it to check for line-of-sight. If there is, an image-source contribution is added.

This diagram shows the first-order image-source reflection.

This diagram shows the rest of the image-source reflections.

Note that there isn't line-of-sight from the source to the third-order microphone image, so no image-source contribution is added for this path.

Figure 1: A simple overview of the algorithm

FFTW3 was used for efficient fast Fourier transforms (FFTs), required for windowed-sinc filtering by convolution. Writing an efficient FFT implementation would constitute a research project on its own, and using the FFTW3 library meant that excessive time was not wasted reinventing a solution to this problem.

Libsndfile allows for audio file output in a variety of formats. The program only officially supports `.wav` and `.aiff` output, but using Libsndfile it would be trivial to add support for other formats, should that become necessary.

The program makes extensive use of configuration files stored in the JavaScript Object Notation format (JSON), and the Rapidjson library is used to parse these files. As with the FFT implementation, writing an efficient parser is not a trivial task, and using a library allowed more time to be spent solving the problem at hand, rather than a problem that had already been solved.

Finally, the Gtest library was used for simple unit-testing of the most integral parts of the program.

Evaluation

The final program works reasonably well, producing realistic reverbs. In long tunnels and large rooms, slap-back and ping-pong echo are emulated, which would not be possible if the algorithm only raytraced the first few reflections. The HRTF is not incredibly convincing, but is able to reproduce simple interchannel level- and time-difference effects.

There is one known bug with the program - with models larger than a few thousand triangles the graphics card sometimes runs out of memory, and the program quits early. This appears to be an issue with the OS X OpenCL implementation, or a driver problem, as when the program quits it does so without throwing an exception - it just quits.

A recurring problem throughout the project was low-frequency estimation. Though high frequencies were easy to model, below around 200Hz the data tables used for material modeling (from Vorländer (2008)) were lacking in detail, and the inherent inaccuracy of ray tracing at low frequencies meant that results were quite unstable. The data tables used provided only 7 octave-bands of frequency data, and for reasons of memory-efficiency all volume information in the program had to be restricted to a single OpenCL vector type, which may contain 2, 4, 8 or 16 elements. Specifically, an 8-component vector type was used, effectively limiting the implementation to 8 frequency bands. Ideally, the implementation would have at least one band per octave of audible spectrum, which would require 10-band data. An improved implementation might use 16-component types to model frequency bands, which would easily cover the audible spectrum, and provide greater-than-octave resolution. However, this would lead to significantly higher memory usage, and would require custom material measurements to be taken, which was deemed beyond the scope of the project.

An interesting extension would be to add multiple source models, in addition to the multiple receiver models (polar-pattern and HRTF). This could be done relatively simply - it would just require the rays to be initialised with volumes depending on their direction, instead of giving them uniform volumes of '1'. However, this would also require a great number of measurements to be taken in order to find appropriate starting volumes for different sources.

It might also be interesting to add user-defined microphone models. Currently, the HRTF attenuation mode works by looking up attenuation coefficients from a table, depending on the direction of the incoming ray. These values were generated using a Python script which carried out frequency-domain analysis of HRTF data from the IRCAM Listen database ((“Listen hrtf database,” 2003)). At the moment, these values are hard-coded, in order to avoid a costly fetch and parse (the tables are quite large), but if a suitable file-format could be created for the tables, it would be reasonably straightforward to allow the user to supply new table values, and therefore new microphone models.

In future projects it might be interesting to try implementing a completely different wave-based model, instead of the current geometric one. It's possible that, using the GPU, a reasonably performant version of the algorithm could be built, which could even be integrated with the current program to provide a different solution for accurate low-frequency estimation.

Finally, the current ray tracing algorithm is fairly naïve. For every intersection test, every primitive in the scene is checked for an intersection. As the speed of the ray trace is limited by the speed of intersection testing, substantial speed-ups may be possible by using a 'spatial data structure' such as a bounding object hierarchy or octree for primitive storage. These data structures drastically reduce the number of primitive checks for each ray intersection, at the cost of increased pre-processing time when the scene is loaded. For the simpler demonstration models used here, these data structures are overkill, but for tracing larger, more complex scenes, they may lead to increased performance.

Bibliography

Cheng, Z. (2014). Making games sound as good as they look: Real-time geometric acoustics on the gpu. Retrieved from http://on-demand.gputechconf.com/gtc/2014/presentations/S4537-rt-geometric-acoustics-games-gpu.pdf.

Cowan, B., & Kapralos, B. (2008, November). Spatial sound for video games and virtual environments utilizing real-time gpu-based convolution. Proceedings of the 2008 Conference on Future Play: Research, Play, Share.

Dalenbäck, B.-I. (2010, May). Engineering principles and techniques in room acoustics prediction. Proceedings of the 2010 Baltic-Nordic Acoustics Meeting.

Dammertz, H., Hanika, J., & Keller, A. (2008). Shallow bounding volume hierarchies for fast simd ray tracing of incoherent rays. Proceedings of the Nineteenth Eurographics conference on Rendering.

Hradský, P. (2011, January). *Utilising opencl framework for ray-tracing acceleration* (Master's thesis). Czech Technical University in Prague.

Kapralos, B., Jenkin, M., & Milios, E. (2004, October). Sonel mapping: Acoustic modeling utilizing an acoustic version of photon mapping. Proceedings of the 3rd IEEE International Workshop on Haptic, Audio and Visual Environments and Their Applications.

Listen hrtf database. (2003). Retrieved from http://recherche.ircam.fr/equipes/salles/listen/

Rindel, J. (2000). The use of computer modeling in room acoustics. *Journal of Vibroengineering*, *4*(3).

Röber, N., Kaminski, U., & Masuch, M. (2007, September). Ray acoustics using computer graphics technology. Proceedings of the 10th International Converence on Digital Audio Effects.

Savioja, L., Lokki, T., & Huopaniemi, J. (2002). Interactive room acoustic rendering in real time. Proceedings of the 2002 IEEE International Conference on Multimedia and Expo.

Schissler, C., Mehra, R., & Manocha, D. (2014). High-order diffraction and diffuse reflections for interactive sound propagation in large environments. *ACM Transactions on Graphics*, *33*(4).

Schröder, D. (2011, February). *Physically based real-time auralization of interactive virtual environments* (PhD thesis). RWTH Aachen University.

Stephens, R. (2013). *Essential algorithms: A practical approach to computer algorithms*. John Wiley & Sons.

Taylor, M., Chandak, A., Mo, Q., Lauterbach, C., Schissler, C., & Manocha, D. (2012). Guided multiview ray tracing for fast auralization. *IEEE Transactions on Visualization and Computer Graphics*, *18*(11).

Vorländer, M. (2008). *Auralization: Fundamentals of acoustics, modelling, simulation, algorithms and acoustic virtual reality*. Springer-Verlag Berlin Heidelberg.