Rendering Impulse Responses for Implicit Surfaces

Noah Weninger
Department of Computing Science
University of Alberta
Edmonton, Canada

December 11, 2018

Abstract

The travel of sounds from sources (instruments, speakers, moving objects, ...) to targets (ears, microphones, ...) is rarely a direct path. Objects in the surrounding environment can interfere, by blocking the direct path and by adding in extra echoes of the original signal. Under some simplifying assumptions, the measurement of this interference for some source and target is called an *impulse response*. Many methods exist for approximating the impulse response for virtual scenes, but all prior work deals with discretized scenes, which limits the amount of detail that can be present. In this work we present a method for approximating the impulse response of *implicit* geometry, which is defined mathematically and can contain an arbitrary amount of detail.

1 Introduction

In everyday life, we constantly use acoustics to more accurately localize ourselves and understand the spaces and materials that surround us. In video games and animated movies, acoustic properties matching the virtual space can greatly enhance immersion. At the same time, acoustics can interfere with our perception. A poorly designed concert hall or conference venue can render speech and music incomprehensible due to excessive echo. Having a building constructed is an expensive way to find that the design has undesirable acoustic properties, so we would prefer to simulate the acoustic properties of the space beforehand. The simulated properties can be used to adapt and evaluate the architecture of a building to fit the desired qualities.

To simulate the acoustic properties of a space, we need a model of it. For architecture, computer-aided engineering (CAE) software is typically used for 3D modeling, which can represent objects using a wide variety of data structures. For multimedia applications, triangular meshes are common, however implicit representation methods are becoming more common as compute power available in consumer graphics processing units (GPUs) increases. The modeling approach that is used limits the selection of algorithms available for performing acoustical simulations. Each algorithm comes with a unique set of advantages and disadvantages, so the careful selection of a modeling method is essential for achieving the desired result.

This paper will focus on implicit modeling methods, which have not previously seen application to acoustic modeling. The detail which sets implicit approaches apart from others is that space is represented as a continuous function. For comparison, a triangular mesh must be fully stored in memory, which limits the amount of detail that is representable. Implicit methods represent surfaces as computational structures, which limits representational power only by the amount of time the user is willing to wait. Implicit methods can be used to represent complex yet highly structured objects with significantly less work than traditional, data driven approaches.

In this work, we found implicit modeling to be a highly effective approach. In particular, the methods described run at a reasonable speed on consumer GPU hardware while producing results of good quality. Although some kinds of objects are difficult to model as implicit surfaces, others become much easier to represent, suggesting that a hybrid approach may be best. The relative simplicity of our approach may also lend to use as an educational tool. The project has been released as open-source under an Apache 2.0 license. Source code is available on GitHub[1].

2 Prior Work

Designing Sound. In the book Designing Sound [Farnell, 2010], some properties of acoustics are described which have implications for this work. The complexities of some of these properties are not accounted for with our method.

[1] https://github.com/nwoeanhinnogaehr/implicit.ir

Other, more advanced methods described in this section account for these additional acoustic properties.

The adiabatic index [Farnell, 2010, p. 55] adjusts for the property of gases where temperature increases in high pressure areas. In turn, this impacts the speed of sound, because the speed of sound increases proportional to the square root of temperature. This is particularly import to account for when simulating outdoor scenes, in which air temperature changes proportional to altitude can affect the propagation of sound.

Thermoviscous attenuation occurs because of the viscosity of the medium [Farnell, 2010, p. 61]. The amount of loss is proportional to frequency (Stokes law), acting as a kind of high pass filter. This has little affect on sound traveling through air because air has low viscosity.

194dB is the maximum possible rarefaction: the low pressure part of a sound wave. If this were not the case, the pressure could become less than a vacuum [Farnell, 2010, p. 58]. However, unipolar compression pulses at higher than 194dB are possible and can occur in situations such as explosions. These extreme situations can cause inaccuracies in simulations which do not explicitly model pressure.

Real-world surfaces can be modeled to have arbitrary frequency distribution of diffuse and specular properties [Farnell, 2010, p. 68].

Oblique boundary loss is a property in which high frequencies traveling parallel to a surface can experience loss due to a boundary layer which resists longitudinal movement [Farnell, 2010, p. 66].

Real Sound Synthesis for Interactive Applications. Before diving into more advanced acoustic simulation methods, we will explore some alternative models and simpler approximations from [Cook, 2002].

To simulate the vibration of a flexible 2D membrane, we can decompose the vibration into modes using a Fourier series as a modeling tool. However, it quickly becomes complicated to determine the wave paths on the membrane, which are needed to determine if a mode appears for a given strike position [Cook, 2002, p. 132]. No general formula is given, even for perfectly square membranes. For more complex objects, the easiest approach is often to pour some sand on the object, vibrate it and observe the modes. In some cases, such as with the Tibetan singing bowls, modes can be so close in frequency that the interfere with each other and low frequency beating can be heard [Cook, 2002, p. 144]. Although this is not the same type of acoustic simulation considered in this work, future work may be able to integrate these methods.

Similarly to how a string can be modeled as a line of alternating masses and springs, a membrane can be modeled as a mesh of masses and springs. This can be computed using a system called a *waveguide mesh* [Cook, 2002, p. 134]. Square grids are the simplest computationally, but they have the disadvantage that the speed of sound is a factor of $\sqrt{2}$ slower along the diagonals. A triangular mesh can be used as a better approximation, but it is very difficult to model perfectly. These methods can easily be extended to 3D, though at great computational cost due to the grid resolution required for high quality results. Generally, the waveguide mesh is known as a first-order finite-difference time-domain (FDTD) method.

Instead of considering a discrete approximation of all wave paths, for some applications it may be sufficient to consider only a finite number of specific wave paths [Cook, 2002, p. 140]. This idea forms the basis for the class of geometric methods for acoustic simulation, the category which our method falls under.

The FAUST physical modeling library: a modular playground for the digital luthier. FAUST is an audio programming language and environment which is often used for physical modeling since it includes a diverse library of models. However, most models are standalone objects which can't be recombined. In [Michon et al., 2018], the authors use an approach inspired by Modalys and CORDIS-ANIMA in which parts of instruments such as tubes, strings or bodies can be assembled in a modular fashion. Unfortunately, FAUST doesn't provide bidirectional connections between modules which makes this kind of abstraction difficult to implement. To remedy this they extend FAUST to support bidirectional communication by creating extensions written in C. The new bidirectional blocks have three inputs and outputs corresponding with left-going waves, right-going waves, and pass-through. Their use inserts a one sample delay at each block in the chain. The toolkit comes with built in models of violin, brass, clarinet, and flute which can be easily reconfigured and recombined. Additionally, the tool `mesh2faust` is provided for making new virtual instrument parts given a 3D mesh. Modeling of instruments is an interesting application of acoustic simulation which has definite musical potential.

An efficient GPU-based time domain solver for the acoustic wave equation. In [Mehra et al., 2012], an approach is developed for accurate acoustic simulation using a hybrid approach in which spaces are subdivided into rectangular subregions. Within a rectangular region the acoustic wave equation can be perfectly solved using the discrete cosine transform. Between each rectangular region or boundary a perfectly matched layer (PML) is added to transfer waves across the virtual barrier. This approach leads to far less numerical error than other

similar approaches, because error within each rectangular region is virtually eliminated. A detailed comparison is given between this technique and the other common approaches: finite element method (FEM), boundary element method (BEM), finite-difference time-domain (FDTD), and spectral methods. Geometric methods similar to our approach are not considered. FEM allows for cells of varying sizes and shapes, but the shapes can affect error so generating good meshes is challenging. BEM expresses values in the field in terms of values only on the boundary so the complexity scales with surface area rather than volume, but the system is very dense because every reacts with everything else. BEM is commonly used for exterior scattering problems, where many of the waves are lost. FDTD can use arbitrary order of accuracy, although a high-order FDTD solver often takes multiple weeks to perform a needed simulation. Spectral techniques often perform better than FEM/BEM/FDTD by using a Fourier or Chebyshev basis. They have less spatial error but similar temporal error. With their implementation, a 1 second simulation of an average scene can be performed within 18 minutes on a GPU (GTX 480), or 5 hours on a CPU. These methods are of relevance to our work but fall within a different class, as they focus on perfect simulation where as our work is more suitable for real-time use.

Toward Wave-based Sound Synthesis for Computer Animation. In [Wang et al., 2018], a technique for high quality offline physical sound synthesis is described. Their approach is to use a sharp-interface FDTD wave solver along with supporting algorithms to process rapidly deforming surfaces. With this technique, they are able to produce extremely realistic simulations of objects such as crash cymbals and water. This computation is performed on a GPU and is scalable to multiple GPUs as it is fully time-parallel. Ultimately, the flexibility of their approach is due to a new abstraction technique called acoustic shaders which describe the objects in the scene and how they change over time. Data generated from these shaders is fed into the solver to produce sound. With this abstraction, users wishing to model new objects need not know anything about acoustics, and the acoustic engine can be upgraded without requiring any changes to the models. The exact acoustic model used is too complex to describe in detail here. Even though their algorithm is massively parallel it still takes a few minutes per second of audio, even on multiple GPUs. However, the sound quality is phenomenal. This work is similarly in a different class from ours, but is of interest due to the idea of acoustic shaders, which inspired some of the design of our implementation.

Application of computational GPU OpenCL kernels for near-realtime audio processing. As GPUs are being increasingly used for audio work, new challenges are arising that are foreign to the research disciplines which typically make heavy use of GPUs. One such issue is the low latency at which many audio applications must be handled. In [Kartsev, 2018], the most performant GPUs are found to have the worst latency, likely because their pipelines and optimization processes are longer and more complex. The author details a number of tricks for minimizing latency in OpenCL, through which he is able to reduce audio latency down to 32 samples. These ideas are worth considering for any high performance audio work performed on a GPU.

3 Background

3.1 Impulse Responses

An acoustic model for certain systems can be represented with an audio-rate signal called an impulse response. To use this representation, we make the assumption that the system is *linear time-invariant* (LTI). Formally, a system is LTI if the map f from inputs to outputs obeys the properties of linearity and time-invariance. A linear system has two properties [Cook, 2002, p. 22]:

- Homogeneity: if $f(x) = y$ then $f(\alpha x) = \alpha y$
- Superposition: if $f(x_1) = y_1$ and $f(x_2) = y_2$ then $f(x_1 + x_2) = y_1 + y_2$

A time-invariant system has the property:

- if $f(x(n)) = y(n)$ then $\forall N$ we have $f(x(n + N)) = y(n + N)$

This limits the kinds of materials we can represent, but in practice is sufficient for most applications. For example, an impulse response is unable to directly model the travel of sound through water, which is not time-invariant due to the effect of the movement of water upon sound propagation. In theory, sound propagation in air is similarly not LTI, though the nonlinearity is negligible under all but the most extreme of circumstances, such as large explosions.

Intuitively, an impulse response records the output of the system given a brief impulse as input. As long as the impulse contains all frequencies of interest, it can be used to find the system's output for an arbitrary signal. In theoretical research, the impulse used is the Dirac delta function, but in practice an ideal impulse is impossible to create so a pulse lasting for exactly one sample is used instead, which contains all frequencies up to the Nyquist frequency. Although a linear system may have an infinite amount of output from a single impulse, all real world systems of interest have an impulse response which is nonzero over a finite interval and therefore can be considered to have finite length. These practical considerations are what makes impulse responses usable for acoustic simulation.

The process by which an impulse response is used to simulate a system for some input is called *convolution*. The convolution of a discrete impulse response a with input b (or vice versa; convolution is commutative) at time $t \in \mathbb{Z}$ is defined as:

$$(a * b)(t) = \sum_{i=-\infty}^{\infty} a(i)b(t-i)$$

Since we assume that a has a finite length, the formula can be reduced to a finite sum. However, for an impulse response of length n and an input signal of length m, the time complexity of this operation is $O(nm)$, which may limit real-time applications. With some improved algorithms, the time complexity can be reduced further, depending on the specifics of the sequences to be convolved. In a well-tuned implementation, convolution can be used to perform acoustic simulation at extremely fast rates given a precomputed impulse response.

3.2 Implicit Geometry

In d-dimensional Euclidean space, a function $f : \mathbb{R}^d \to \mathbb{R}$ implicitly describes a surface by the set of points $L_0(f) = \{x \in \mathbb{R}^d \mid f(x) = 0\}$. A simple implicit function to consider is the d-sphere of radius r: $f(x) = \|x\|_2 - r$, which is more commonly given as $f(x) = \|x\|_2^2 - r^2$. These two functions are not the same, but they describe the same surface because $f(x) = 0$ has the same solutions in either case. However, the former definition has an additional property: $|f(x)| = \min_{\omega \in L_0(f)} \|x - \omega\|_2$. Geometrically, $|f(x)|$ is the Euclidean distance to the nearest point on the implicit surface described by $L_0(f)$. If this is the case we call f a *distance function*. Further, we can interpret the sign of $f(x)$: for $f(x) < 0$ we are *inside* the sphere, and for $f(x) > 0$ we are *outside* the sphere. If this additional property also holds then we call f a *signed distance function*. These properties turn out to be essential for efficiently computing necessary operations on implicit surfaces.

Given a signed distance function describing our scene, there are two primary operations we will want to perform on it. The first is to find the first point of intersection between a ray and the surface, if one exists. Formally, for some ray beginning at p and traveling along the unit vector d, we would like to find the smallest $t > 0$ such that $p + dt \in L_0(f)$. A method to find this is given in Algorithm 1. This intersection operation is perhaps most easily motivated by computer graphics, where it can be used to find which object intersects a particular ray from the camera position. The same operation can also be used to trace the propagation of sound waves. The second operation we would like to perform is calculation of the *surface normal* at some point $p \in L_0(f)$. The surface normal at p is a unit vector which is perpindicular to the surface surrounding p. By definition it is only defined at points for which the surface is smooth. Over a signed distance field, the normal is equivalent to the gradient $\nabla f(x)$. In practice this gradient is often computed numerically, for ease of implementation and so that some approximation can be given at non-differentiable points. The definition of signed distance fields lends to these particularly concise operations, which will play a key role in our method.

It might appear obscure how one would construct a signed distance function to represent an arbitrary scene. However, in practice most common shapes can be described simply. Well known formulas are easy to compose into a wide variety of scenes. Beyond ordinary scenes, fractal composition and noise deformation are almost trivial to express and can create some very bizzare and beautiful shapes. Some surfaces can be challenging to express as an exact distance field, but are simple to find an under-approximation for. To handle these cases, it is safe to relax our definition to require that the distance function is at most the true distance: $|f(x)| \leq \min_{\omega \in L_0(f)} \|x - \omega\|_2$; however, it is important to note that with a very poor approximation sphere tracing may take a long time to converge.

4 Methodology

The problem tackled by this work is to render an impulse response given an implicitly described scene, a sound source, and a sound target. Here the sound source is an acoustic monopole: an ideal point which radiates sound equally in all directions. The target is a specific implicit surface within the scene. Then, at a high level, our method

Algorithm 1 The *sphere tracing* algorithm [Hart, 1996]. This method exploits the property that surrounding some point x, there is a sphere of radius $|f(x)|$ which does not strictly contain any points in $L_0(f)$. Constants MinDist and MaxDist are used to guarantee termination.

```
function SphereTrace(f, p, d)
    t ← 0
    while t < MaxDist do
        x ← f(p + dt)
        if x < MinDist then
            break
        end if
        t ← t + x
    end while
    return t
end function
```

sends many impulses out from the source at different angles and measures how long each takes to reach the target, if at all. Every time an impulse bounces from a surface, it inverts in phase, so we would also like to know how many bounces occurred before the impulse reached the target. Given this information, an impulse response can be constructed as a sum of the impulses traced over all directions after they have been time-shifted, phase-inverted, and scaled due to attenuation accordingly.

Our method uses the sphere tracing algorithm to render impulse responses of implicit surfaces, in a similar manner to how ray tracing methods are used for triangular meshes. To fully trace wave paths from source to target, sphere tracing must be performed once for each bounce. The angle at which the ray bounces depends on properties of the surface. For a perfectly specular surface, the ray is reflected perfectly across the normal of the surface at the intersection point. For a perfectly diffuse surface, the ray is assigned a direction from a cosine distribution around the surface normal. Other types of reflectivity can be achieved by interpolating between the specular and diffuse rays or by deforming the surface itself through the addition of noise to the signed distance field.

Possible source rays are iterated over using spherical coordinates. The two angles $\theta \in [0, \pi)$ and $\phi \in [0, 2\pi)$ have their range divided into a whole number of values which are iterated though to produce rays spanning the unit sphere. The number of divisions chosen has an strong impact on the quality of the final rendering, so it should be sufficiently large. To reduce aliasing error, random offsets of up to one pixel in magnitude are added to coordinates. With diffuse surfaces, it is particularly important that the results are summed over multiple runs to ensure there are enough samples obtained. For specular surfaces this has less impact, but it can still increase quality slightly. For some scenes it may be desirable to have the source only span a subset of all angles. Thus the range to iterate and the number of runs can be specified individually for each scene.

The sound target, which may also be known as the microphone, eardrum, destination, receiver, or sink, is specified as a surface with some additional properties assigned to it. This surface is part of the scene and can be regarded as a regular surface for the purposes of sphere tracing, but each point on it is also labeled with an integer identifier and an angular filter. After each iteration of sphere tracing completes, the object which was intersected is evaluated to determine if it was a target surface. If so, iteration is aborted and the extra properties are returned for use in constructing the impulse response. The integer identifier is used to identify which channel the wave path belongs to, which for example can be used to create stereo impulses. The angular filter is simply a 3D vector f which is used along with the direction vector d of the ray of intersection to scale the impulse for a path by $\max(\langle d, f \rangle, 0)$. It can be used to make a target which only responds strongly to rays traveling at a specific angle, which would otherwise be very difficult to do by the shape of the surface alone.

The speed of sound, given in meters per second, can be adjusted individually for each scene. Attenuation is handled simply by following the inverse square law, which states that sound intensity is inversely proportional to the square of the distance. Other attenuation has extremely little effect under most circumstances and is ignored here.

To aid with scene design, we also output two images depicting the scene visually. For the first output, the scene is rendered using the standard sphere tracing algorithm, using a perspective camera and basic diffuse lighting. These images help identify problems with the scene description, which can be error prone due to the tedious nature of specifying distance fields and parameters. The second output is a visual representation of the wave paths that are found by the path tracer. To produce this output, the sphere tracing algorithm records how close it is to the nearest target surface at each iteration. An image is then produced depicting this minimum distance for all source

rays. This image can be used to judge whether the expected wave paths are discovered.

Although our method is very simple, it should be sufficient for use in music or virtual environments. This simplicity makes the algorithms easy to extend. Much of the described approach was chosen somewhat arbitrarily to fit our immediate goals, and can be easily modified to fit the needs of the user.

5 Usage

Usage of our implementation requires a Linux PC, or something compatible. Porting to other platforms is expected to be relatively easy, and may be explored in the future. The only two non-standard prerequisites are the Rust compiler and a GPU supporting OpenCL 1.2 with the `cl_khr_fp64` and `cl_khr_int64_base_atomics` extensions. Using OpenCL on a CPU or other architecture may be possible, but a GPU is strongly recommended because of the highly parallel nature of the computation.

Once prerequisites have been installed, a pre-existing scene can be rendered using the command `$ ./run.sh SCENE_NAME`, where `SCENE_NAME` is one of the file names in the directory `src/scenes/` stripped of the file extension. At the time of writing, the available example scenes are `ball_room`, `donut_room`, `sphere_room`, `box`, `grating`, and `cylinder`. An optional argument `--debug` can be specified after the scene name to indicate that only the debug image should be rendered and not the impulse. For a full list of parameters, run with `--help`. The run script writes all output to the `output` directory, clobbering any existing files. Make sure to save anything of importance before running the script again. Impulses are written to files named like `impulse000.wav` and debug files have a PNG extension.

Creating a new scene can be performed by implementing a handful of OpenCL functions and defining some constant values. A description of each of the items is provided below:

- `double MAX_DIST` is the maximum distance in meters to travel while sphere tracing before bailing out and reporting no intersection. It is important to set this to a reasonable value when the scene is not an enclosed space.
- `int MAX_BOUNCES` is the maximum number of ray bounces to perform before bailing out. For specular materials this should be set higher for than diffuse materials.
- `double SPEED_OF_SOUND` is the speed of sound to use when generating the impulse response.
- `double SAMPLE_RATE` is the sample rate to use for the impulse response.
- `bool DIFFUSE` specifies whether the surfaces in the scene are diffuse. In the future this may be made variable across surfaces within the same scene.
- `int NUM_TARGETS` specifies how many identifiers there are for the target surfaces. Typically this is set to 2, indicating a stereo impulse. Identifiers are numbered from 0 through `NUM_TARGETS`-1.
- `int NUM_SUBSCENES` specifies the number of subscenes to render, which are integer parameterized variations on the same scene.
- `int NUM_RUNS` specifies the number of runs to average results across.
- `int IMPULSE_LEN` specifies the number of samples in the impulse response to generate.
- `double3 SOURCE` is the audio source location in 3D space.
- `double SOURCE_THETA_MIN` is the smallest θ (spherical coordinates) to sweep when generating source rays.
- `double SOURCE_THETA_MAX` is the largest θ to sweep.
- `double SOURCE_PHI_MIN` is the smallest ϕ to sweep.
- `double SOURCE_PHI_MAX` is the largest ϕ to sweep.
- `double3 DEBUG_ORIGIN` is the camera coordinate for the visual render of the scene, produced as a debugging aid.
- `double3 DEBUG_DIR` is the camera direction.

- `double3 DEBUG_LIGHT_DIR` is the light direction.
- `double3 DEBUG_MAX_DIST` is the sphere tracing bailout distance for the debug image.
- `Target sdf_target(double3 p, int subscene_id)` is a function which returns the nearest target surface to `p`, given a subscene ID between 0 and `NUM_SUBSCENES`-1. `Target` objects are constructed using the constructor function `target(double distance, int target_id, double3 axis_filter)`, which takes parameters as described in Section 4. A function `target_min` is provided for taking the union of two target distance fields.
- `double sdf_scene(double3 p, int subscene_id)` is a function which returns the signed distance field for the scene at point `p`, for the given `subscene_id`. This function should call `target_sdf` to include the target distance in the distance field so that targets are not missed during sphere tracing.

Many of these fields can generally be left at the default values, but it is important to ensure that they are consistent with each other. For example, if the source location is placed inside of an object (where the signed distance field is negative), then no rays will ever reach the target. In case this was done by accident, the debug images produced can help to diagnose the issue.

Due to a bug in the OpenCL shader cache managed by the proprietary NVIDIA driver, clearing the shader cache is required in between runs on NVIDIA hardware. The run script handles this automatically by deleting the directory `~/.nv/ComputeCache/`. Users should evaluate whether this is a safe action to perform on their system before using the script.

To simulate movement within the scene, multiple subscenes can be set up with the desired movement. An included file, `convolve.py`, performs real-time convolution with an impulse response that is changed over time. The provided convolver uses the short-time fast Fourier transform to linearly interpolate between impulse responses with configurable overlap. The script requires the open-source dafxpipe software[2], which was developed by the author. Currently, only a JACK audio backend is supported.

Additional documentation may be available on the project's GitHub page.

6 Implementation

The bulk of the implementation is written in OpenCL. A Rust program using the `ocl` library is responsible for queuing the OpenCL kernels and writing output to disk.

The largest difficulty in implementation arose from the intricacies of floating point precision. We define one unit of distance as one meter. For the sphere tracing algorithm, a small ε is used to determine if an object has been hit by a ray. Similarly, the numerical differentiation code requires some ε' as a step distance. Because we have no guarantees about the thickness of a surface, the gradient may be undefined at the surface. Therefore, upon intersection we need to step back from the intersection point by some ε'' before we can evaluate the gradient. For this to work, we must have that $\varepsilon'' > \varepsilon' + \varepsilon$. If these values are made too small, then floating point precision degrades the quality of the results. If they are too large, then the gradient can become inaccurate, and the backwards step by ε'' may even step right through a surface. Finding reasonable values for these constants was deemed impossible using 32-bit floats, so doubles are used instead.

Generation of random numbers on the GPU presents an interesting challenge. With all execution totally deterministic and with no way of storing global state on the GPU across kernel executions, the code instead makes use of a well known public-domain GLSL hash function which has been translated to OpenCL. It required quite a lot of tweaking of scale and shift values to achieve a good distribution. There is definite room for improvement in this area.

7 Results

Our implementation was tested on a variety of scenes and listening tests were performed by the author. The most complex scene tested included an order-8 Mandelbulb fractal, which required 120 seconds per run on a NVIDIA Titan X GPU. All example scenes included with the implementation take under 10 seconds per subscene on either the Titan X or GTX 1070. A selection of artifacts (impulse responses and debug images) are available for download on the Internet Archive[3].

[2] https://github.com/nwoeanhinnogaehr/dafxpipe
[3] https://archive.org/details/ImpulseResponsesOfImplicitSurfaces

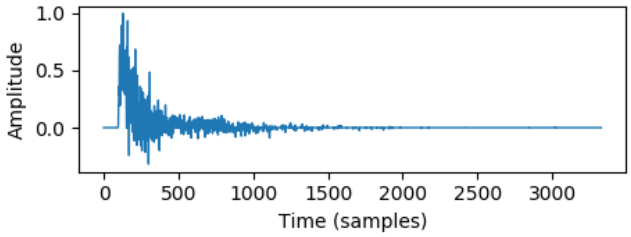

Figure 1: An impulse response produced by the software.

To ensure that enough rays reach the target for the generated impulse responses to converge in temporal coverage, 8 to 16 runs of 2^{20} rays (kernel dimensions of $2^{10} \times 2^{10}$) are usually required for scenes with diffuse surfaces. For more complex scenes, this number may need to be increased further. Specular surfaces usually only need a single run to reach agreeable quality.

An issue that sometimes arises is that surface normals are undefined in sharp corners. This can cause unpredictable behavior because the gradient can be very small or even zero. Adding a random offset to starting coordinates or sizes can usually resolve this problem. This problem can likely be solved completely with more careful calculation of surface normals. With diffuse scenes this issue is mostly avoided due to the inherent noise introduced into the gradient.

The impulse responses for specular scenes are highly sensitive to initial parameters due to the large number of reflections. To see why this might be the case, consider the tremendous complexity of the debug image shown in Figure 2, which is produced from a very simple scene description. Diffuse scenes do not suffer from this problem due to their inherent randomness.

8 Discussion

Listening tests reveal the generated impulse responses to be of good quality. After they have been applied to some sound source, a listener with knowledge of the scene layout can determine the target position just from the audio, using cues such as the sound becoming muffled when the target moves behind an occluding object. Unfortunately, we could not locate a baseline set of reproducible impulse responses to compare our results to.

Many other papers describing methods for acoustic simulation report significantly worse running times. However, it is difficult to compare with vastly different algorithms such as FDTD or BEM, which have different uses and scope. Our implementation is perhaps best compared to other geometric approaches, but we were unable to locate any results for scenes similar to those tested in our work.

Although the objective accuracy of our method remains to be evaluated, the results have definite applications to multimedia in their present state.

9 Limits

As with other geometric methods for acoustic simulation, our approach uses a particle model of sound, which cannot accurately simulate wavelengths that are larger than the objects in the scene. Roughly, at the speed of sound in air (about 343 meters/second), a simulation with objects one meter in size would only be reasonably accurate for frequencies of 343 Hz or greater. In practice we found the results to be agreeable nonetheless, but our approach does theoretically face this issue.

An additional problem is that some wave paths which travel through narrow openings can be missed. Although the random offsets applied to source paths should mitigate this somewhat, it still has the potential to occur. The effect of this would likely also be mitigated slightly by the problem discussed in the previous paragraph, which suggests that only very high frequency sounds should be able to pass through a very narrow passage.

As mentioned previously, impulse responses are limited to modeling linear time-invariant systems, of which the propagation of sound through air only approximates under normal conditions. Therefore, our approach fails to capture certain properties of sound which occur in extreme circumstances.

With signed distance fields, it can be very difficult to model scenes that are mostly data-driven, that is, those which contain a lot of nonprocedural detail. This kind of detail is very common in real-world scenes, where objects

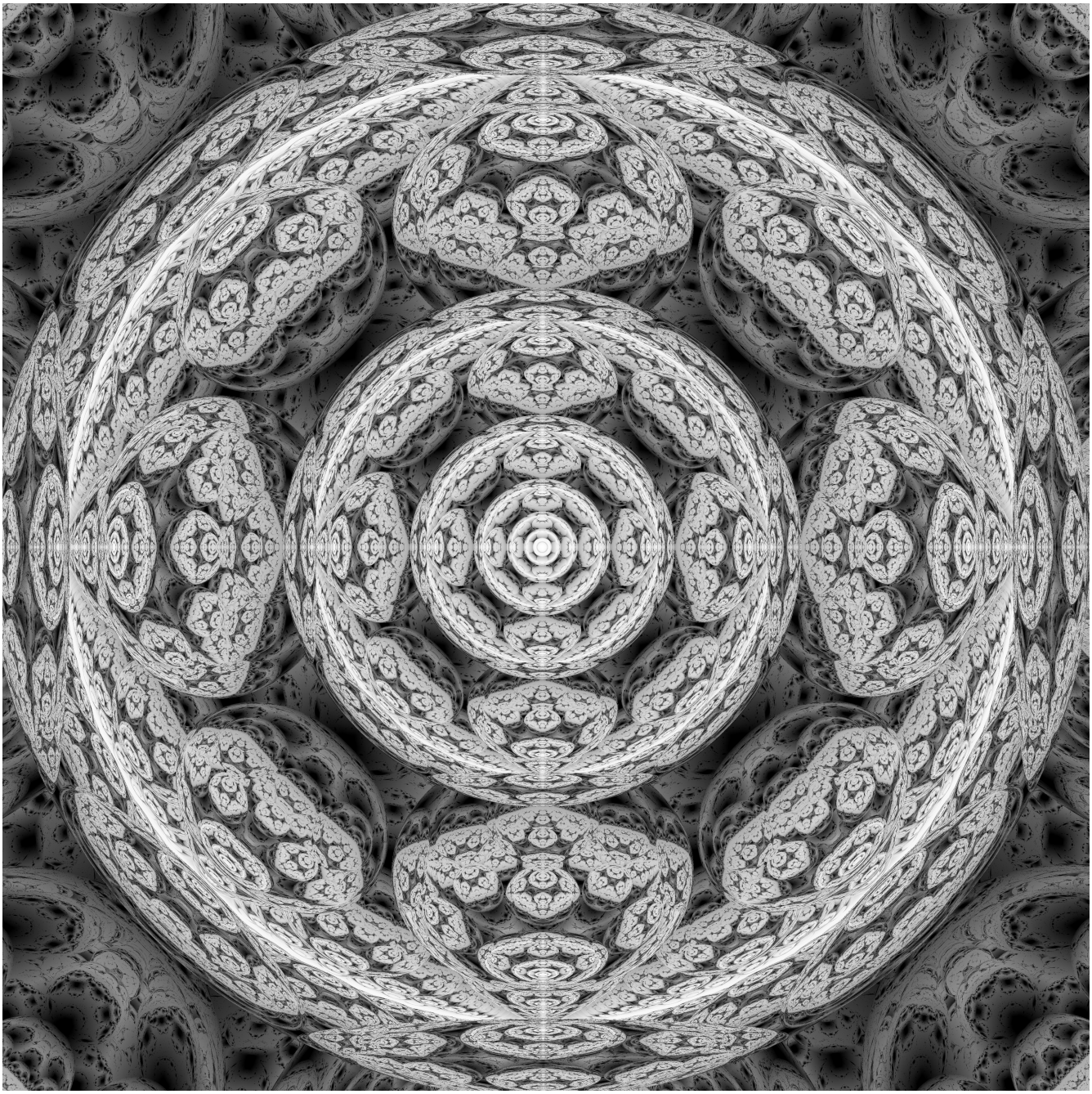

Figure 2: A debug image produced by a variant of the `sphere_room` scene.

can have very peculiar shapes that are difficult to model mathematically. For this reason our work is likely better suited for use in virtual environments and theoretical research.

10 Future Work

In our implementation, an individual ray corresponds to at most one output impulse, which is recorded if the ray intersects a target. However, the iterative nature of the sphere tracing algorithms provides intermediate points along the traced path, each of which could be used as an implicit target location. With this data one could could extract a dense collection of impulse responses across a variety of target locations in the scene. One concern with this method is that points near to surfaces will receive more samples than those far away due to the convergence properties of sphere tracing. Depending on the application, this may or may not be an issue: in a video game for example, players are usually located on the ground, so accuracy of impulse responses high up in the air may not be of concern. Regardless, a robust interpolation method will be needed to assign impulse responses to each point in space. Although there are still some challenges to sort out, this improvement could significantly reduce compute time for applications that require a large number of target locations within a scene.

Our repository of example scenes remains rather simple and small. Inclusion of more component distance functions and composition operations could expand the space of scenes which are easy to construct. More scenes including fractal shapes are particularly of interest, since they are much more difficult to render using alternative acoustic simulation approaches. To increase usability, these improvements would be ideally paired with an interactive scene visualization tool, to improve workflow.

Currently, we do not pass any sound through solid objects. Objects could be given material properties which would allow configurable amount of sound to pass directly through, with the rest reflected either along a diffuse or specular path. The sphere tracing algorithm would need to be adjusted to handle stepping through the inside of objects, a case which is currently ignored with an assertion that the distance field must have a positive value.

The Metropolis light transport algorithm may be usable as an enhancement to the path tracer to improve speed and accuracy. In essence, the algorithm works by using information from prior samples to determine what paths should be explored next. After one path is found from source to target, the algorithm will explore many nearby paths in an effort to gather more samples of paths which intersect the target. Areas of the scene which do not result in any intersections with the target will not be explored further. This algorithm has similarity to Monte Carlo reinforcement learning algorithms. Other reinforcement learning algorithms for value estimation such as temporal difference learning may also be good estimators of impulse responses via path length.

11 Conclusion

Practically, most of these results could be achieved by fine voxelization or triangulation of the implicit surface. With the knowledge that areas with fine detail are more likely to behave as diffuse surfaces, we can further increase the power of this approximation. In addition, hardware raytracing support built into recent GPU models may be able to significantly accelerate traditional impulse response rendering approaches. However, the relative simplicity of our approach is perhaps what makes it worthwhile to explore. We hope that it can later be expanded into a feature-rich tool with applications to education and multimedia.

12 Acknowledgements

The author would like to acknowledge the excellent feedback, discussions and comments from Dr. Abram Hindle over the course of this project.

Some of the code for working with implicit surfaces is inspired by the articles available on the personal website of Íñigo Quílez[4], the content of which is available under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 license. Unfortunately this license is incompatible with other open-source licenses and is generally considered unsuitable for software, so the available code could not be used in our implementation.

[4] http://iquilezles.org/www/index.htm

References

[Cook, 2002] Cook, P. R. (2002). *Real sound synthesis for interactive applications*. AK Peters/CRC Press.

[Farnell, 2010] Farnell, A. (2010). *Designing sound*. MIT Press.

[Hart, 1996] Hart, J. C. (1996). Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12(10):527–545.

[Kartsev, 2018] Kartsev, P. F. (2018). Application of computational gpu opencl kernels for near-realtime audio processing. In *Proceedings of the International Workshop on OpenCL*, page 22. ACM.

[Mehra et al., 2012] Mehra, R., Raghuvanshi, N., Savioja, L., Lin, M. C., and Manocha, D. (2012). An efficient gpu-based time domain solver for the acoustic wave equation. *Applied Acoustics*, 73(2):83–94.

[Michon et al., 2018] Michon, R., Smith, J. O., Chafe, C., Wang, G., and Wright, M. (2018). The faust physical modeling library: a modular playground for the digital luthier. In *Proceedings of the 1st International Faust Conference (IFC-18), Mainz (Germany)*.

[Wang et al., 2018] Wang, J.-H., Qu, A., Langlois, T. R., and James, D. L. (2018). Toward wave-based sound synthesis for computer animation. *Unknown. TODO*.