Simulating Acoustics with Ray-Tracing in Rust

Kristofer Rye

January 27, 2020

Abstract

Ray tracing, a common method used in computer graphics to produce photo-realistic images, has natural applications to the study of acoustics. Sound moving through space can be modeled through particles which represent the wavefront as it moves through space, and sufficient numbers of particles can produce a well-resolved profile of received sound that is emitted from that point. We implement a primitive simulation system to approximate the acoustics of a modeled space and examine its resulting characteristics.

1 Introduction

Ray tracing is a common method used in the field of computer graphics to simulate the motion of light through space, and is commonly used to produce photo-realistic images. The idea of ray tracing in computer graphics originally came from the problem of computing the projection of shadows, [1] but developed in both accuracy and speed after it was popularized through its use in computer graphics. Originally conceived in the field of optics [4], ray tracing eventually reached popularity in cases where it was practical. Ray tracing has also seen use in physics research settings, especially for modeling the propagation of seismic waves, radio signals, and underwater sound.

A few things about sound make ray tracing particularly useful in the modeling of acoustics. First, since sound in air moves at a velocity far less than that of light, this means its motion through the space can be accurately approximated with particles moving at non-relativistic speeds.

Rust is a programming language of rising prominence with a stated goal of "empowering everyone to build reliable and efficient software." [3] Rust was selected for this project because of a few key distinguishing factors. These include Rust's exceptional memory safety guarantees, the inherent fearlessness of concurrency of Rust code, and the fact that Rust can be compiled both to native machine code (on x86_64 and similar devices) as well as WebAssembly, which allows it to be embedded within the browser rather cleanly.

In addition, Rust's type system, specifically its rich support for generic functions and generic types, means that all of the algorithms with which we are concerned can be implemented for all types that support the underlying operation

generics. As a lucid example, this means that the same implementation of our triangle intersection algorithm can work in both two and three dimensions, as it makes use of the dot product which is well-defined in both dimensions.

2 Algorithms

The system we developed allows for the measurement of the acoustics of a modeled space with a specifiable level of detail. Wavefronts are approximated as particles carrying frequency and amplitude data, and are emitted omni-directionally (uniformly distributed, and randomly) within the space, and objects are modeled as either spheres (with a given origin and radius) or collections of triangles.

Our simulation consists of a loop (Algorithm 1) that continues endlessly until all sounds emitted at the start of the simulation have either been received or would be imperceptibly weak by the receiver. For each iteration through the simulation loop, every sound ray is checked against every object in the scene for an intersection ("hit"), and the intersection results are stored in a container which automatically sorts the stored sounds in ascending order by time, such as a binary tree. As a result, the time complexity of checking every sound with every object has time complexity $\mathcal{O}(m \cdot n)$ where m denotes the number of sounds and n denotes the number of objects. The claim that this worst-case time complexity is the best possible without advanced techniques needs verification, but hopefully intuitively makes sense.

Algorithm 1 The simulation structure

```
Objects ← [room geometry]
Objects ← [receivers]
for emitter ∈ Emitters do
    Sounds ← emitter.emit
end for
repeat
    for sound ∈ Sounds do
        I ← []
        for object ∈ Objects do
            I ← sound.hit?(object)
        end for
        hit ← I.first
        if hit.object is a Receiver or
        sound.amplitude * hit.reflectance < ϵ
        then
            hit.object.hits ← hit
            Sounds.delete(hit.sound)
        else
            hit.sound.bounce(hit.object)
        end if
    end for
until Sounds = ∅
```

The algorithms for identifying intersections of sound (approximated by rays) and surfaces (approximated by triangles and spheres) are taken from relevant literature sources. Notably, we use the fast and lightweight algorithm presented by Möller and Trumbore in [2], which can be used not only to determine *whether* a given ray intersects with a triangle, but also *exactly where*, and *at exactly what time*, both of which are relevant to our simulation. The ray-sphere intersection was derived using Wolfram Mathematica to generate a parameterized form of the sphere equation with one parameter (time), and this was then translated into code. Since a ray can intersect with a sphere at exactly zero, one, or two points, the intersection returns both points and the earlier time is selected. Unit normal vectors are generated in both intersection algorithms and are used by the scene simulator to compute in what direction the outgoing ray moves.

One limitation of our system is that it assumes perfectly elastic collisions with un-movable surroundings, and—perhaps more significantly—that sound exclusively bounces at a lower amplitude when it interacts with a surrounding. In reality, a wavefront interacts with the actual materials of the wall on a highly detailed basis, and the level of detail required proves to be impractical. Instead, our system requires that entire surfaces (groups of triangles) are approximated with a single coefficient representing how much of the sound they reflect, and as additional detail is needed, additional surfaces must be added. Furthermore, we assume exactly one medium with one speed of sound, which is not wholly realistic. A given space could have varying speeds of sound in different areas as factors like lighting induce thermal differences and air currents. The difference in sound profile is assumed to be negligible, but in practice might be somewhat significant.

An alternative or more primitive approach to implementation would have rays stepping forward an arbitrarily small amount, checking for bounces off each object at each time step, and moving in accordance with a *computed* speed of sound, rather than a constant speed of sound.

3 Results

4 Conclusion

References

[1] Arthur Appel. "Some techniques for shading machine renderings of solids". In: *Proceedings of the April 30–May 2, 1968, spring joint computer conference on - AFIPS '68 (Spring)*. ACM Press, 1968. DOI: 10.1145/1468075.1468082. URL: https://doi.org/10.1145/1468075.1468082.

[2] Tomas Möller and Ben Trumbore. "Fast, Minimum Storage Ray–Triangle Intersection". In: *Journal of Graphics Tools* 2.1 (Jan. 1997), pp. 21–28. DOI: 10.1080/10867651.1997.10487468. URL: https://doi.org/10.1080/10867651.1997.10487468.

[3] *Rust Programming Language*. URL: https://www.rust-lang.org.

[4] G. H. Spencer and M. V. R. K. Murty. "General Ray-Tracing Procedure†". In: *J. Opt. Soc. Am.* 52.6 (June 1962), pp. 672–678. DOI: 10.1364/JOSA.52.000672. URL: http://www.osapublishing.org/abstract.cfm?URI=josa-52-6-672.