BACHELOR-ARBEIT

vorgelegt an der
Technischen Hochschule Würzburg-Schweinfurt
in der Fakultät Informatik und Wirtschaftsinformatik
zum Abschluss eines Studiums im Studiengang Informatik

Simulating Room Acoustics Using Ray Tracing

Angefertigt an der Fakultät für Informatik und Wirtschaftsinformatik der Technischen Hochschule Würzburg-Schweinfurt

Erstprüfer: Prof. Dr.-Ing. Frank Deinzer

Zweitprüfer: Prof. Dr.-Ing. Arndt Balzer

Abgabetermin: 2. 04. 2024

Eingereicht von: Christina Reichel aus Würzburg

Hiermit versichere ich, dass ich die vorgelegte Bachelorarbeit/Masterarbeit selbstständig verfasst und noch nicht anderweitig zu Prüfungszwecken vorgelegt habe. Alle benutzten Quellen und Hilfsmittel sind angegeben, wörtliche und sinngemäße Zitate wurden als solche gekennzeichnet.

Würzburg, den

(Unterschrift)

Hiermit willige ich ein, dass zum Zwecke der Überprüfung auf Plagiate meine vorgelegte Arbeit in digitaler Form an PlagScan übermittelt und diese vorübergehend (max. 5 Jahre) in der von PlagScan geführten Datenbank gespeichert wird, sowie persönliche Daten, die Teil dieser Arbeit sind, dort hinterlegt werden.

Die Einwilligung ist freiwillig. Ohne diese Einwilligung kann unter Entfernung aller persönlichen Angaben und Wahrung der urheberrechtlichen Vorgaben die Plagiatsüberprüfung nicht verhindert werden. Die Einwilligung zur Speicherung und Verwendung der persönlichen Daten kann jederzeit durch Erklärung gegenüber der Fakultät widerrufen werden.

Würzburg, den

(ggf. Unterschrift)

Übersicht

TEXT DEUTSCH

Abstract

TEXT ENGLISCH

Contents

Chapter 1

Introduction

Anyone who ever moved places knows the amazing difference between the sound of talking inside an empty room as opposed to talking in that same room once it's filled with furniture. Acoustics as a field has long been dedicated to exploring these kinds of differences, as well as developing methods to ensure that a room's acoustics fit its purpose.

As with all fields, software tools have been developed to aid in adjusting a room's acoustics. Several methods have been developed that allow an engineer to simulate the sound of a planned room before building and furnishing it, letting them check whether the acoustics hold up to their intended requirements.

The two most common methods of simulation are numeric and geometric approaches. Numeric approaches work by getting the equation representing the wave's propagation in the room, then calculating a solution to it numerically. This leads to the most accurate results, but has a high computation cost attached to it, making it less viable for real-world uses.

Geometric methods instead model a sound wave as a large set of individual rays originating from the same spot, then simulate them bouncing through the room, similarly to how light is modelled in graphics ray tracers. This works fine for sounds at small wave lengths/high frequencies, but introduces errors for lower frequencies as a sound's wave properties are entirely discarded. In turn, geometric methods are much faster than numeric approaches, to a point where a room's acoustics can be simulated in real time [3]. At time of writing, most commercially available acoustics simulation tools use ray tracing or similar geometric methods [14].

Both of these methods can of course also be used to simulate entirely hypothetical rooms without any intent to build them, or rooms that can't exist in the real world to begin with, which can help physicists study properties of these theoretical places that can then be applied to real ones.

An example for a case that would be interesting to study but near impossible to measure in real life would be the acoustics of a rapidly rotating room. In order to differ from the echo the room

creates while static, the room's rotation would need to reach a velocity at least one order of magnitude below the speed of sound. Building such a rotating room in the real world may be possible at a small scale, but becomes exponentially harder the bigger the room gets.
In order to simulate such a room, the simulation would need to be able to account for the walls of the room moving while the sound wave itself is traversing the scene. This is mostly unexplored thus far. Bilibashi et al. [2] have explored simulating rays bouncing between a set of moving points, namely cars, but their approach is not viable for a full room acoustics simulation.
Existing simulations have only concerned themselves with static scenes. For dynamic scenes, existing tools such as EAR and research such as that by Chandak et al. [3] work around having to account for movement by taking a static version of the scene at a given time and bouncing rays through that instead, then repeating this process for every point in time at which acoustics are simulated.
This works well in most cases, but leads to problems when objects in the room move fast enough that the distance they would travel in the time a ray is bouncing through the scene becomes significant. To start alleviating this problem, this thesis aims to create a new way to simulate bouncing rays through a moving room without errors introduced by the older approach.

1.1 Scope of This Thesis

This thesis proposes a method to simulate rays bouncing through arbitrary scenes with moving receivers and/or objects, assuming all movement within the scene is known at time of calculation. An improved way of checking for intersections between rays and objects is developed, accommodating for this new requirement.
Optimisations are evaluated and a time-based chunking method is developed to avoid needless intersection checks.
The goal of this research is to lay part of the groundwork required to allow accurate simulation of acoustics in moving rooms.
Three test cases are developed for this and compared to an implementation more akin to existing methods: An empty scene with the sound receiver approaching the sound emitter at 1/9th the speed of sound, a square room rapidly rotating and a large, L-shaped room also rapidly rotating around one of its ends, with the receiver and emitter both sitting in said end.
Not within the scope of this thesis is development of a fully accurate simulator including effects such as the differing bouncing behaviour sound waves show at different frequencies. Only a proof of concept that shows that the idea of this new simulation method works is developed.
Side effects of moving scenes, such as sounds emitted by moving objects, are also discarded as

they are irrelevant to the changed intersection logic.
In the real world, rapid rotations such as the ones in the example scenes would create a centrifugal force, which would lead to air pressure differing in different parts of the room, in turn changing the behaviour of sound waves bouncing around the room. Other rapid movements would create other forces and pressure gradients. Exploring these changes requires further research and is outside the scope of this thesis.
A note-worthy side effect that gets ignored is mass inertia: The example case where this would become relevant is the inside of a linearly moving enclosed room, such as a driving car. Due to mass inertia, sound waves travelling inside this moving room would behave the same as if the car stood still. Since this effect is only relevant in a niche scenario and it can be simulated using a method that ignores movement entirely, it can be ignored for this research.
Real-time applications cannot use this proposed method as it requires knowledge of objects' future movements ahead of time. Further research is required to develop an alternative method for real-time or dynamic simulations.

1.2 Outline

In chapter 2, a basic outline on existing research and the general functionality of geometric acoustics simulation is given. The following chapters cover new research based on these fundamentals.
chapter 3 then focuses on the expanded logic required to bounce rays through moving scenes. This involves deriving new equations to check for intersections with moving objects. The performance cost of these new checks is compared to that for static objects.
To alleviate part of this cost, in chapter 4, a commonly used optimisation that divides a scene into separate chunks and only checks for intersections with objects in the relevant chunks is adapted to moving scenes.
For chapter 5, a proof-of-concept implementation of the methods developed in this thesis is developed and compared to an implementation based on static scenes. This evaluation is both looking at changes in simulation quality and performance from the static methods to the newly developed ones as well as seeing where the logic proposed in this thesis still falls short and requires further research.

Chapter 2

Fundamentals

Chapter 3

Bouncing Rays Through Moving Scenes

To bounce a ray through a scene, checks need to be performed to know which objects it intersects with and thus bounces off of. These checks are usually done by modelling objects using an equation that describes whether a point is on the object surface or not, then injecting the ray's function into that equation and resolving. With static objects, this check is trivial: The only parameter to resolve for is the intersection time t, and said parameter only occurs in the ray function, usually making for a first-degree polynomial function.
As an example, a surface S can be modelled using a point P_1 on it as well as its normal n. Then, to tell if any point p is on S, the vector from P_1 to p can be compared to n. If they're orthogonal to each other (thus their dot product is 0), p is on S:

$$(p - P_1) \cdot n = 0 \tag{3.1}$$

For a polygon defined by its corner points P_1 to P_n, the same equation can be used, with the difference that n needs to be determined from a set of at least three of the points. The cross product of two vectors is always orthogonal to both vectors, so using points P_1 through P_3, n can be determined as

$$n = (P_2 - P_1) \times (P_3 - P_1) \tag{3.2}$$

Additionally, a polygon needs to further check whether p is actually inside the bounds defined by its corners. For triangles, this can be done using barycentric coordinates.
Barycentric coordinates describe a point on a triangle (or another simplex) by separating a triangle into three parts, each with the given point and two corners of the triangle as its vertices. Each coordinate then describes the relative size of that triangle with respect to the full triangle. Thus, each point inside the triangle can be described by coordinates (α, β, γ), with $0 \leq \alpha \leq 1$,

$0 \leq \beta \leq 1$, $0 \leq \gamma \leq 1$ and $\alpha + \beta + \gamma = 1$. If these conditions do not apply to p, it is outside the triangle.
The naive way to calculate barycentric coordinates would be to calculate the areas of P_1P_2p, P_1P_3p, P_2P_3p and divide them by the area of $P_1P_2P_3$ [12]:

```
normal = cross((p3 - p1), (p2 - p1));
p1p2p = dot(normal, cross((p1 - p), (p2 - p)));
p1p3p = dot(normal, cross((p3 - p), (p1 - p)));
p2p3p = dot(normal, cross((p2 - p), (p3 - p)));
p1p2p3 = p1p2p + p1p3p + p2p3p;
alpha = p2p3p / p1p2p3;
beta = p1p3p / p1p2p3;
gamma = p1p2p / p1p2p3;
```

This can be simplified by only computing two of the barycentric coordinates and calculating the last one from the first two instead; if α and β have already been computed, $\gamma = 1 - \alpha - \beta$. This saves a division operation. For further optimisation, the Lagrange identity $(a \times b) \cdot (c \times d) = (a \cdot c)(b \cdot d) - (a \cdot d)(b \cdot c)$ can be used to replace the cross products with dot products, as shown by Ericson [6].
In order to calculate the intersection point p, the equation describing a ray then needs to be injected into the surface equation. A ray R's position at time t, with R starting at the point P at time t_0 and travelling in the direction v can be defined as follows:

$$R(t) = P + (t - t_0) \cdot v \tag{3.3}$$

This equation can then be injected into the equation describing the primitive and resolved. As an example, using (3.1), this calculation goes as follows:

$$\begin{aligned} (P + (t - t_0) \cdot v - P_1) \cdot n &= 0 \\ t &= \frac{(P_1 + t_0 \cdot v - P) \cdot n}{v \cdot n} \end{aligned} \tag{3.4}$$

If t exists and is greater than t_0, the point $p = R(t)$ describes the intersection point and can be used for further checks as well as bouncing.
Existing tools use this same method for intersection checks when simulating dynamic or moving scenes. A snapshot copy of the scene at the time a ray is launched is made, then intersection checks and bouncing logic use this static scene. This approach to moving scenes will be called the Snapshot approach.

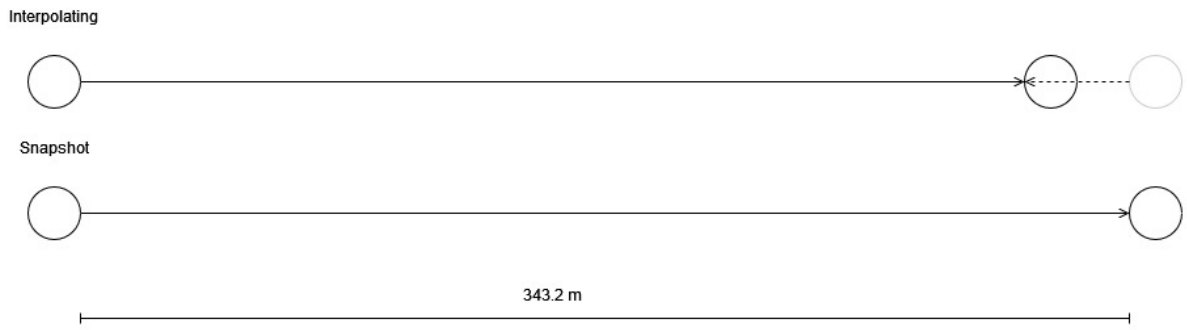

Figure 3.1: Difference between ray travelling distance using the newly developed interpolating method (top) as opposed to the snapshot method (bottom). In the interpolated version, the ray only travels part of the distance as the receiver travels the remainder.

This snapshot approach comes with a few advantages: The simple and well-optimised intersection logic from static scenes can be used without changes. Also, crucially, knowledge of how the scene will move over the time the ray spends bouncing around it is not required. All data necessary to simulate the bouncing is available at the time the ray is emitted, without a need for information on how the scene will continue to move. This is especially helpful for real-time simulation of dynamic scenes, as data about how the scene will continue to move is not fully known at runtime.

The major downside of this snapshot approach is that it tends to introduce errors when objects or receivers move at high speeds. As a simple example case, take the scene described in 3: A receiver starts 343 meters away from an emitter and moves towards it at 1/9th the speed of sound, roughly 38 meters per second (137.2 kilometers per hour, a speed most modern cars can reach without problems).

Using the snapshot approach, a ray traveling directly from emitter to receiver would arrive after travelling the full 343 meters, taking 1 second for it to arrive at the receiver. In actuality, in the time the ray takes to travel the first 90% of that distance, the receiver has already travelled the remaining 10%, making for a response time of 0.9 seconds rather than 1 second.

To avoid this issue, new intersection logic that takes objects' movements into account is needed. This check becomes more complicated than its static counterpart as the equation describing the object still needs to be resolved for t, but now t is also part of the equation itself. In this chapter, equations are established and solved to perform intersection checks with moving spheres and surfaces, as those two should already be sufficient to simulate most scenes. Resolution for other primitives, such as quadric surfaces, should be derivable using a similar method.

The general scheme for this is as follows: First, the object is modelled in a way where its posi-

tion (or other parameters) is also a function of t. Then, like with a normal check, the function modelling the ray is injected into this equation, then resolved. This will, in the examples below, result in a polynomial function of a higher degree (2nd and 3rd for spheres and surfaces, respectively), which can then be resolved.
Unlike with most ray tracing implementations, the magnitude of the ray's direction vector v becomes significant: Since the movement of objects checked for intersections is also dependent on time, objects and the ray must move at the same time scale. v must accordingly be scaled to fit the velocity of the ray and the implementation's scale for t and coordinates. Otherwise, the intersection calculation will yield incorrect results as at the resulting intersection times, the ray will either not have travelled to the resulting intersection point yet or will have already surpassed it.
Note that since the resulting polynomial will be of at least 2nd degree, it can have multiple roots. Only roots greater than t_0 are relevant as others would represent intersections from before the ray is even launched. Different primitives might also require different checks to filter out invalid roots depending on how they are modelled, such as the check for whether a point is inside a triangle described above. Of the remaining roots, the first one represents the actual intersection time, the remaining ones can be discarded as they'd only take place if the ray didn't bounce off the first intersection already.

3.1 Intersection Checks for Spheres

A sphere C is modelled as having a radius r and a collection of keyframes k_n, each with a center point c_{k_n} and a time t_{k_n}. Intersection calculations need to be run separately for each set of consecutive keyframes k_1 and k_2.
The center point c_t of C at time t can be defined as a blend between the two keyframes' center points c_{k_1} and c_{k_2}:

$$c_t = m \cdot c_{k_1} + (1 - m) \cdot c_{k_2} \tag{3.5}$$

m represents the proportion of c_{k_1} as opposed to c_{k_2} at a given time. The definition of m depends on the interpolation mode. For the scope of this thesis, linear interpolation is used, leading to m being defined as

$$m = \frac{t_{k_2} - t}{\Delta t} \tag{3.6}$$

with

$$\Delta t = t_{k_2} - t_{k_1} \tag{3.7}$$

For non-linear interpolation modes between two keyframes (such as using a sinusoidal function), m is defined differently. The math below is still the same until (3.10) where m is resolved, at which point the new definition of m must be substituted instead of the definition above. Resolving from there should be trivial, depending on the definition of m.
For interpolation modes that work with more than two keyframes (such as splines), c_t would instead need to be defined using that other interpolation function, and the scope of which set of keyframes applies to which range of t would need to be limited accordingly. Exploring this is outside the scope of this thesis.

For any given time t, the surface of C is then defined as all points p where the distance to c_t is the sphere's radius:

$$\|p - c_t\|^2 = r^2 \tag{3.8}$$

Injecting the ray equation (3.3) in place of p in (3.8):

$$\|P + (t - t_0) \cdot v - c_t\|^2 = r^2 \tag{3.9}$$

The squared norm of a vector can be replaced with its dot product with itself ($\|v\|^2 = v \cdot v$). This will allow for the equation to be resolved into a sum of multiple dot products.

$$(P - c_{k_2} + t \cdot v - t_0 \cdot v + m \cdot (c_{k_2} - c_{k_1})) \cdot (P - c_{k_2} + t \cdot v - t_0 \cdot v + m \cdot (c_{k_2} - c_{k_1})) = r^2$$

As vector addition inside dot products is distributive (i.e. $(a + b) \cdot c = a \cdot c + b \cdot c$), this dot product can be resolved into a sum of several dot products, the factors of which don't require further calculations. In order to get the equation's shape closer to that of a polynomial, the fact that scalar multiplication is distributive over addition can be used to factor out t and m from each summand, then group factors that involve the t and m to the same degree. Defining $\Delta c = c_{k_2} - c_{k_1}$ for simplicity, this results in a function of t and m:

$$\begin{aligned} &\|P - c_{k_2}\|^2 - 2 \cdot t_0 \cdot P \cdot v + 2 \cdot t_0 \cdot c_{k_2} \cdot v + t_0^2 \cdot \|v\|^2 - r^2 \\ &+t \cdot 2 \cdot (P \cdot v - c_{k_2} \cdot v - t_0 \cdot \|v\|^2) + m \cdot 2 \cdot (P \cdot \Delta c - c_{k_2} \cdot \Delta c - t_0 \cdot v \cdot \Delta c) \\ &+t \cdot m \cdot 2 \cdot v \cdot \Delta c + t^2 \cdot \|v\|^2 + m^2 \cdot \|\Delta c\|^2 \\ &= 0 \end{aligned} \tag{3.10}$$

Now, to get a polynomial function of t, m needs to be resolved. The equation for variations of this logic that use different interpolation modes will diverge from here on out, but should be trivially solvable depending on the definition of m. Replacing m with its definition from (3.6) for linear interpolation yields the following equation:

$$\begin{aligned} &\|P - c_{k_2}\|^2 - 2 \cdot t_0 \cdot P \cdot v + 2 \cdot t_0 \cdot c_{k_2} \cdot v + t_0^2 \cdot \|v\|^2 - r^2 \\ &+t \cdot 2 \cdot (P \cdot v - c_{k_2} \cdot v - t_0 \cdot \|v\|^2) + \frac{(t_{k_2} - t) \cdot 2 \cdot (P \cdot \Delta c - c_{k_2} \cdot \Delta c - t_0 \cdot v \cdot \Delta c)}{\Delta t} \\ &+\frac{t \cdot (t_{k_2} - t) \cdot 2 \cdot v \cdot \Delta c}{\Delta t} + t^2 \cdot \|v\|^2 + \frac{(t_{k_2} - t)^2 \cdot \|\Delta c\|^2}{\Delta t^2} \\ &= 0 \end{aligned}$$

The partials of the equation created from m can now be resolved into individual parts with different polynomial degrees. The resulting summands can then be grouped by their degree to create a polynomial

$$d_2 t^2 + d_1 t + d_0 = 0 \tag{3.11}$$

with

$$d_2 = \|v\|^2 \cdot \Delta t^2 + \|\Delta c\|^2 - 2 \cdot v \cdot \Delta c \cdot \Delta t \tag{3.12}$$

$$d_1 = 2 \cdot ((P - c_{k_2}) \cdot v \cdot \Delta t^2 - t_0 \cdot \|v\|^2 - (P - c_{k_2} - t_0 \cdot v) \cdot \Delta c \cdot \Delta t + t_{k_2} \cdot v \cdot \Delta c \cdot \Delta t - t_{k_2} \cdot \|\Delta c\|^2) \tag{3.13}$$

$$\begin{aligned} d_0 = &(\|P - c_{k_2}\|^2 + 2 \cdot t_0 \cdot (c_{k_2} - P) \cdot v + t_0^2 \cdot \|v\|^2) \cdot \Delta t^2 \\ &+t_{k_2} \cdot 2 \cdot (P - c_{k_2} - t_0 \cdot v) \cdot dc \cdot \Delta t + t_{k_2}^2 \cdot \|dc\|^2 - r^2 \cdot \Delta t^2 \end{aligned} \tag{3.14}$$

If r also needs to be varied between keyframes, replace r^2 with the according interpolated value $(m - r_{k_1} + (1 - m) r_{k_2})^2$ and resolve accordingly.
The real roots of (3.11) represent all times at which the sphere and ray would theoretically intersect. Note that all roots t where $t < t_{k_1}$ or $t > t_{k_2}$ must be discarded because the surface is

not described by k_1 and k_2 outside the time frame between them. Additionally, all roots where $t \leq t_0$ must be discarded as these intersections would happen before the ray is launched, behind its starting location.
Of the roots remaining after this filter, the lowest result for t is the one where the ray and sphere intersect, assuming the ray does not bounce off of a different object before that. The coordinates at which the intersection takes place can be calculated by calculating $R(t)$.

3.2 Intersection Checks for Surfaces

Similarly to spheres, a polygonal surface S with $o \geq 3$ corners can also be modelled using a set of keyframes k_n, each with points $P_{1..o,k_n}$ for the corners of the surface, with o being consistent between keyframes. As with spheres, intersection calculations need to be run separately for each set of consecutive keyframes k_1 and k_2.
Using m from (3.6) with the same caveats, each point $P_{n,t}, 1 \leq n \leq o$ is calculated as a blend between $P_{n,1}$ and $P_{n,2}$:

$$P_{n,t} = m \cdot P_{n,k_1} + (1 - m) \cdot P_{n,k_2}, 1 \leq n \leq o \tag{3.15}$$

Assuming all points of the polygon are within one surface, the surface equation from (3.1) can be made a function of t by replacing $P_{1..3}$ with their time-dependent counterparts $P_{1..3,t}$:

$$(p - P_{1,t}) \cdot ((P_{2,t} - P_{1,t}) \times (P_{3,t} - P_{1,t})) = 0 \tag{3.16}$$

Injecting (3.3) into (3.16):

$$(P + t \cdot v - t_0 \cdot v - P_{1,t}) \cdot ((P_{2,t} - P_{1,t}) \times (P_{3,t} - P_{1,t})) = 0 \tag{3.17}$$

The vector cross product is distributive over addition (i.e. $(x + y) \times z = x \times z + y \times z$), which can be used to split up the single cross product describing the surface normal into several cross products with single factors, allowing them to be resolved easier:

$$(P + t \cdot v - t_0 \cdot v - P_{1,t}) \cdot (P_{2,t} \times P_{3,t} - P_{2,t} \times P_{1,t} - P_{1,t} \times P_{3,t} + P_{1,t} \times P_{1,t})$$

The cross product of a vector with itself is always a vector of zeroes, which in turn is the identity element for addition (i.e. adding it to another vector just results in the other vector), thus the last of these four cross products ($P_{1,t} \times P_{1,t}$) can be discarded:

$$(P + t \cdot v - t_0 \cdot v - P_{1,t}) \cdot (P_{2,t} \times P_{3,t} - P_{2,t} \times P_{1,t} - P_{1,t} \times P_{3,t}) \tag{3.18}$$

Each of these cross products should be solved individually before resolving the full equation. Using (3.15) to describe a cross product between two generic points $P_{a,t}$ and $P_{b,t}$:

$$P_{a,t} \times P_{b,t} = ((1-m) \cdot P_{a,k_2} + m \cdot P_{a,k_1}) \times ((1-m) \cdot P_{b,k_2} + m \cdot P_{b,k_1}) \tag{3.19}$$

This can again be simplified by using the fact that the cross product is distributive over addition. Additionally, scalar factors can be extracted from cross products as the cross product linearly scales with the individual vectors' magnitude, so for a scalar a and two vectors x, y, $x \times (a \cdot y) = a \cdot (x \times y)$. This can be exploited to resolve the equation to a polynomial function of m:

$$\begin{aligned} P_{a,t} \times P_{b,t} = P_{a,k_2} \times P_{b,k_2} + m \cdot (-2(P_{a,k_2} \times P_{b,k_2}) + (P_{a,k_1} \times P_{b,k_2}) + (P_{a,k_2} \times P_{b,k_1})) \\ + m^2 \cdot ((P_{a,k_2} \times P_{b,k_2}) - (P_{a,k_1} \times P_{b,k_2}) - (P_{a,k_2} \times P_{b,k_1}) + (P_{a,k_1} \times P_{b,k_1})) \end{aligned} \tag{3.20}$$

As with (3.10), to get a function of t, m needs to replaced with its definition ((3.6) in this case):

$$\begin{aligned} P_{a,t} \times P_{b,t} = \\ P_{a,k_2} \times P_{b,k_2} + \frac{(t_{k_2} - t) \cdot (-2(P_{a,k_2} \times P_{b,k_2}) + (P_{a,k_1} \times P_{b,k_2}) + (P_{a,k_2} \times P_{b,k_1}))}{\Delta t} \\ + \frac{(t_{k_2} - t)^2 \cdot ((P_{a,k_2} \times P_{b,k_2}) - (P_{a,k_1} \times P_{b,k_2}) - (P_{a,k_2} \times P_{b,k_1}) + (P_{a,k_1} \times P_{b,k_1}))}{\Delta t^2} \end{aligned}$$

Using the distributive law and the binomial theorem, This can be resolved to a second-degree polynomial

$$P_{a,t} \times P_{b,t} = f_{2,a,b} t^2 + f_{1,a,b} t + f_{0,a,b} \tag{3.21}$$

with

$$f_{2,a,b} = (P_{a,k_2} \times P_{b,k_2}) - (P_{a,k_1} \times P_{b,k_2}) - (P_{a,k_2} \times P_{b,k_1}) + (P_{a,k_1} \times P_{b,k_1}) \tag{3.22}$$

$$\begin{aligned} f_{1,a,b} = -\Delta t \cdot (-2(P_{a,k_2} \times P_{b,k_2}) + (P_{a,k_1} \times P_{b,k_2}) + (P_{a,k_2} \times P_{b,k_1})) \\ -2 \cdot t_{k_2} \cdot ((P_{a,k_2} \times P_{b,k_2}) - (P_{a,k_1} \times P_{b,k_2}) - (P_{a,k_2} \times P_{b,k_1}) + (P_{a,k_1} \times P_{b,k_1})) \end{aligned} \tag{3.23}$$

$$\begin{aligned} f_{0,a,b} = \Delta t^2 \cdot (P_{a,k_2} \times P_{b,k_2}) \\ +t_{k_2} \cdot \Delta t \cdot (-2(P_{a,k_2} \times P_{b,k_2}) + (P_{a,k_1} \times P_{b,k_2}) + (P_{a,k_2} \times P_{b,k_1})) \\ +t_{k_2}^2 \cdot ((P_{a,k_2} \times P_{b,k_2}) - (P_{a,k_1} \times P_{b,k_2}) - (P_{a,k_2} \times P_{b,k_1}) + (P_{a,k_1} \times P_{b,k_1})) \end{aligned} \tag{3.24}$$

Replacing the cross products in (3.18) with the polynomial from (3.21) yields:

$$(P+t\cdot v-t_0\cdot v-P_{1,t})\cdot(f_{2,2,3}t^2+f_{1,2,3}t+f_{0,2,3}-f_{2,2,1}t^2-f_{1,2,1}t-f_{0,2,1}-f_{2,1,3}t^2-f_{1,1,3}t-f_{0,1,3}) = 0$$

Injecting (3.15) for $P_{1,t}$ and introducing $g_n = f_{n,2,3} - f_{n,2,1} - f_{n,1,3}$ and $\Delta P_1 = P_{1,k_2} - P_{1,k-1}$ for readability results in this equation:

$$(P + t \cdot v - t_0 \cdot v - P_{1,k_2} + \frac{t_{k_2} \Delta P_1}{\Delta t} - \frac{t \cdot \Delta P_1}{\Delta t}) \cdot (t^2 g_2 + t g_1 + g_0) = 0 \tag{3.25}$$

Again using the fact that vector dot products are distributive over addition to shape this into a sum of several dot products, then extracting t and grouping the individual summands by their polynomial degree, this can be resolved to a third degree polynomial

$$t^3 d_3 + t^2 d_2 + t d_1 + d_0 = 0 \tag{3.26}$$

With

$$d_3 = g_2 \cdot v - \frac{g_2 \cdot \Delta P_1}{\Delta t} \tag{3.27}$$

$$d_2 = g_2 \cdot P - t_0 \cdot g_2 \cdot v - g_2 \cdot P_{1,k_2} + \frac{t_{k_2} \cdot g_2 \cdot \Delta P_1}{\Delta t} + g_1 \cdot v - \frac{g_1 \cdot \Delta P_1}{\Delta t} \tag{3.28}$$

$$d_1 = g_1 \cdot P - t_0 \cdot g_1 \cdot v - g_1 \cdot P_{1,k_2} + \frac{t_{k_2} \cdot g_1 \cdot \Delta P_1}{\Delta t} + g_0 \cdot v - \frac{g_0 \cdot \Delta P_1}{\Delta t} \tag{3.29}$$

$$d_0 = g_0 \cdot P - t_0 \cdot g_0 \cdot v - g_0 \cdot P_{1,k_2} + \frac{t_{k_2} \cdot g_0 \cdot \Delta P_1}{\Delta t} \tag{3.30}$$

This polynomial can then be solved using a general cubic formula such as the one described by Abramowitz and Stegun [1] or Flocke's Algorithm [8].

As with spheres, the real roots of this equation are all the points in time at which the ray and the surface would meet. Roots t where $t < t_{k_1}$ or $t > t_{k_2}$, as well as roots where $t \leq t_0$, must be discarded as explained above.
For the remaining roots, another check needs to be done for whether the intersection point $R(t)$ is actually inside the polygon, and not just on the same surface. With triangles, this can be easily done using the barycentric coordinates as described above.
The root with the lowest value that satisfies the above conditions again represents the time where the ray and polygon intersect, assuming there is no intersection happening before. If no root satisfies these conditions, no intersection happens.

3.3 Computational Cost

To gauge the additional computation cost for interpolated intersection checks as opposed to static ones, consider the analytical solution for static surface intersection calculations as described in (3.4) and using (3.2).
Naively calculating a cross product requires 6 multiplications and 3 subtractions. The dot product requires just 3 multiplications, as does multiplying a scalar onto a coordinate vector. This means that to calculate the intersection time for a static surface, a total of 15 additions/subtractions, 15 multiplications and 1 division is required. As n is always the same for a surface, it can be cached, reducing the cost to only 6 additions/subtractions, 9 multiplications and 1 division.
For the interpolated surface checks, calculating $f_{0..2,a,b}$ as per (3.22) to (3.24) alone already takes 71 additions/subtractions and 124 multiplications. As $g_{0..2}$ takes 3 sets of f, it thus requires 213 additions/subtractions and 372 multiplications. Since $g_{0..2}$ remains the same for a pair of surface keyframes independently of the incoming ray, it can be cached for each keyframe pair. The same applies for Δt and ΔP_1, which otherwise would introduce one and three subtractions respectively.
Then calculating the surface intersection as per (3.26)-(3.30) requires 14 additions/subtractions, 42 multiplications and 3 divisions, assuming that $g_{0..2} \cdot v$ and $\frac{g_{0..2} \cdot \Delta P_1}{\Delta t}$ are only calculated once each and then reused.
Additionally, this requires resolving the resulting polynomial, the cost of which is implementation dependent, but also rather high compared to the static check. As a reference, in the worst case of none of the polynomial's factors being 0, the Rust library `roots` (as of version 0.0.8) used in the proof-of-concept, requires 13 additions/subtractions, 58 multiplications and 12 divisions, plus trigonometric and square/cube root functions, the computation cost of which depends

on the CPU architecture and cannot be reduced to a number of simple operations.
In total, the amount of calculations, especially multiplications, to perform balloons up with this more complex method.

3.4 Looping Scenes And Full Algorithm

A full algorithm to check for intersections then needs to traverse every pair (or set, for interpolation modes working with more than two keyframes) of keyframes and run the intersection check as described above.
For a surface, this algorithm looks as follows:

```
// define keyframes and ray here
for index in 0..(keyframes.len - 1) {
    k1 = keyframes[index];
    k2 = keyframes[index+1];
    (d3, d2, d1, d0) = polynomial_parameters(k1, k2);
    // get all potential intersections
    roots = solve_cubic_function(d3, d2, d1, d0);
    intersection_time = null;
    for root in roots {
        // filter intersections
        if root < ray.launch_time {
            continue;
        }
        if root < k1.time || root > k2.time {
            continue;
        }
        // potential further checks here

        if intersection_time == null
            || root < intersection_time {
                intersection_time = root
        }
    }
    // no need to check future keyframes
```

```
    // if an intersection was found
    if root != null {
        return root;
    }
}
// no intersection
return null;
```

Chapter 4

Optimisation through spatial and temporal division

As Whitted already noted in 1980 [15], even in a ray tracing system with static checks, intersection calculations take up the vast majority of processing time (between 75-95% in Whitted's case). Since this effect will only increase with more expensive intersection checks as discussed in section 3.3, the amount of checks run per ray should be reduced as much as possible. A method for this will be evaluated in this chapter.

4.1 Chunks vs. Bounding Volumes

One common optimisation for ray tracing systems is to limit the amount of intersection calculations by eliminating objects the ray cannot intersect with in a simpler way before running proper checks. There are two general sets of methods used for this:
Bounding Volume Hierarchies (BVHs), first proposed by Clarke [4], work by enclosing each object in the scene within a volume containing it. This bounding volume uses a simpler geometric primitive that allows for faster intersection checks than the object itself, usually quadric surfaces or spheres. These bounding volumes are then grouped into bigger bounding volumes, forming a hierarchical tree structure. Rays then walk down the tree structure, checking for intersections with the corresponding bounding volumes. If a ray does not intersect with a branch's bounding volume, any objects within that branch can be ignored for further intersection checks.
Another method first proposed as a Three Dimensional Digital Differential Analyzer by Fujimoto and Iwata [7], instead divides a scene into separate cells (chunks), with each chunk keeping a list of which objects are inside it. Rays can then traverse from chunk to chunk along their trajectory and only check for intersections with the objects contained in the chunk they're cur-

rently in.
Since objects can move around the scene, using one of these methods without changes becomes inefficient. If, for example, a receiver moves from one end of the scene to the other over the course of ten seconds, its bounding volume would extend over all of that distance for the entirety for the scene, despite it not touching the majority of it for the most part. Similarly, it would be kept in its starting position's chunk for the entirety of the scene despite leaving that area very early, making for needless intersection checks if a ray enters that area at a later time.
For this use case, chunks become a lot more efficient than BVHs: When taking movement over time into account, each object would need separate bounding volumes for separate segments of time, forcing a ray to not just check one bounding volume, but multiple per object. This also means that in order to be able to create meaningful bounding volume hierarchies, each object's bounding volumes would need to be separated at the same points in time, which can lead to redundancies if objects move at different times. Calculating a useful BVH becomes near impossible.
The amount of chunks, in turn, does not change: They can be adapted simply by storing not just which objects are inside them, but also when each object enters and exits the chunk. If chunk contents are calculated correctly, this means that no intersection checks take place for objects that aren't inside the given chunk at the given time.

4.2 Data Structure

In a simple system, a chunk stores a list where each entry represents an object inside it. To accommodate for objects moving in and out of chunks, entries will instead contain three fields: One holding the index of the object in question, one holding the time at which the object enters the chunk and one holding the time at which the object leaves the chunk. Since the latter two fields might both be optional if the scene starts/ends with the object inside the scene, this can be nicely represented using a sum type such as Rust's Enumerators or C's union types with different states:

```
// Object stays within chunk for the whole scene
// only store the index
Static(object)
// Object enters and exits chunk at the given times
Dynamic(object, time_entry, time_exit)
// Object enters chunk at the given time
```

```
// and stays until the end
Final(object, time_entry)
```

As the scene's start time is known and the state of objects before it is irrelevant, a state containing only an exit time is not necessary as it can be modelled using the `Dynamic` state with a `time_entry` matching the scene's starting time. Using a more common product type system, chunk entries can instead be represented as a struct or class where the entry and exit times are optional or nullable fields.
A ray traversing this scene can now simply check when it enters and exits a chunk and pick out the objects to check for intersections with accordingly. When using sum types, the space requirements for static objects only increase by one byte denoting the type's variant (with even that potentially getting left out, as Herzog showed [10]). For moving objects, only up to two additional fields plus the variant field are required, with the timestamp fields' size depending on the implementation. Compared to the performance gains from avoiding needless intersection checks, this additional space requirement is comparatively minimal.

4.3 Calculating Chunks

Before shooting rays through a scene, all objects must be stored correctly within their respective chunks. The input to a chunk calculation algorithm would thus be a set of n surfaces $S_{0..(n-1)}$, each with a varying number m of keyframes $K_{0..(m-1)}$. The information held by keyframes is the same as defined in section 3.2.
Additionally, the starting coordinates of the very first chunk `xmin, ymin, zmin` as well as the chunk sizes `wx, wy, wz` need to be known. For the starting coordinates, the lowest coordinates any object in the scene ever has can be used. The chunk sizes can be determined by determining a number of chunks to use for calculation, then calculating the difference between the scene's lowest and highest coordinates in each dimension and dividing that by the number of chunks. To avoid errors with objects being at the very edge of the last chunk, it is recommendable to slightly pad out the scene's lowest and highest coordinates from the actual extremes found in objects.
A naive algorithm to place a surface in its appropriate chunks could then look like this:

```
for m in 1..(surface.num_keyframes - 1) {
    keyframe_first = surface.keyframes[m-1];
    keyframe_second = surface.keyframes[m];
    // find the highest and lowest x, y and z values
```

```
    (max_coords, min_coords) = find_max_and_min_coords(
        keyframe_first.points,
        keyframe_second.points
    );
    // floor() to round down to the index of the chunk
    chunk_min_x = floor((min_coords.x - xmin) / wx);
    chunk_min_y = floor((min_coords.y - ymin) / wy);
    chunk_min_z = floor((min_coords.z - zmin) / wz);
    chunk_max_x = floor((max_coords.x - xmin) / wx);
    chunk_max_y = floor((max_coords.y - ymin) / wy);
    chunk_max_z = floor((max_coords.z - zmin) / wz);
    for x in chunk_min_x..chunk_max_x {
        for y in chunk_min_y..chunk_max_y {
            for z in chunk_min_z..chunk_max_z {
                chunks[x][y][z].add_surface_entry(
                    {
                        time_start: keyframe_first.time,
                        time_end: keyframe_second.time,
                        index: surface.index
                    }
                );
            }
        }
    }
}
```

Note that to avoid errors after the last keyframe's time, the last keyframe then needs to also be processed on its own, with each chunk touched by this last keyframe's version of the surface getting an according `Final` entry. This will be omitted from the algorithms' pseudocode for brevity.

This naive approach comes with the problem that if an object traverses a long distance between two keyframes, it will again be needlessly included in all chunks traversed for the entirety of its lifetime. An easy solution for this would be to insert 'pseudo-keyframes' interpolated from the actual keyframes and running the above calculation between those.

The naive way to determine the position of these pseudo-keyframes while avoiding wrong chunk

entries would be to move through time from the first keyframe's time to the second keyframe's time, then write new chunk entries whenever the chunks at the given time change:

```
function add_to_chunks(key_first, key_second) {
 last_time = key_first.time;
 time = key_first.time + 1;
 key_middle = interpolate(key_first, key_second, time);
 while time != key_second.time {
  if key_middle.chunks() != key_first.chunks() {
   write_chunks(
                key_first.time,
                time - 1,
                index,
                key_first.chunks()
            );
   key_first = key_middle;
   last_time = time;
  }
  time += 1;
  key_middle = interpolate(key_first, key_second, time);
 }
 write_chunks(last_time, time, index, key_second.chunks);
}
```

This is obviously inefficient performance wise as keyframes need to be interpolated for every single step despite the chunks potentially staying the same for many of said steps. A more efficient approach could use a divide-and-conquer approach, continually halving the range between the middling and first keyframe until they are the same, somewhat akin to binary search:

```
function add_to_chunks(key_first, key_second) {
 time = key_first.time
 while time != key_second.time {
  time = avg(key_first.time, key_second.time);
  key_middle = interpolate(key_first, key_second, time);

  while key_middle.chunks() != key_first.chunks() {
```

```
    time = avg(key_first.time, time);
    key_middle = interpolate(key_first, key_second, time);
   }
   while key_middle.chunks() == key_first.chunks
             && time != key_second.time {
    time += 1;
    key_middle = interpolate(key_first, key_second, time);
   }
   write_chunks(
             key_first.time,
             time - 1,
             index,
             key_first.chunks()
          );

   key_first = key_middle;
  }
}
```

This can be optimised further by fully behaving like binary search, always halving the range between the first and middle keyframe in both directions, but in the tests performed for this thesis, this version of the algorithm was already fast enough for its time cost to be neglegible compared to the intersection check/bouncing time.

4.4 Traversing Chunks

When using chunks, shooting rays no longer simply entails checking for intersections with all objects and choosing the earliest one. Instead, the traversal of the ray through the separate chunks is modelled, then intersection checks are performed for the objects in each chunk the ray enters.

Many algorithms have been developed to model rays moving from one chunk to the next. Most of them keep track of the chunk the ray is in as well as the distance the ray has already travelled in one form or another. Adapting them to time-based chunks thus requires also keeping track of the time that has elapsed additionally to, or instead of, the travel distance. Additionally, the intersections within a chunk can no longer be calculated when moving to it, as the exit time is

not known at this point. Instead, intersection checks need to be performed when moving on to the next chunk.
For this thesis and the proof-of-concept implementation, a slightly simplified version of the algorithm by Cleary and Wyvill [5] (called CW88 in this thesis) will be adapted.
CW88 works by keeping track of the distance a ray needs to travel until it arrives at the next chunk in each dimension using three variables `dx, dy, dz` as well as keeping track of the distance a ray needs to travel between two chunk borders in the same dimension in three additional variables `deltax, deltay, deltaz`. The `delta*` variables are initialised based on the ray's direction cosines, with `d*` being initialised as the percentage of the corresponding `delta*` variable relevant to the starting chunk. The bounds of the scene in each dimension are stored in variables `sx, sy, sz`. Once `dx >= sx` or the equivalent in another dimension becomes true, the ray has gone out of bounds without hitting any object.
Chunks are stored in a one-dimensional array, where a chunk at a given x-, y- and z- index is found at the index $x \cdot n^2 + y \cdot n + z$, with n being the number of chunks in each dimension. This array, however, only holds a single value per chunk indicating whether it contains objects to begin with. The chunks that do contain objects are stored in a hash map or similar data structure, using the array index as a key.
This index isn't newly calculated with every step. Instead, it is stored in a variable `p`, with three variables `px, py, pz` storing the value to add to the index when entering a new chunk in the given dimension.
All in all, the original CW88 algorithm works as follows, with initialisation of y- and z-related variables being omitted for brevity:

```
// init cx as direction cosine for x
// init chunkx, chunky, chunkz as chunk index
p = chunkx * pow(n, 2) + chunky * n + chunkz;
mx = pow(n,2);
chunk_start = xmin + chunkx * wx;
if cx == 0 {
 // no movement in x direction
 dx = MAX_INT;
 sx = 0;
} else {
 if cx > 0 {
  deltax = wx/cx;
  px = mx;
```

```
  dx = (chunk_start + wx - x) / wx * deltax;
  sx = (n-chunkx) * deltax;
 } else {
  deltax = -wx/cx;
  px = -mx;
  dx = (x - chunk_start) / wx * deltax;
  sx = (chunkx + 1) * deltax;
 }
}
// init y and z variables by the same schema here
while true {
 if dx <= dy && dx <= dz {
  if dx >= sx {
            return null;
        }
  p += px;
  dx += deltax;
 } else if dy <= dx && dy <= dz {
  if dy >= sy {
            return null;
        }
  p += py;
  dy += deltay;
 } else {
  if dz >= sz {
            return null;
        }
  p += pz;
  dz += deltaz;
 }
    if !chunk_contains_objects(p) {
        continue;
    }
    intersection = intersection_check(p);
    if intersection != null {
```

```
        return intersection;
    }
}
```

In order to use CW88 with time-based chunks, new variables are introduced to also keep track of the elapsed time. `tx, ty, tz` become the equivalent of `dx, dy, dz` for time while `deltatx, deltaty, deltatz` are the timed version of `deltax, deltay, deltaz`. `deltatx, deltaty, deltatz` are simply initialised as `deltax/velocity, deltay/velocity, deltaz/velocity`, with `velocity` representing the ray's velocity, usually the speed of sound. `tx` is initialised as `dx/velocity + t0`, with t0 being the ray's starting time. `ty` and `tz` can be initialised by the same scheme.

As distance travelled and time passed directly scale, the `deltat*` and `t*` variables could be omitted with `t*` being calculated from `dx` whenever needed. In practice, keeping them as a separate variable means that updating `d*` only takes a single addition per step, whereas the cost of calculating `t*` from `d*` every step would take both an addition and a division.

This does however allow for removal of both `delta*` and `d*` as `deltat*` and `t*` can be used to track the distance to the next chunk in the same way. The `s*` variables then need to be calculated using `deltat*` rather than `delta*`.

Another necessary change is that, as discussed above, the intersection check done at the end of the loop is not done for the newly entered chunk that matches `p`, but the chunk the ray just left instead. In order to then keep the time the ray entered the last chunk, a new variable `tlast` is introduced and initialised as the ray's starting time.

The full traversal algorithm, omitting all variable initialisation for brevity, then looks as follows:

```
while true {
 if tx <= ty && tx <= tz {
        intersection = traverse_chunk(p, tx, tlast);
        if intersection != null {
            return intersection;
        }
        tlast = tx;
  p += px;
  tx += deltatx;
  if tx >= sx {
            return null;
```

```
        }
 } else if ty <= tx && ty <= tz {
        intersection = traverse_chunk(p, ty, tlast);
        if intersection != null {
            return intersection;
        }
        tlast = ty;
  p += py;
  ty += deltaty;
  if ty >= sy {
            return null;
        }
 } else {
        intersection = traverse_chunk(p, ty, tlast);
        if intersection != null {
            return intersection;
        }
        tlast = tz;
  p += pz;
  tz += deltatz;
  if tz >= sz {
            return null;
        }
 }
    if !chunk_contains_objects(p) {
        continue;
    }
    intersection = intersection_check(p);
    if intersection != null {
        return intersection;
    }
}

function traverse_chunk(p, t_dimension, tlast) {
    if !chunk_contains_objects(p) {
```

```
            continue;
        }
        intersection = intersection_check(p);
        if intersection != null
            && intersection.time <= t_dimension
            && intersection.time >= tlast {
            return intersection;
        }
        return null;
}
```

Intersections that are outside of the entry/exit time for the current chunk are discarded. This is to avoid errors when a ray would intersect with the given object in a chunk it will enter later, but would also hit another object before it. If the other object is not inside the chunk the ray starts out in, but the object that would get hit later is, that later intersection would otherwise incorrectly be returned without ever checking for an intersection with the other object. This error can also occur with the original CW88 algorithm, but incidentally becomes easier to fix through this adaptation.

Chapter 5

Evaluation

In order to evaluate the effectiveness of the newly developed interpolating intersection logic, three test cases will be used and compared to the snapshot method.
The first test case is simple: A receiver moves towards an emitter at 1/9th the speed of sound. It starts 342.2 meters away from the emitter. The emitter sends out a sine wave at 440Hz for 1 second. No other objects are in the scene.
This is essentially the example described in 3 and is used to demonstrate the basic differences between interpolated and snapshot methods.
The second test case takes place inside a rotating rectangular room. Said room is 4 meters in width and length and 3 meters tall. The receiver is in the very middle of the room, with the emitter being 1.2 meters above it. The room rotates around the Z-axis once per second.
This is used to test whether the differences between interpolated and snapshot methods lead to a notable difference in a slightly more practical scenario.
The third test case takes place inside a rotating L-shaped room, as denoted in IMAGE, with the receiver being at the origin and the emitter being 0.5 meters above it. The room again rotates around the Z-axis, but takes three seconds for a full rotation.
This case is used to demonstrate the shortcomings of the interpolating intersection logic and the need for further research for realistic simulations.

5.1 Testing Conditions

Tests were performed on a stock AMD Ryzen 3600xt CPU with 16GB of 3200MHz DDR4 RAM. All 12 logical cores are used for parallel processing.
The proof-of-concept used for testing was written in Rust, using the `nalgebra` crate for algebra functions, `roots` for polynomial solving and `rayon` for parallelisation.

The first test case with the approaching receiver is run using one ray per sample as it only attempts to simulate sound travelling from emitter to receiver without considering bounces from a room around it. The ray is always directed at the receiver.
The other two test cases are run using 10 million rays per impulse response, which is enough to get an approximate T_{60} accurate to the 10th of a second. This is sufficient to draw conclusions about the scenes while requiring a reasonable amount of compute time. To account for variance introduced by randomness, 3 runs are done per simulation method and scene.
For the two rotating scenes, only one impulse response is calculated and applied to all samples. This is because the starting condition of a rotating room is always the same: As both rooms rotate around the emitter's and the receiver's position and the emitter emits rays randomly in all directions, the relative position of the room to the receiver and emitter is always the same, making for the same impulse response at all times.
Simulations would become much more expensive for scenes where this condition doesn't apply, as that would mean calculating individual impulse responses for each sample. For a 1 second sample at 44100 KHz and a roughly 20 minute IR calculation time (rounded down from the snapshot method test results below), this would take $44100 \cdot 20m = 882000m = 14700h = 612.5d$. Running simulations for such scenes would require optimisations, much stronger hardware and/or conceits on ray count to be able to run in a reasonable time.
All walls in scenes that contain any have a material roughly resembling smooth concrete walls' behaviour for high frequencies. The absorption coefficient, based on data by acoustic.ua (`https://www.acoustic.ua/st/web_absorption_data_eng.pdf`), is 0.02, meaning that a ray retains 98% of its energy after bouncing. Data for diffusion coefficients is not publicly available, so by guesswork, a value of 0.1 was used, meaning that 90% of the energy is specularly reflected, while 10% is diffusely reflected. In practice, this means that rays have a 10% chance to bounce in a random direction rather than reflecting off the surface normally.

5.2 Approaching Receiver Scene

In theory, two effects should be observable here.
Firstly, the ray would take 1 second to arrive at the emitter's position. As the emitter is travelling towards the ray and covering a tenth of the distance in the time the ray travels the remainder, the sound should already start at 0.9 seconds.
Secondly, due to the emitter's fast movement, the doppler effect would lead to a shift in frequency.

Using the well-known doppler effect formula with a propagation speed $c = 342.2m/s$, an emitter speed $v_s = 0m/s$, a receiver speed $v_r = 342.2/9m/s = 38.0\bar{2}m/s$ and a base frequency $f_0 = 440Hz$, the resulting frequency can be calculated as

$$f = \frac{c + v_r}{c + v_s} f_0 = \frac{342.2m/s + 38.0\bar{2}m/s}{342.2m/s} \cdot 440Hz = 488.\bar{8}Hz \tag{5.1}$$

As expected, the first effect can be observed in the interpolated version of the simulation, but not in the simulation using the snapshot method.
This is because for the snapshot method, only the initial position of the receiver is relevant to the distance a ray needs to travel until it arrives at said receiver. Thus, the ray has to travel the full 343.2 meters for the initial impulse response, as shown in 3.
The interpolated version instead takes this effect into account correctly.
One notable detail is that the interpolated method also simulates the doppler effect more accurately. The resulting signal from the interpolated method exactly matches the $488.\bar{8}Hz$ calculated in (5.1), while the snapshot method instead arrives at a frequency of approximately $495Hz$.
Performance wise, the simulations barely differ. This is presumably because each impulse response is calculated using only a single ray which only needs to check for intersections with a single object, rendering the increased intersection check costs insignificant.
Additionally, the snapshot method implementation has the overhead of re-calculating the chunks for each snapshot scene. This is insignificant in simulations where many rays are used as the chunk calculation cost is small by comparison, but for this single-ray simulation, the overhead evens out the performance gained from having a cheaper intersection calculation.
A noteworthy implementation detail becomes apparent from the waveform resulting from the simulation: As the input signal is a single sine wave and the doppler effect would only raise its frequency, the resulting wave should still be a pure sine wave. In contrast, the actual result for both simulation types features aliasing effects.
This is because the simulation is run as the same sample rate as the input signal. For each input sample, one impulse response is calculated. As the doppler effect speeds up the incoming signal, but the sped up signal is not upsampled accordingly, thus creating the same effect as if the signal was downsampled and stayed at the same frequency.
To alleviate this, one would need to either run the signal through a low-pass filter ahead of time or have both the input signal and simulation at a higher sample rate, then filter and downsample to the target sample rate afterwards.
The former works, but might damage a signal more complex than a sine wave and requires knowledge of the scene ahead of time. For a more complex scene where the downsampling

Run	Snapshot 1	Snapshot 2	Snapshot 3	Interp. 1	Interp. 2	Interp. 3
Time	0:22:35	0:22:21	0:22:22	4:36:24	4:21:59	TODO
T_{60}	5.23s	5.16s	5.19s	5.23s	5.27s	TODO

Table 5.1: Rotating Cube Test Results

Run	Snapshot 1	Snapshot 2	Snapshot 3	Interp. 1	Interp. 2	Interp. 3
Time	0:24:10	0:24:11	0:23:48	7:00:35	7:00:56	7:00:14
T_{60}	5.31s	4.98s	4.99s	5.07s	5.35	5:44

Table 5.2: Rotating L-Shaped Room Test Results

effect cannot be easily calculated prematurely, this approach becomes unusable.
The latter leads to increased computation cost as more impulse responses need to be calculated for more signals, but works with any arbitrary scene.

5.3 Rotating Cube Scene

There is very little audible difference between the snapshot and interpolated version of this scene. This was to be expected: Since the distance sound travels from wall to wall hardly changes for a rotating scene and the angle it bounces at becomes insignificant as the reflection becomes more diffuse over time, the late reverberation in the scene becomes almost the same.
Still, a minor difference can be observed for the reverberation time T_{60}, as listed in 5.3: In the interpolated simulation, T_{60} is slightly higher, so the reverberation rings out for longer.
As rays still lose the same amount of energy per bounce and thus take the same amount of bounces until their energy is depleted, this implies rays take a longer time between bounces on average, which in turn means they generally travel a longer distance between bounces.
This could partially be due to a bounce into a corner, which would normally make for two bounces off each side of the corner, becoming a single bounce as the other side of the corner moves out of the way before a ray can reach it.
Performance wise, the interpolated method expectedly is much more expensive. As seen in 5.3, the computation time increases roughly tenfold. This was expected as discussed in section 3.3

5.4 Rotating L-Shaped Room Scene

Note that this scene shows a large limitation of the interpolated simulation as it stands: As the back surface of the room is rotating in a circle, it is possible for it to hit a ray while that ray

is moving away from it. This would not happen in the real world as pressure gradients would guide the sound wave away from the wall.
The behaviour for sound waves getting hit by a wall from behind is thus not defined by physics. For the sake of this simulation, the rays are discarded as they can't bounce off the wall. This case can be made less likely by having the room rotate at a slower speed: The lower the rotation speed, the flatter the angle rays have to bounce off the wall at to be able to then get hit by it from behind. That solution, however, is antithetical to the goal of this research, which is to simulate rapidly rotating rooms.

Chapter 6

Outlook and Further Research

As mentioned in the scope section, the research from this thesis alone is not yet sufficient to fully simulate rotating rooms and other scenes involving rapid movement as the physical forces created in those scenarios and their effect on the sound waves are disregarded.
In the case of rotating rooms in particular, the centrifugal force created by the rotation would lead to pressure gradients within the room. Exploring the exact effect those pressure gradients have on sound waves and how to simulate this using geometric methods would require collaboration between physicists and computer scientists.
As a general outline, air pressure differences probably have an effect on the propagation speed and direction of sound waves. With the rotating L-Shaped room used as one of the examples for this thesis, this implies that sound waves are directed away from the rotating walls. A ray tracer would thus presumably have to calculate the pressure gradients ahead of time, then have the rays travel in curved rather than linear trajectories accordingly.
Since the ray equation used to derive the intersection check equations in chapter 3 would thus change, new equations would have to be derived using a different formula to model the ray according to the changed requirements. The derivation can still follow the same scheme as the one shown in this thesis. For surface intersection checks, only the last part of the derivation from (3.25) onward (after already having calculated and substituted $g_{0..2}$) would diverge from the derivation in this thesis.
Another field for further research could be real-time applications. The intersection logic developed for this thesis can only work for pre-calculated rooms as all movements in the scene need to be known ahead of time, rendering dynamic scenes such as ones developing in real time incompatible.
A dynamic approach could work by not calculating the rays' entire movements at emission time, but instead keeping track of all rays moving through the scene at a current point and incremen-

tally continuing each ray's journey through the updating scene at recurring intervals.
Considering the higher cost for intersection checks, the additional memory costs to keep track of rays and the requirement for physics calculations for a proper realistic simulation as outlined above, performance might also become a problem for real-time scenes even when using a fast simulation method such as Chandak et al.'s AD-Frustum [3], and new optimisations will be needed.

Bibliography

[ASS48] M. Abramowitz, United States. National Bureau of Standards, and I.A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Applied Mathematics Series. U.S. Government Printing Office, 1948.

[BVD20] D. Bilibashi, E. M. Vitucci, and V. Degli-Esposti. "Dynamic Ray Tracing: Introduction and Concept". In: *2020 14th European Conference on Antennas and Propagation (EuCAP)*. 2020, pp. 1–5. DOI: 10.23919/EuCAP48036.2020.9135577.

[Cha08] A. Chandak, C. Lauterbach, M. Taylor, Z. Ren, and D. Manocha. "AD-Frustum: Adaptive Frustum Tracing for Interactive Sound Propagation". In: *IEEE Transactions on Visualization and Computer Graphics* 14.6 (2008), pp. 1707–1722. DOI: 10.1109/TVCG.2008.111.

[Cla76] J.H. Clark. "Hierarchical geometric models for visible surface algorithms". In: *Commun. ACM* 19.10 (Oct. 1976), pp. 547–554. ISSN: 0001-0782. DOI: 10.1145/360349.360354.

[CW88] J.G. Cleary and G. Wyvill. In: *The Visual Computer* 4 (1988), pp. 65–83. DOI: 10.1007/BF01905559.

[Eri04] Christer Ericson. *Real-Time Collision Detection*. Crc Press, 2004.

[FI85] A. Fujimoto and K. Iwata. "Accelerated Ray Tracing". In: *Computer Graphics*. Ed. by Tosiyasu L. Kunii. Tokyo: Springer Japan, 1985, pp. 41–65. ISBN: 978-4-431-68030-7.

[Flo15] N. Flocke. "Algorithm 954: An Accurate and Efficient Cubic and Quartic Equation Solver for Physical Applications". In: *ACM Trans. Math. Softw.* 41.4 (Oct. 2015). ISSN: 0098-3500. DOI: 10.1145/2699468.

[Hav00] V. Havran. "Heuristic Ray Shooting Algorithms". PhD thesis. Czech Technical University, Prague, Nov. 2000.

[Her] B. Herzog. *Rust Binary Analysis, Feature By Feature*. https://research.checkpoint.com/2023/rust-binary-analysis-feature-by-feature/. Accessed: 2024-02-16.

[Rag10] N. Raghuvanshi, J. Snyder, R. Mehra, M. Lin, and N. Govindaraju. "Precomputed wave simulation for real-time sound propagation of dynamic sources in complex scenes". In: *ACM SIGGRAPH 2010 Papers*. SIGGRAPH '10. Los Angeles, California: Association for Computing Machinery, 2010. ISBN: 9781450302104. DOI: 10.1145/1833349.1778805.

[SM09] P. Shirley and S. Marschner. *Fundamentals of Computer Graphics*. Taylor & Francis, 2009. ISBN: 9781568814698.

[SS15] L. Savioja and U.P. Svensson. "Overview of geometrical room acoustic modeling techniques". In: *The Journal of the Acoustical Society of America* 138.2 (Aug. 2015), pp. 708–730. ISSN: 0001-4966. DOI: 10.1121/1.4926438.

[Tho17] Matthew Reuben Thomas. "Wayverb: A Graphical Tool for Hybrid Room Acoustics Simulation". MA thesis. University of Huddersfield, Jan. 2017. URL: http://eprints.hud.ac.uk/id/eprint/33919/.

[Whi80] T. Whitted. "An Improved Illumination Model for Shaded Display". In: *Communications of the ACM* 23.6 (1980), pp. 343–349. DOI: 10.1145/358876.358882.

List of Figures

List of Tables