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 alleviate this problem, this thesis proposes a method to simulate the acoustics of a moving room without errors.

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. Additionally, a method is developed to to losslessly and efficiently store the multiple impulse responses created by re-calculating the impulse responses for different points in time. The goal of this research is to lay the groundwork 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. 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 rome 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. A real-time approach could work by not calculating the rays' entire movements at emission time, but instead keeping track of all moving rays and incrementally continuing their journey through the now updated scene at recurring intervals.

1.2 Outline

Fundamentals ch2
intersecion checks ch3
chunking ch4
irs ch5
eval ch6

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.

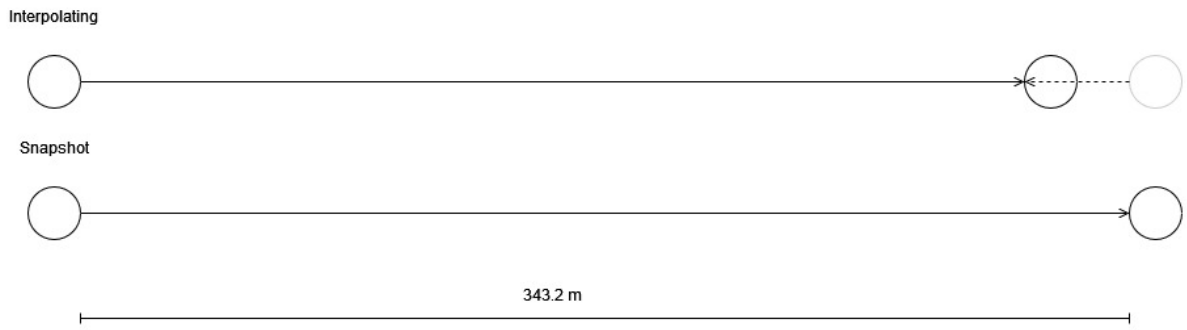

Figure 3.1: Difference between ray travelling distance using the newly developed interpolating method (top) as opposed to the snapshot method (bottom). In the interpolated version, the ray only travels part of the distance as the receiver travels the remainder.

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/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, 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. 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 it 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.
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 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 can instead contain three fields: One containing the index of the object in question, one containing the time at which the object enters the chunk and one containing 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

4.4 Traversing Chunks

Chapter 5

Time-Based Impulse Responses

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. URL: https://doi.org/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. URL: https://doi.org/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. URL: https://doi.org/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. eprint: https://pubs.aip.org/asa/jasa/article-pdf/138/2/708/13242401/708_1_online.pdf. URL: https://doi.org/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