



UHASSELT



Maastricht University

KNOWLEDGE IN ACTION

Faculteit Wetenschappen **School voor Informatietechnologie**

master in de informatica

Masterthesis

GPU-accelerated sonar mapping

Brent Berghmans

Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :

dr. Jeroen PUT

De transnationale Universiteit Limburg is een uniek samenwerkingsverband van twee universiteiten in twee landen: de Universiteit Hasselt en Maastricht University.



UHASSELT

KNOWLEDGE IN ACTION

www.uhasselt.be

Universiteit Hasselt
Campus Hasselt:
Martelarenlaan 42 | 3500 Hasselt
Campus Diepenbeek:
Agoralaan Gebouw D | 3590 Diepenbeek

2021
2022



Maastricht University

Faculteit Wetenschappen ***School voor Informatietechnologie***

master in de informatica

Masterthesis

GPU-accelerated sonel mapping

Brent Berghmans

Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :

dr. Jeroen PUT

UNIVERSITEIT HASSELT

MASTERPROEF VOORGEDRAGEN TOT HET BEHALEN VAN DE GRAAD
VAN MASTER IN DE INFORMATICA

GPU-Accelerated Sonel-Mapping

Auteur:

Berghmans Brent

Promotor:

Dr. Michiels Nick

Co-promotor:

Dr. Put Jeroen

Begeleider(s):

De heer Moonen Steven

Academiejaar 2021-2022



Acknowledgement

I would like to thank everyone who has supported me through the completion of my thesis. First is dr. Michiels Nick whom I would like to thank for their support and help, especially during the start of the thesis. Next, I'm extremely grateful for the support of dr. Put Jeroen. Without their support and encouragement I don't think I would have been able to complete this thesis. Finally, I would also like to express my gratitude to Moonen Steven for their unending support and willingness to assist me on short notice.

Summary

This thesis takes a look at the sonel mapping technique developed by Bill Kapralos, Michael Jenkin and Evangelos Milios.[4] Sonel mapping is a technique based on Monte-Carlo simulations. The technique is able to provide an estimate for the acoustic simulation of a room. The idea of a GPU-accelerated version of the sonel mapping technique is explored as an answer to the goal of developing an acoustic simulation technique that is fast enough to be used in interactive media such as video games. The technique is examined, after which the implementation of an GPU-accelerated version of the technique is detailed alongside additions and changes that were added as a result of implementing it on a GPU. The development of the implementation was done with the OptiX ray tracing framework in order to make use of hardware-accelerated ray tracing. Furthermore, the implementation is analysed by performing a series of tests. After which, an evaluation of accuracy and performance is made alongside a list of bottlenecks and future improvements. The accuracy of the implementation is evaluated by comparing it to the open-source room acoustic software i-Simpa. From the results of this analysis we conclude that improvements to the algorithm need to be made as it does not converge to a correct estimate. However, it is shown that the GPU-accelerated implementation, with the right configuration, does reach speeds sufficient for interactive purposes.

Contents

1	Introduction	5
2	Background	7
2.1	Computer Graphics	7
2.1.1	Path Tracing	8
2.1.2	The Rendering Equation	9
2.1.3	Photon-Mapping	10
2.2	Room Acoustic Simulation	11
2.2.1	Wave Solvers	11
2.2.2	Geometric Acoustics	11
2.3	Stochastic Sonel Mapping	12
2.3.1	Sonel Tracing	12
2.3.2	Sonel Gathering	13
3	Implementation	16
3.1	Scene Setup	16
3.1.1	3D Environment	16
3.1.2	Sound Sources	16
3.1.3	OptiX Setup	17
3.2	Sonel Tracing	18
3.3	Sonel map and Sound sources	25
3.4	Sonel Gathering	26
3.5	Echogram construction	29
3.6	Validation	31
3.6.1	i-Simpa	31
3.6.2	Test Setup	31
3.6.3	Echogram Validation	32
3.6.4	Performance Analysis	36
3.6.5	Performance comparison with i-Simpa	43
4	Conclusion	45
5	Appendix	47
5.1	Code	47

Chapter 1

Introduction

The world of room acoustic simulations can be split into two methodologies. On the one hand we have wave solvers that simulate the physical properties of sound waves in order to simulate sound. Whilst this is the most accurate method to simulate acoustics it is also a computational expensive task to perform. The other approach to simulating room acoustics is the use of ray tracing. Ray-tracing is gathering a lot of attention lately because of the introduction of ray acceleration hardware implemented in modern graphics processing units (GPUs). The technique is known for creating photo-realistic images but a lesser known use-case is room acoustics.

Developed as an alternative to wave solvers, ray tracing for room acoustics was introduced as a computational feasible method for simulating sound, this branch of room acoustics falls under the category of geometric acoustics. Geometric acoustics methods are less accurate than wave solvers but they are significantly faster to compute and the results are accurate enough to provide useful information about the sound propagation in the simulated environment.

Research of GPU-accelerated ray tracing in computer graphics has been advanced significantly in the past couple of years. These advances have made their way into commercial software and have allowed creators to decrease their iteration time which allows for more iterations and better quality of work. The aim of this thesis is to aid in the advances of geometric acoustics techniques to provide similar improvements to the workflow of sound engineers.

In this thesis we will be taking a look at a specific geometric acoustics technique called sonel-mapping which is similar to the photo-realistic algorithm of photon-mapping.[3][4] The technique is a good choice for GPU acceleration seeing as photon-mapping has been implemented on GPUs already.[5][6][7] Recent research has also provided an innovative technique to accelerate the nearest-neighbour search which makes use of ray tracing. [10]

The sonel mapping technique provides an interesting opportunity to be accelerated. After the sonel map has been constructed it can be stored and later used for the gathering step. This could be a useful for acoustic engineers. They might want to run the same simulation multiple times with minor changes to the parameters. In some of these cases, the constructed sonel map could be re-used which would decrease the need computation time to perform the simulation. By accelerating the algorithm with GPUs an even further performance benefit could be achieved.

Furthermore, if the gathering step can be accelerated enough by making use of hardware accelerated ray tracing, it would be possible to use acoustic simulations in interactive applications.

One such use case could be video games. A lot of sound sources in video games are static or scripted and as such, the sonel map of these items could be constructed beforehand. The sonel maps can then be loaded into memory during loading screens or dynamically, before encountering the sound source. With this sonel map, realistically simulated acoustics could be provided to the player. This would enhance the realism of the game and the immersion of the player.

If the gathering step can be performed in the same time as one frame is rendered, the acoustics simulation could be updated to the players location and direction for each frame. However, this means that

gathering step needs to be calculated within 33ms or less, assuming a minimum frame-rate of 30 frames per second.

For the implementation of this thesis the ray tracing framework OptiX was chosen. The choice was made because I had prior experience with CUDA and the OptiX framework is written on top of CUDA. The framework makes it easy to utilise the capacities of the GPU in the context of ray tracing.

Chapter 2

Background

2.1 Computer Graphics

One of the goals of computer graphics research has been to create photo-realistic images with computers. This means using a computer to synthesize a picture that is indistinguishable from reality. Many techniques and algorithms have been developed to accomplish this goal and we can divide them into two branches: rasterization and ray tracing.

The ray tracing branch of computer graphics uses the physics of light particles to construct a 2D image based on a 3D scene. The idea is to follow the path that light particles take as they bounce around the environment, starting from a light source. It is similar to how our eyes see things or how cameras take pictures. The light particles start their journey from the light source, they bounce around the environment and some particles eventually make their way to an observer (an eye or camera).

Whilst the idea of ray tracing follows the physics of how light works, in some techniques such as path tracing, the way we trace the light particles is actually reversed because only very few of the emitted light particles actually end up at an observer. Most light particles just bounce around and are never observed by anyone or anything. This is no problem in the real world but for a computer it is a waste of computing power to simulate light particles that do not contribute to creating the final image. For this reason path tracing starts the light particle trajectory at the observer and then tries to find a way to a light source instead.

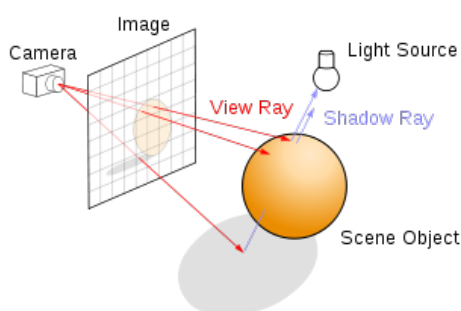


Figure 2.1: A sphere being drawn on an image plane using ray tracing. The image was taken from Wikipedia.

Even though ray tracing was the best known technique for creating photo-realistic images the hardware at the time was not capable enough to produce high quality images in a feasible time. Because of this, rasterization was the go-to technique to render images on a screen. Instead of relying on light physics, the rasterization technique determines the order in which 3D items should be rendered on the screen and then uses the vertices of a 3D model to project the geometry (often a triangle) onto the image raster. The material of the model is then used to color in the pixels that were drawn. Because there is no simulation of light a standard rasterized image does not look photo-realistic. Therefore a lot of techniques have been developed as an addition to rasterization in order for it to look more photo-realistic.

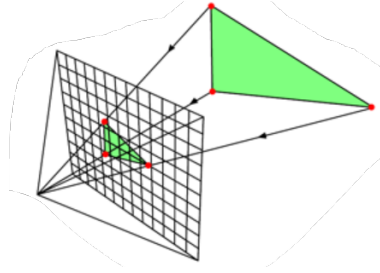


Figure 2.2: A triangle being projected on an image plane using rasterization. Image taken from scratchapixel.

Today, ray tracing is widely used in all applications that need to create a photo-realistic image and rasterization is mainly used in applications where images need to be constructed at interactive frame-rates of 30-60 frames per second such as video-games. Recent strides in graphics hardware have made it possible to use a combination of both techniques in order to increase the fidelity of the images on screen however, ray tracing is not yet fast enough to entirely replace rasterization.

Ray-tracing can also be used outside of the world of computer graphics, Therefore we will dive in deeper on two ray tracing techniques to discuss how they work so we may subsequently dive into their use in acoustic simulation later.

2.1.1 Path Tracing

Path-tracing is the name of a specific ray tracing technique. It was designed as an improvement to regular ray tracing. In early ray tracing algorithms, when a ray intersected with an object in the scene, only light that was directly coming from the light sources was considered. An essential problem with these algorithms was that they did not support indirect lighting. To create a photo-realistic images it is important to both support direct and indirect lighting which is called **global illumination**.

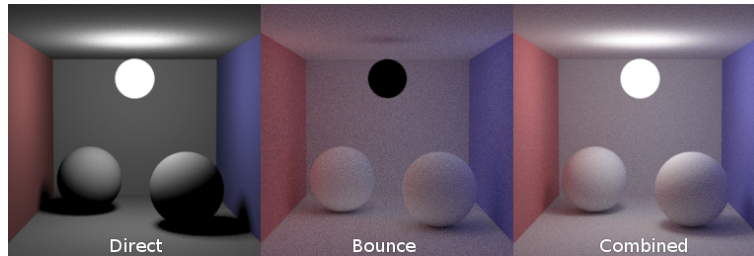


Figure 2.3: A scene rendered with direct, indirect and combined lighting. Image taken from smerity.com

The key principle of global illumination is that every object in a scene contributes to the lighting of said scene. An object within the scene will reflect and scatter all light particles that hit it. The amount of reflection and scattering will depend on the material properties of the object. Some objects are very good reflectors and will primarily reflect light, whilst other objects are very diffuse and will primarily scatter it instead. Most objects are a combination of both. The reflected light then goes on its way and it will encounter a different object or it is observed by the observer.

As previously mentioned, it is common for ray tracing algorithms to start the process from the observer. This is the case for path tracing as well. It differs from early ray tracing algorithm by tracing the ray recursively through the scene. The ray starts off at the observer, after which it will encounter an object. From here, a new ray will be traced in a randomized direction taking into consideration the angle of the incoming ray as well as the material properties of the object. This process then repeats until the ray no longer encounters objects (a ray miss) or until a pre-configured maximum amount of bounces has been reached (also called maximum depth). In path tracing, lights are no longer handled as separate entities and are instead added as regular objects in the scene that emit light.

In other words, when a ray hits an object in path tracing, it needs to know the amount of light that is going from the hit-location towards the origin of the ray. This is called the outgoing light. The outgoing

light is a combination of the light that is emitted by the object itself and the amount of light that is indirectly hitting that spot and is subsequently reflected in the direction of the incoming ray. The formula that this process provides a solution for is known as **the rendering equation**.

2.1.2 The Rendering Equation

The rendering equation was introduced by David Immel et al. and James Kajiya.[1] It is an integral equation that describes the outgoing radiance of a point on a surface as a sum of the emitted radiance and reflected radiance. The formulaic notation of this equation is:

$$L_o(\mathbf{x}, \omega_o, \lambda, t) = L_e(\mathbf{x}, \omega_o, \lambda, t) + \int_{\Omega} f_r(\mathbf{x}, \omega_i, \omega_o, \lambda, t) L_i(\mathbf{x}, \omega_i, \lambda, t) (\omega_i \cdot \mathbf{n}) d\omega_i \quad (2.1)$$

$L_o(\mathbf{x}, \omega_o, \lambda, t)$ is the outgoing spectral radiance going in direction ω_o coming from the position \mathbf{x} at timestamp t . The light has wavelength λ .

$L_e(\mathbf{x}, \omega_o, \lambda, t)$ is the spectral radiance emitted from the position \mathbf{x} in the direction ω_o at timestamp t for wavelength λ .

$f_r(\mathbf{x}, \omega_i, \omega_o, \lambda, t)$ is the bidirectional reflectance distribution function. This function determines how much radiance coming in from direction ω_i , at position \mathbf{x} , is reflected in the direction of ω_o . The function also takes into account the wavelength λ and the timestamp t .

$L_i(\mathbf{x}, \omega_i, \lambda, t)$ is the incoming spectral radiance coming from the direction ω_i towards position \mathbf{x} at timestamp t for wavelength λ .

$(\omega_i \cdot \mathbf{n})$ is a scaling factor. It scales the contribution of the incoming radiance depending on the angle between the surface normal \mathbf{n} and the direction of the incoming radiance ω_i . When the angle between the surface normal and the incoming direction is large, the radiance will be spread out over a larger surface area and therefore, the reflected radiance will be less than when the angle is small.

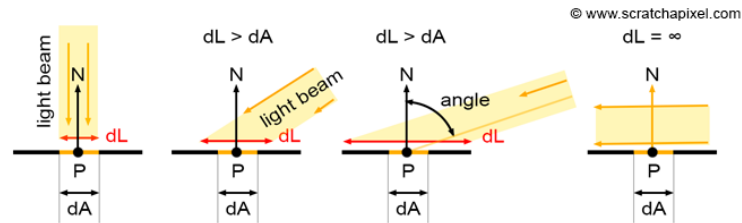


Figure 2.4: The scaling factor visualized. Image taken from scratchapixel.

The integral $\int_{\Omega} d\omega_i$ represents that we are taking the sum of all radiance coming from every direction in the hemisphere Ω , which is aligned with the surface normal of \mathbf{n} at position \mathbf{x} .

Sampling Unit Hemisphere

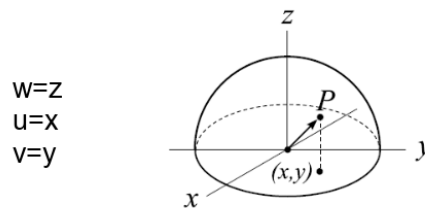


Figure 2.5: The scaling factor visualized. Image taken from github.

Most ray tracing algorithms attempt to solve this equation by using Monte Carlo. The technique explained in the previous section, path tracing, is one such algorithm.

In path tracing, the ray tracing starts at a pixel that we want to determine the color of. The ray then travels through the scene. It subsequently hits an object. The value of the pixel will then be determined by the light that is emitted by the object at the hit-location of the ray as well as the indirect lighting reflecting from this position to the pixel. By selecting a random direction in the hemisphere aligned with the normal of the surface at the hit-position and recursively applying ray tracing, samples will be gathered to estimate the value of the integral part of the rendering equation. Combined with the emitted light, which is a property of the object itself, we have an estimate solution for the rendering equation. By increasing the amount of rays through each pixel the estimate will become more accurate and provide better results.

2.1.3 Photon-Mapping

Photon-mapping is another ray tracing technique that attempts to solve the rendering equation by using Monte Carlo integration. Unlike path tracing, it is a biased algorithm and because of this it does not converge to the correct solution. However, by increasing the number of samples, the result will get closer to the correct solution of the rendering equation.

Different from path tracing, photon-mapping starts tracing rays from the light sources and not from the observer. While it is more computationally efficient to start the ray tracing process from the side of the observer it does make the calculation of certain light paths more difficult. One such light paths is what is called caustics. Caustics happen when light is refracted by transparent objects like glass in such a way that the light is focused and will appear as bright patterns on the surface of the next object.

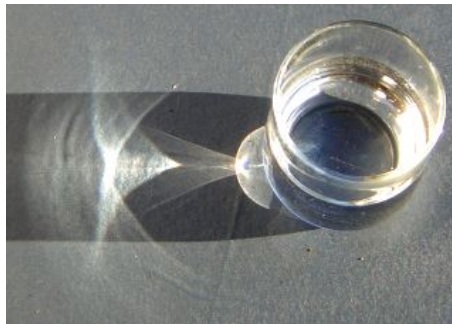


Figure 2.6: Caustic caused by a glass of water. Image taken from Wikipedia.

The key characteristic of photon-mapping is that the algorithm does not immediately use the simulation of light particles to calculate the value of the pixels on the screen. The first step in the algorithm ray-traces light particles from light sources in the scene until they reach the maximum bounce depth or until they exit the scene. Each hit (or bounce) will be stored in the photon-map. The photon-map is a data structure where particle hits are stored alongside their energy, wavelength, position, incident angle, etc. This step of the algorithm is also known as the **photon generation** step.

After the photon-map has been constructed, a second ray tracing step will be performed this time starting from the observer. The ray traversal is similar to path tracing but the calculation of the outgoing spectral radiance is where it differs. To get an estimate value for the outgoing spectral radiance, the photon-map is consulted and the value is estimated based on the nearby photon hits and their properties. This step is called the **photon gathering** step.

2.2 Room Acoustic Simulation

Room acoustics is a branch within acoustics that looks at how sound waves behave within an enclosed space. It takes a look at how each sound frequency behaves in relation to the properties of the enclosed space. This not only means the layout of the space but also what materials the space is composed of and how these materials affect the sound propagation.

The study of these properties are important for multiple fields. These fields include sound proofing, sound staging, performance halls, etc. Being able to measure these properties by means of simulations aids in the design of these spaces. In the world of room acoustic simulation there are two main branches: Wave-solvers and geometric acoustics.

2.2.1 Wave Solvers

Wave-solvers are pieces of software that simulate the propagation of sound waves by solving the wave equation, which is also known as the Helmholtz-Kirchoff equation. This method of simulation provides high levels of accuracy at the cost of being very computationally expensive. Historically this method of simulating acoustics was considered unpractical and unsuitable for anything but the simplest of geometries.

2.2.2 Geometric Acoustics

Instead of solving the wave equation like wave-solvers, the field of geometric acoustics aims to simulate the acoustics of a room by modeling sound waves in a simpler way. This was done because wave-solvers were considered unusable and it is still the main technique in common acoustic software found today.

A variety of techniques have been developed within this branch of acoustics and the most well-known techniques are ray-tracing, the image-source method, beam-tracing and sonel-mapping.

2.3 Stochastic Sonel Mapping

Sonel mapping is a technique developed by Bill Kapralos, Michael Jenkin and Evangelos Milios.[4] It is a Monte-Carlo simulation of room acoustics based on the similarly named technique photon mapping. Like photon mapping, sonel mapping is a two-pass algorithm whereby firstly a map of sonels (sound particles) is constructed and secondly the sonel map is then used to construct an echogram which can be transformed into a room impulse response by using some audio processing. These two steps are respectively called the **sonel tracing** step and the **sonel gathering** step.

2.3.1 Sonel Tracing

In the sonel tracing step sonels are emitted from one or more sound sources within a 3-dimensional scene. Each sonel can be viewed as a bundle of information and contains the following data: energy, frequency, direction, position and distance traveled. It is important to keep track of distance that the sonel has traveled as this will be used to calculate at which time the receiver will hear the sound carried by the respective sonel.

In reality sound is composed of a whole range of frequencies. To reduce computational complexity, the simulation has a fixed number of pre-configured frequencies and the amount of sonels emitted per discrete frequency can be configured individually. A sound source produces sound with a certain amount of energy, often expressed in decibels (dB). The amount of energy per sonel depends on the energy with which the sound source produces sound. Typically, the energy of a sound source is expressed in decibels so it is required to transform this decibel value back into energy. This can be done by inverting the formula to calculate the decibel amount and then dividing the calculated energy by the total amount of sonels that will be emitted.

$$E_{sonel} = \frac{10^{L/10}}{N_{sonel}} * 10^{-12} W/m^2 \quad (2.2)$$

- E_{sonel} Is the energy of one sonel in W/m^2 .
- L Is the energy of the sound source expressed in decibels (dB).
- N_{sonel} Is the amount of sonels emitted from the sound source.
- $10^{-12} W/m^2$ Is the smallest amount of energy that is audible for humans.

The amount of sonels that are emitted from a sound source can be freely chosen, with a higher number increasing the accuracy of the simulation and giving a better estimate for the echogram calculated in the sonel gathering step. There is a minimum amount of sonels required which depends on the size of the sound receiver.

$$N_{min} = \frac{4(v_s t_{max})^2}{r_k^2} \quad (2.3)$$

- N_{min} Is the minimum amount of sonels required.
- v_s Is the speed of sound in air: $343 \frac{m}{s}$.
- t_{max} Is the maximum duration of the simulation in seconds.
- r_k Is the radius of the receiver sphere in meters.

The direction of each sonel is randomized and the distribution depends on the characteristics of the sound source that is being simulated. A common and easy approach is to simulate a sound source as emitting sound in all directions. In this case, the direction of each sonel is a uniformly random one from a sphere around the sound source location.

After a sonel has been emitted we can trace its path through the scene by using ray tracing. When the ray intersects with the surface of an object within the scene we say that the sonel has hit a surface at the location x_{sonel} . After a sonel has hit a surface it can be either reflected diffusely or specularly, or it can be absorbed by the surface in which case the ray tracing process will be terminated for this sonel. Which action to take is determined in a stochastic manner. Each surface in the scene has its probabilities of reflection with δ being the probability of a diffuse reflection, s being the probability of a specular

reflection and $1 - (\delta + s)$ being the probability of absorption, with $(\delta + s) \in [0, 1[$. When a sonel hits a surface, a uniformly distributed number ξ is generated with $\xi \in [0, 1]$ and the action is determined using the following method:

- $\xi \in [0, \delta]$: diffuse
- $\xi \in]\delta, (\delta + s)]$: specular
- $\xi \in](\delta + s), 1]$: absorption

If the sonel is not absorbed after encountering a surface it will be reflected. This reflection can be either diffuse or specular. A specular reflection will bounce the sonel in a manner similar to a mirror where the angle of incidence is the same as the angle of reflection. A diffuse reflection is more complex. Instead of reflecting in a singular direction, a diffuse reflection will scatter the incoming energy in all directions, see image 2.7. To simulate the scattering effect of diffuse reflections a random direction in the unit hemisphere around the surface normal will be chosen and the sonel will continue its journey through the scene.

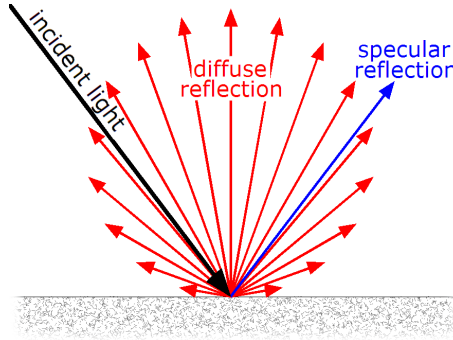


Figure 2.7: A comparison between specular and diffuse reflections.

1

Before a sonel is reflected we have to update its distance-traveled and position properties. The distance between its previous position and its current hit location x_{sonel} is added to its distance-traveled property. The position of the sonel is then updated to match the hit location. If the sonel will be reflected diffusely and after both of these properties have been updated, a copy of the sonel will be stored in the so-called **sonel map**. The sonel map is a collection of all the times that a sonel has hit a surface within the scene. After storing the sonel in the sonel map its direction will be updated and the ray tracing can continue. This process will keep on repeating recursively until the sonel is absorbed or until the ray misses all objects in the scene.

In summary, in the sonel tracing step each sound source emits a chosen amount of sonels. These sonels will be emitted in a randomized direction. The path of each sonel will be simulated by using ray tracing recursively. When a ray intersects with the scene, the sonel has hit some surface and will be subsequently absorbed or reflected. If the sonel is absorbed, or if the ray has missed every object in the scene (a so-called ray-miss), the ray tracing is finished. If the sonel is reflected it can either be a specular reflection or a diffuse reflection. If the reflection is diffuse the sonel-hit will be recorded in the sonel map otherwise it will not.

After each sonel has finished its ray tracing the sonel map is complete and can then be used in the subsequent sonel gathering step.

2.3.2 Sonel Gathering

Like the photon gathering step in photon mapping the sonel gathering step uses the previously constructed sonel map to estimate the sound energy levels at the location of a given observer. In the case of sonel mapping, the observer is a sound receiver and it can be modeled as a sphere which receives sound from every direction. To calculate what the sound receiver is able to hear we make use of ray tracing.

A number of visibility rays are traced from the sound receiver in randomized directions. The visibility rays have to be cast for each discrete frequency as each ray will only look for sonels of the same frequency. Similar to the ray tracing done in the sonel tracing step, the rays in the sonel gathering step can be

absorbed or reflected specularly or diffusely. However, unlike the previous step, the ray tracing process also ends when a diffuse reflection is made. A specular reflection can be seen as a continuation of the ray tracing process. The incoming ray will be reflected in a mirror-like manner where the outgoing angle is equal to the angle of incidence. When a diffusely-reflected ray encounters a surface in the scene, the sonels within the sonel map are used to estimate the sound energy levels at the receiver. This is done by consulting the sonel map to retrieve the sonels that are nearest to the location x_{vis} where the visibility ray has hit the surface. The process to determine which action to take when a surface is encountered is the same as when a sonel encounters a surface, see 2.3.1 for more information.

During the ray tracing process it is important to keep track of the distance that has been traveled by each visibility ray as it will later be added to the distance that has been traveled by the encountered sonels to calculate at which time the sound carried by the sonel will be heard by the receiver. When a visibility ray intersects with a surface in the scene the sonel map will be consulted to find the nearest sonels within a configured radius r_{lookup} . The radius r_{lookup} can be freely chosen and smaller values will produce a better estimate as long as there are sufficient sonels within this radius. The process of finding the nearest items within a certain radius is called the **nearest-neighbour search**.

The goal of the gathering step is to construct what is called an **echogram** which is a table of sound energy levels per frequency during a given time interval. Because the timestamp of each sonel is a real number we are required to divide this interval into discrete buckets. Each bucket can be seen as a step in the simulation which has a duration of t_{res} , which is also known as the time-resolution. A resolution of $5 - 10ms$ is advised. With t_{max} being the maximum length of the simulation, the number of required buckets can be calculated with the following formula.

$$N_{buckets} = \frac{t_{max}}{t_{res}} \quad (2.4)$$

In order to add a sonel to the corresponding bucket we need to calculate the timestamp t_{sonel} at which the sonel can be heard at the receiver. This is done by calculating the total distance traveled by the sonel l_{total} which is equal to the distance traveled by the sonel l_{sonel} plus the distance traveled by the visibility ray l_{vis} .

$$l_{total} = l_{sonel} + l_{vis} \quad (2.5)$$

After the total distance has been calculated it can be used to calculate the timestamp t_{sonel} . This can be done by taking the total distance traveled l_{total} and dividing it by the speed of sound v_s which is $343 \frac{m}{s}$ in air.

$$t_{sonel} = \frac{l_{sonel}}{v_s} \quad (2.6)$$

Now that the t_{sonel} is known we can use it to determine which bucket the sonel should be added to in the echogram. This can be done by taking t_{sonel} and dividing it by previously mentioned time-resolution t_{res} . The resulting number is then floored to change it to an integer. The floor function is used because it will correctly assign the bucket so that a sonel with timestamp t_{sonel} is assigned to the bucket b_{fi} where $i * t_{res} \leq t_{sonel} < (i + 1) * t_{res}$. With b_{fi} being the i th bucket in the echogram of frequency f .

$$i = \lfloor \frac{t_{sonel}}{t_{res}} \rfloor \quad (2.7)$$

This bucket calculation is used to determine which bucket energy needs to be added to when a visibility ray is reflected diffusely at a certain location x_{vis} in the scene. When such a reflection occurs, the sonel map is consulted to estimate the sound energy coming from x_{vis} and going towards the observer. The N_{sonel} nearest sonels to x_{vis} are retrieved from the sonel map and their energy is scaled to account for air attenuation using the formula:

$$E_r = E_o e^{-ml} \quad (2.8)$$

- E_r is the attenuated sonel energy.
- E_o is the original sonel energy.
- m is the air absorption coefficient.
- l is distance traveled through the air.

The air absorption coefficient m is a function of temperature, frequency, humidity and atmospheric pressure. Commonly temperature, humidity and atmospheric pressure are pre-configured and do not change during the simulation which means the air absorption coefficient can be simplified as a function of only frequency. Values for m can be found in the work of Bass *et al.*[2]

l Is the combined distance traveled by the sonel and the visibility ray which has previously been mentioned as l_{total} . After the sound energy of each sonel has been attenuated the assumption is made that each sonel will contribute to the same echogram bucket and therefor the sonel energy is further scaled by dividing it with N_{sonel} . If the amount of sonels within a radius of r around x_{vis} is less than N_{sonel} the sonel energy is divided by the actual amount of found sonels N_{found} instead. After the energy of each sonel has been scaled it can be added to its corresponding echogram bucket b_{fi} using equation 2.7.

Chapter 3

Implementation

The focus of this thesis is to create a Sonel-Mapping implementation that utilizes the power of GPUs to speed up both the generation and gathering steps. The implementation is based off of the sonel-mapping technique (see section 2.3) by Kapralos et al.

Sonel-mapping, like photon-mapping, is made up of 2 stages. A sonel tracing stage followed by the acoustic rendering stage.[4]

- **Sonel tracing stage:** Emits sonels from sound sources which are then ray-traced throughout the scene. Each collision with the environment is saved and used to construct the sonel map for the following stage.
- **Acoustic rendering stage:** Sends out visibility rays from the listener positions and combined with the previously constructed sonel map calculates an estimation of an echogram.

3.1 Scene Setup

3.1.1 3D Environment

In order to simulate the propagation of sonels a description of a 3-dimensional environment is needed. To this end my implementation uses an OBJ-loader to load an OBJ-file into memory.

3.1.2 Sound Sources

To be able to emit sonels in the sonel tracing stage we need to have a description of the sound sources in the room. In my implementation, sound sources are defined in-code by creating a *SoundSource* object. This *SoundSource* object is defined as being composed out of a position and a list of *SoundFrequency* objects. A *SoundSource* is a simple container of the frequencies that it is composed out of. Because of this, the bulk of information about the sound produced by this source is defined in the *SoundFrequency* objects. For a C++ description of the *SoundSource* class, see listing 3.1

Each *SoundFrequency* object is made up out of a frequency, the amount of sonels to emit and a list of decibel values. This list of decibel values is meant to represent the energy at which this frequency emits sonels at a given timestamp. Each index of this list corresponds to a single time-step, exactly like the echogram explained in section 2.3.2. In other words, given the i th decibel value L_i in this list, the time at which these sonels will be emitted corresponds to $t_{emit} = i \times t_{res}$. Going forward, the values in the decibels-array will be called **decibel-steps**. The C++ code for *SoundFrequency* can be found in listing 3.2.

Listing 3.1: C++ code of the SoundSource class.

```

class SoundSource {
public:
    vec3f position;           // Vector of 3 floats

    unsigned int frequencySize; // The number of different frequency values.
    SoundFrequency* frequencies;
}

```

Listing 3.2: C++ code of the SoundFrequency class.

```

class SoundFrequency {
public:
    unsigned int frequency;
    unsigned int sonelAmount;

    // The number of times sound is emitted at certain decibel levels.
    unsigned int decibelSize;
    float* decibels;
}

```

3.1.3 OptiX Setup

OptiX is a ray-tracing framework developed by Nvidia which is built on top of CUDA. CUDA is a technology, also developed by Nvidia, to allow developers to write software that directly runs on GPUs. OptiX provides an API which is able to make efficient use of a GPU to accelerate ray-tracing algorithm. Acceleration structures and ray-traversal algorithms are implemented by the framework itself and the developer is free to make use of these items to build a ray-traced application on top of it. It is out of the scope of this thesis to teach the fundamentals of OptiX but it is useful to explain how the framework is setup in my implementation.

To start the ray-tracing process, OptiX has to be provided a so-called **traversable**. Traversables can be seen as graphs detailing how the scene can be traversed so that OptiX can determine whether or not a ray intersects with any of the objects within the scene. There are multiple types of traversables that can be used within the OptiX API but the ones relevant to my implementation are the **geometry acceleration structure** and the **instance acceleration structure**.

As the name "acceleration structure" implies, these traversables are created in a way so that the ray-tracing process can be accelerated. Typically these acceleration structures are based on the bounding volume hierarchy model but different implementations are implemented as well depending on the hardware it is executed on.

Geometry acceleration structures are the most commonly used type of acceleration structure. This type of acceleration structure aims to accelerate the ray-tracing process for geometric data within the scene. The OptiX API has built-in support for triangles, curves, spheres as well as for custom primitives. In my implementation a triangle acceleration structure is used to accelerate the traversal of the environment as the input data is a collection of 3D-meshes built out of triangles. Next to this, 2 custom primitive acceleration structures are used. One primitive acceleration structure is used to represent the sonels within the sonel map and the other acceleration structure represents the sound sources.

The instance acceleration structure is composed out of the previously mentioned geometry acceleration structure and it represents the entry point for the OptiX API to traverse the scene for its ray-tracing process.

Custom primitive acceleration structures are required to have an axis-aligned bounding-box (or AABB) to represent each object within them. For the sonel acceleration structure this is done by taking the position x_{sonel} as centre point of a cube with sides of length $2r_{lookup}$, which is the radius of the nearest neighbour search during the sonel gathering step. This acceleration structure is later used to perform the nearest neighbour search during the sonel gathering step. Using ray-tracing for accelerated nearest

neighbour search is a technique developed by Yuhao Zhu.[11] Sound sources have been implemented in a similar manner. Sound sources are defined as spheres in my implementation so the bounding box has sides equal to the diameter of the sphere.

Each separate acceleration structure has its own visibility mask. During the ray-tracing process, rays can be configured with their own visibility mask to select which acceleration structures will be used during the ray-traversal process. In my implementation it is both used to ensure the correct execution of the algorithm as well as optimisation. For instance, the first step in the sonel gathering process is to ray-trace visibility rays to see where they intersect with the environment. These visibility rays can be configured to ignore the sonel acceleration structure which could otherwise give faulty results (as sonels are not part of the environment) but also speeds up the ray-traversal process.

3.2 Sonel Tracing

The first step in the sonel tracing stage is to emit and trace the sonels. Sonels are emitted from the pre-configured *SoundSources* and their corresponding *SoundFrequencies*. Each *SoundFrequency* may itself have multiple decibel values which requires multiple emissions of sonels. In order for us to simulate everything we have to loop over all decibel values of all *SoundFrequencies* of all *SoundSources*. The pseudo-code of this step is:

Listing 3.3: Pseudo-code for the high-level sonel emission loop.

```
foreach soundSource in soundSources
    foreach soundFrequency in soundSource.frequencies
        timestep = 0

        foreach decibel in soundFrequency.decibels
            emitSonels(soundFrequency.sonelAmount, decibel, timestep * tRes)
            timestep++
```

Because this algorithm is being implemented on the GPU a choice has to be made to decide about which loops will be executed on the GPU and which will be executed on the CPU. In my implementation we have opted to only execute the inner-most loop on the GPU itself. This was done to reduce the amount of memory needed on the GPU to store the sonel hits during the ray-tracing process. Because dynamic memory allocation negatively impacts performance on GPUs it is best to pre-allocate the buffers in which the sonel-hits will be stored. The maximum size of the simulation is equal to:

$$N_{buffer_{max}} = N_{bytes} N_{depth} \sum_{i=0}^{N_{sources}} \sum_{j=0}^{N_{freq_i}} N_{sonel_{ij}} N_{decibels_{ij}} \quad (3.1)$$

- N_{bytes} is the amount of bytes required to store one sonel hit.
- N_{depth} is the maximum amount of allowed reflections/bounces for a single sonel.
- $N_{sources}$ is the total amount of configured *SoundSource* objects.
- N_{freq_i} is the amount of configured *SoundFrequency* objects for *SoundSource* i .
- $N_{sonel_{ij}}$ is the amount of sonels to emit for *SoundFrequency* j of *SoundSource* i .
- $N_{decibels_{ij}}$ is the amount of times that *SoundFrequency* j of *SoundSource* i will emit sound at a certain decibel level.

Even though it is computationally advantageous to move more loops of listing 3.3 to the GPU it will significantly increase the memory required to run a simulation. If each simulation is only required to simulate one *SoundFrequency* the required amount of memory on the GPU is significantly reduced as it removes the summations of equation 3.1 and can thus be simplified to the following:

$$N_{buffer_{gpu}} = N_{bytes} N_{depth} N_{sonel_{ij}} N_{decibels_{ij}} \quad (3.2)$$

After each simulation the data can be copied over from the GPU memory to the CPU memory and the next simulation can begin. Because memory operations aren't as costly when working on the CPU the required memory of the entire simulation will be significantly less than calculated in equation 3.1. This is because the pre-allocated buffer has to make the assumption that each sonel will reach the maximum reflection depth N_{depth} . In reality however, this is not the case because of the russian-roulette approach used during the sonel tracing stage as explained in subsection 2.3.1. Given a probability of α_{abs} which represents the probability of a sonel being absorbed by a surface within the scene. We can calculate the estimated memory requirement for the CPU by substituting N_{depth} in equation 3.1 by the estimated amount of bounces $E(N_{bounces})$.

$$E(N_{bounces}) = \sum_{n=1}^{N_{depth}} n[(1 - \alpha_{abs})^{n-1} - (1 - \alpha_{abs})^n] \Rightarrow \quad (3.3)$$

$$E(N_{buffer_{total}}) = N_{bytes} E(N_{bounces}) \sum_{i=0}^{N_{sources}} \sum_{j=0}^{N_{freq_i}} N_{sonel_{ij}} N_{decibels_{ij}} \quad (3.4)$$

In my implementation N_{depth} was freely chosen, in future implementations it can be advantageous to explore the idea of it being calculated based on the estimated amount of bounces $E(N_{bounces})$ during the simulation. But this idea will be further explored in section ??.

Before the ray-tracing process can be started a sonel-buffer of size $N_{buffer_{gpu}}$ is allocated on the GPU to store the sonel hits. The sonel-buffer represents a three-dimensional array of *Sonel* objects but is implemented as a one-dimensional array to increase performance as multi-dimensional arrays requires multiple allocation calls which are costly on GPUs. When the buffer is viewed as a three-dimensional array, each page can be viewed as the sonel-buffer for each value in the decibels array of a *SoundFrequency*. In each page, each row can be seen as an individual buffer of size N_{depth} which stores all sonel-hits of a singular sonel as it is ray-traced through the scene. This visualisation can be seen in image 3.1. A sonel is composed of a **position**, **incidence**, **energy**, **distance**, **time**, **frequency** and **frequencyIndex**. For a C++ description of the *Sonel* class, see listing 3.4.

Listing 3.4: C++ description of the sonel class.

```
class Sonel {
public:
    vec3f position;
    vec3f incidence;

    float energy;
    float time;
    float distance;

    uint32_t frequency;
    uint16_t frequencyIndex;
};
```

After the creation of the sonel-buffer, the data that is required on the GPU side is copied from the CPU to the GPU. In my implementation this includes the *SoundFrequency* that is currently being simulated, a pointer to the allocated sonel-buffer and general parameters like the speed of sound, the position of the sound source and the time resolution t_{res} . The frequency-index is not relevant right now and is only needed during the construction of the echogram in the sonel gathering step. In my implementation, the list of frequencies is discrete and therefore each frequency can be given a unique index from 0 until N_{freq} . The frequency-index will be stored as part of the sonel-hit during the ray-tracing process.

The ray-tracing process is composed out of multiple steps. First in the process is the so-called ray-generation step. In this step rays are generated and initiated. Subsequently the ray-tracing step is started upon the generated ray. Once the ray-tracing step is started the process can branch off in two directions. The ray can either fail to find an intersection with the scene (a ray miss) or it can encounter a hit (a ray hit). If a ray miss occurs it will cause the ray-tracing process to halt for this single ray. If a

ray hit occurs we will store the sonel information within the sonel-buffer and decide which further action to take in a probabilistic manner as explained in section 2.3.1.

In order to initiate the ray-tracing process in OptiX a **pipeline** is required. An OptiX pipeline is a description of **programs** which handle various aspects of execution during the ray-tracing process. In other APIs these programs are often called **shaders**. The main programs that can be implemented in a pipeline are the following:

- **Ray generation program:** This program is the starting point of the ray-tracing process. It is the program that is executed first and it is called multiple times in parallel. A ray-tracing pipeline is launched with configurable dimensions. The ray generation program is executed for each value within these dimensions.
- **Intersection program:** The intersection program is executed when the ray-tracing process needs to inquire whether or not a ray has intersected with a primitive. OptiX has built-in intersection programs for certain types of primitives such as triangles, spheres and curves. For custom primitive types it has to be implemented in the pipeline.
- **Any-hit program:** Any-hit programs are executed when a ray encounters an intersection that may or may not be the closest intersection from the ray origin.
- **Closest-hit program:** The closest-hit program is called upon when the ray-tracing process has found the closest intersection to the ray origin.
- **Miss program:** This program is executed when the ray has failed to encounter an intersection. It can occur when the ray is traveling in a direction that does not intersect with any object in the scene or when the maximum ray distance is too short to encounter an object.

For completeness, there are also **exception**, **direct callable** and **continuation callable** programs but these go beyond the scope of the thesis.

As mentioned in the description of the ray generation program the ray-tracing process is initiated with certain dimensions. The common use case for these dimensions in graphics ray-tracing is to set them to the resolution of the screen or window. This will initiate a ray generation program for each pixel that needs to be ray-traced in parallel. In my implementation the x-dimension is set to $N_{sonel_{ij}}$ and the y-dimension is set to $N_{decibels_{ij}}$. With these dimensions, each sonel in each decibel-step will execute its own ray generation program. One can opt to loop over all decibel-steps within the ray generation program itself but by making use of these dimensions we can count on the scheduling of OptiX to optimise the parallel traversal of rays. The launching and scheduling of ray generation programs is handled behind the scenes by OptiX itself.

Each ray in the ray-tracing process can carry with a payload. This payload is in the form of multiple unsigned integers. To be more flexible with the data structure of the payload, my implementation uses these two of these integers to store a 64-bit pointer of a ray data object that is allocated on the stack space of the ray generation program. In my implementation, the ray data represents a sonel and includes the following fields:

- **Ray index (unsigned 64-bit integer):** This field represents a unique integer identifier of the ray.
- **Depth (unsigned 16-bit integer):** This field counts the number of diffuse reflections that have been made by the ray.
- **Distance (float):** The total distance that has been traveled by the ray.
- **Energy (float):** The energy with which the sonel is emitted.
- **Time offset (float):** This field is a result of sound being allowed to be emitted after the start of the simulation. In the original sonel mapping method the timestamp of a sonel is calculated solely based on the distance traveled by the ray. In my implementation, sonels can be emitted at any point during the simulation. As a result the timestamp of when the sonel was emitted has to be added to the timestamp based on the distance. This field stores that offset.
- **CudaRandom:** This field is a self-written wrapper over the CUDA random number generator API.

The ray-index is constructed in such a way that it is also the start index of the sonel-buffer part that was reserved for the respective ray/sonel. The formula for calculating the ray-index is:

$$index_{ray} = N_{depth} * N_{sonel_{ij}} * index_{decibel} + index_{sonel} * N_{depth} + N_{depth_{current}} \quad (3.5)$$

- $index_{ray}$ is the ray-index.
- N_{depth} is the maximum amount of reflections or bounces that a sonel is allowed to make until it is forcefully terminated.
- $N_{sonel_{ij}}$ is the amount of sonels that have to be emitted for the *SoundFrequency* that is currently being simulated. Which is *SoundFrequency* j of *SoundSource* i .
- $index_{decibel}$ is the index of the decibel level with which this sonel/ray was emitted. Each increment of this index corresponds with an increment of t_{res} seconds within the simulation. With t_{res} being the temporal resolution of the simulation.
- $index_{sonel}$ is the index of the ray that was emitted. If a *SoundFrequency* is configured to emit N_{sonel} sonels then the value will be in the range $[0, N_{sonel}[$. This index is not unique as it will be repeated once for each decibel level or in other words $N_{decibel}$ amount of times.
- $N_{depth_{current}}$ is the amount of reflections which the sonel has already made.

The C++ description of the ray data can be found in listing 3.5.

Listing 3.5: C++ description of the ray data class used during the sonel tracing stage.

```
class TracingRayData {
public:
    CudaRandom random;

    uint64_t index;
    uint16_t depth;

    float distance;
    float energy;
    float timeOffset;
}
```

Breaking down the construction of the ray-index formula, we start at the end of the formula. If there was only one sonel to simulate in the entire simulation, the sonel can make a maximum of $N_{depth} - 1$ reflections and thus N_{depth} sonel-hits would have to be stored. In this case, a buffer of the size N_{depth} would suffice and the ray-index would be equal to $N_{depth_{current}}$, the amount of reflections that have already been made. This is the last term of equation 3.5 and it represents the index within the buffer dedicated to the ray of size N_{depth} .

Continuing with this breakdown, the simulation is extended to simulate $N_{sonel_{ij}}$ amount of sonels. Each of these sonels requires a dedicated buffer of size N_{depth} which makes for a buffer with a total size of $N_{depth}N_{sonel_{ij}}$. Each sonel can be given an index in the range of $[0, N_{sonel_{ij}}[$ where each number is used exactly once, this is the sonel-index or $index_{sonel}$. The buffer can be thought of as a table where each row corresponds to a buffer of size N_{depth} . Each sonel can use its sonel-index to find its respective row which is what the second term in equation 3.5 represents. To select the correct column within a row in order to store a sonel-hit, the $N_{depth_{current}}$ value of the sonel can be used as explained in the previous paragraph.

Extending the simulation further to simulate $N_{decibel_{ij}}$ amount of decibel levels, the simulation will now emit $N_{sonel_{ij}}$ for each decibel level. In addition to the sonel-index, each sonel will also be assigned a decibel-index ($index_{decibel}$). Each sonel that is emitted for a specific decibel level will receive the same decibel-index. The simulation now requires a buffer of size $N_{depth}N_{sonel_{ij}}N_{decibel_{ij}}$. The buffer can be thought of as a 3-dimensional table where each page is of size $N_{depth}N_{sonel_{ij}}$ and there are

$N_{decibel_{ij}}$ pages. To select a page, the decibel-index has to be multiplied by the size of each page which is $N_{depth}N_{sonel_{ij}}$. This is what the first term in equation 3.5 represents. After selecting a decibel-page, the previous paragraph explains how to select the specific row and column to store a sonel-hit.

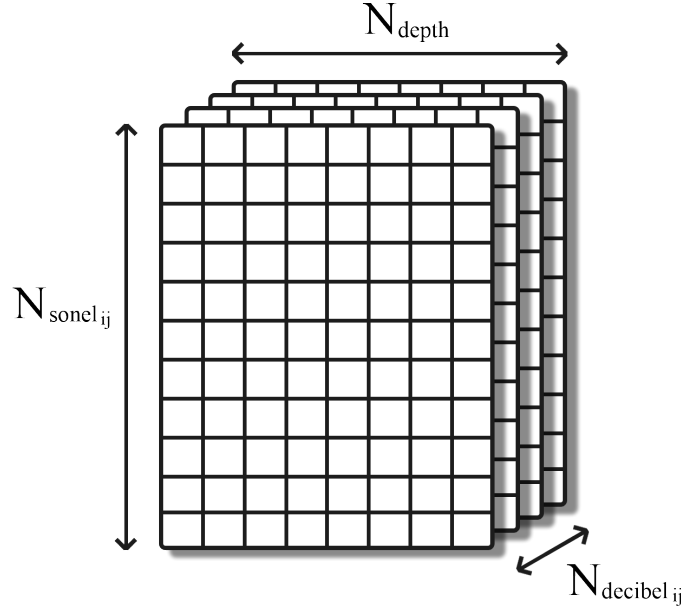


Figure 3.1: The visual representation of the sonel-buffer structure.

In the ray generation program of my implementation, the ray data is initialized by using the values stored in the launch parameters. The ray-index is initialized by employing equation 3.5, the decibel-index and sonel-index are retrieved from the OptiX API. Furthermore, the depth and distance fields are initialized with the value 0. To fill in the energy field, the decibel level is extracted from the *SoundFrequency* and subsequently transformed into sound energy by employing equation 2.2. The sonel amount N_{sonel} can also be found in the *SoundFrequency*.

The *CudaRandom* field needs to be initialized with a seed value. It is important that the seed value is unique for each ray generation program that is executed. Because I made the choice to execute a certain part of the high-level execution loop on the CPU (see listing 3.3), the OptiX pipeline will be initiated multiple times. This means that we can not rely on the ray-index to provide a unique seed value. To remedy this issue a frame-index is introduced. After the execution of a pipeline has finished, the frame-index is incremented with the size of the buffer that was allocated. In the ray generation program, the frame-index can be combined with the ray-index to calculate a unique seed value for *CudaRandom*.

After initializing the ray data, my ray generation program generates a random direction that will be used to initiate the ray traversal process. As previously mentioned, sound sources are modeled as a sphere which means that the sonel can be emitted in any direction. To generate a uniformly distributed random direction, the *CudaRandom* field of the ray is used. The *CudaRandom* instance uses the CUDA API to generate three normally distributed numbers between $[-1, 1]$. Each number represents a component of a point on a unit cube. Afterwards the vector constructed from these points is normalized resulting in a unit vector. The ray traversal process can now be initiated, starting from the position of the sound source in the direction of the vector that has been generated. For the pseudo-code description of the ray generation program see listing 5.1.

After initiating the ray traversal OptiX can execute the intersection, any-hit, closest-hit or miss program. During the sonel tracing stage the any-hit program is not required as we are only interested in the closest hit. Therefore the any-hit program is not implemented or enabled in the sonel tracing OptiX pipeline. Because the model used to represent the environment of the simulation is constructed from triangles, OptiX uses its built-in intersection program.

The closest-hit program is implemented and it is where the implementation constructs the sonel map. When the closest-hit program is executed it means that the ray (or sonel) has intersected with the

geometry of the scene. As explained in section 2.3.1, the next action is decided by chance. A uniformly distributed number ξ between 0 and 1 is randomly generated by using the *CudaRandom* field of the ray data. In my implementation the probabilities δ and s are static and thus the same for every object. With δ and s being the probabilities of a diffuse and specular reflection respectively. A future improvement would be to extract these probabilities from the material properties of the object. If ξ is greater than $\delta + s$, the sonel is absorbed and no new ray traversal will be initiated. In this case there is no sonel-hit to store in the sonel map. However, after the tracing step, the sonel-buffer will be iterated over and if the sonel did not reach a depth of N_{depth} the remaining sonel entries will be uninitialized and thus we are required to mark the end of the buffer with a *null* value. This is done by marking the sonel at the ray-index as having a frequency of 0. Which can not happen during normal execution of the simulation.

When the ray/sonel is not absorbed, the distance field of the ray data is updated by adding the distance that was traveled between the ray origin and the current intersection. The position x_{hit} of the intersection is calculated by using the barycentric coordinates supplied by the OptiX API to interpolate the vertex data of the model. Afterwards the distance can be calculated by calculating the length of the vector between x_{hit} and the origin of the ray x_{ray} . The length of this vector is then added to the distance field of the ray data.

If the ray/sonel is reflected diffusely, the sonel-hit will be stored in the sonel-buffer. The fields of the *Sonel* at the ray-index are initialized with the following values:

- **Position** is set to the hit location x_{hit} .
- **Incidence** is set to the direction of the ray.
- **Energy** is copied over from the ray data.
- **Distance** is copied over from the ray data.
- **Time** is set to be equal to $time = \frac{rayData.distance}{v_s} + rayData.timeOffset$.
- **Frequency** is copied over from the *SoundFrequency*.
- **FrequencyIndex** is copied over from the *SoundFrequency*.

Following a diffuse reflection, the depth and ray-index fields of the ray are incremented. In case the ray/sonel is reflected in a specular manner, no updates need to be done to either the sonel-buffer or ray data. This is because a specular reflection is treated as a continuation of the path of the current ray/sonel. However, in both cases a new direction needs to be provided to continue the ray in the direction of the reflection.

The direction of the new ray depends on the type of reflection. For a specular reflection, the direction of the new ray will be a mirror-like reflection of the direction of the current ray. This means that the angle between the incoming direction and the surface normal θ_{inc} is equal to the angle between the outgoing direction and the surface normal θ_{out} . Given an incoming direction \vec{d}_{inc} and a surface normal \vec{d}_n , the new direction \vec{d}_{out} can be calculated with the following equation:

$$\vec{d}_{out} = 2(\vec{d}_n \cdot \vec{d}_{inc})\vec{d}_n - \vec{d}_{inc} \quad (3.6)$$

With $(\vec{d}_n \cdot \vec{d}_{inc})$ being the dot product between \vec{d}_n and \vec{d}_{inc} . The surface normal is retrieved from the normal-map of the model of the scene. In cases where there is no normal data attached to the model, the normal is calculated by using the vertex data of the intersected triangle.

For diffuse reflections, the new direction will be a uniform distributed randomly generated unit vector within the hemisphere aligned with \vec{d}_n . The implementation of generating such a vector makes use of the *CudaRandom* field included in the ray data. The function for generating a unit vector in a sphere is re-used and the vector is multiplied by -1 if it is outside of the hemisphere. Whether or not a vector is outside of the hemisphere can be calculated by checking if the cosine angle between the generated vector and the surface normal is negative. With this newly generated vector, the ray traversal process is initiated again in similar fashion as in the ray generation program.

To complete the closest-hit program, a check is added to ensure that the ray traversal process is stopped before the maximum sonel-depth has been reached. When the ray traversal process is stopped because

the maximum depth has been reached, a null-sonel is added to the sonel-buffer as if the sonel was absorbed.

The remaining program in the pipeline is a miss program. A ray-miss is treated similar to a sonel being absorbed or reaching the configured maximum depth. This means that on a ray-miss event, a null-sonel is added to the sonel-buffer at the ray-index. Furthermore the ray traversal is ended.

This concludes the implementation of the sonel tracing stage. The CPU can now be instructed to launch the OptiX pipeline and the execution of the CPU function is suspended until the OptiX pipeline completes. After every pipeline completion, the CPU copies the sonel-buffer back to CPU memory and compacts it by removing all null-sonels and uninitialized sonels. The code for this compacting step can be found in listing 3.6. Afterwards, a new sonel-buffer is allocated on the GPU and data for the next *SoundFrequency* is copied over. After repeating this process for every *SoundFrequency* for every *SoundSource* we are left with a list of sonels that can be used to construct the **sonel map**.

Listing 3.6: C++ code of how the sonel-buffer is compacted on the CPU.

```
void compactSonels(Sonel* sonelBuffer, uint64_t sonelBufferSize,
                  uint16_t maxDepth, std::vector<Sonel>& sonelList) {
    uint64_t rowSize = sonelBufferSize / maxDepth;

    for (uint64_t row = 0; row < rowSize; row++) {
        for (uint16_t column = 0; column < maxDepth; column++) {
            Sonel sonel = sonelBuffer[row * maxDepth + column];

            // If the frequency is 0, this is a null-sonel.
            // This means that the current row is done.
            if (sonel.frequency == 0) {
                break;
            }

            sonelList.push_back(sonel);
        }
    }
}
```



Figure 3.2: On the left a traditional nearest neighbour search. On the right a visualization of the ray traced nearest neighbour search.

3.3 Sonel map and Sound sources

During the sonel gathering step, visibility rays will be traced originating from the list of sound receivers. After these rays intersect with a surface on the scene and get diffusely reflected, the sonel map is used to retrieve the sonels within a specified radius. The amount of sonels that are generated during the sonel tracing step can range from thousands to millions depending on the configuration of parameters. If one would search through this list linearly for each diffuse reflection of a visibility ray, it would result in a significant computational cost. Because of this, it is important that the sonel map is constructed in such a way that searching for sonels near a given position is as cheap as possible.

Traditionally, 3-dimensional k-d trees were used as the data structure of choice for photon/sonel maps as their structure allow for a **nearest neighbour search** with a complexity of $O(\log n)$. In my implementation I have opted to use a new nearest neighbour search technique develop by Yuhao Zhu.[11] The new technique transforms the nearest neighbour search problem into a ray tracing problem. In turn, this ray tracing problem is solved by employing hardware-accelerated ray tracing. Because our implementation aims to accelerate the sonel mapping algorithm by using GPUs it is well suited to our problem. The performance advantage of this approach over other nearest neighbour libraries available for GPUs is claimed to be between 2.2 times and 65.0 times speedup.[11]

The nearest neighbour search problem can be visualized as finding the sonels that are within a certain radius r_{vis} of a given position x_{vis} as seen in image 3.2. In this new approach the problem is converted to a ray tracing problem. This can be done by creating a sphere around each sonel with radius r and then finding all spheres that intersect with the ray fired from x_{vis} in the direction of the surface normal. As seen in image 3.2.

As mentioned in section 3.1.3, the sonel map is constructed as a geometry acceleration structure separate from the geometry acceleration structure of the scene geometry. The sonel map is configured to be a "custom primitive" type of geometry acceleration structure. When an geometry acceleration structure is constructed from custom primitives, the input of the acceleration structure has to be a list of axis-aligned bounding boxes (AABB). As a result each sonel in the sonel-buffer has to have a corresponding AABB representing it. This is done by taking the position of the sonel x_{sonel} and then constructing two new positions by subtracting and adding the radius r_{vis} to x_{sonel} . These two new positions will be $x_{aabb_{min}}$ and $x_{aabb_{max}}$ respectively which represent the minimum and maximum corner of the AABB.

In order to create a geometry acceleration structure the OptiX API requires an array of *OptiXBuildInput* objects. The *OptiXBuildInput* object describes a single primitive within the the acceleration structure. It is composed of several fields, the most important ones being the **type** field and the field containing the geometric data of the primitive. The name of this field depends on the aforementioned type field. For the sonel map the type is that of custom primitives and therefore the data field is called **aabbBuffers**. The aabbBuffers field needs to be assigned a pointer value that point towards the AABB data on the GPU. Because multiple allocation calls are costly and slow, an AABB buffer with the size of the sonel list is allocated in one allocation call. The pointer required for the aabbBuffers field is calculated by adding the index of the sonel to the pointer pointing to the start of the buffer.

Data from sonel can not directly be added to the primitive itself during the construction of the geometry acceleration structure. However, it the two can be connected, during the ray tracing process, by making

use of the **shader binding table (SBT)**. The SBT is an array containing information about the location of programs and their parameters. **Records** can be added to the SBT containing any data wanted by the developer. During an intersection in the ray traversal process, a primitive is linked to a record in the SBT if they have the same SBT-index. Meaning, in a simplified manner, the i th entry in the *OptixBuildInput* array is connected to the i th SBT record.

During the sonel tracing stage, the records of the SBT contained information about the scene model such as vertex indices, vertices and normals. As explained earlier, in section 3.1.3, during the sonel gathering step there will be three different types of items in the simulation: models, sonels and sound sources. The structure of the SBT-records is the same for each type and thus includes fields for model data, a pointer to a sonel and a pointer to a sound source as well as an enum indicating the type of primitive. The unused fields are set to *null*.

The pointers have to be point towards the data on the GPU itself. Similar to the AABB data, the buffers for SBT-record data are allocated in one call each. Afterwards the data is copied over in one memory copy operation for each type of data.

After constructing the AABB primitives for the sonel geometry acceleration structure and adding the respective SBT-records, the sound sources are added to the simulation. Sound sources are implemented in the same manner as sonels. However, the structure of a *SoundSource* is quite complex with it containing multiple *SoundFrequency* objects, which in turn contain an array of decibel levels. Before creating an acceleration structure for the sound sources, each *SoundSource* is transformed into a list of *SimpleSoundSource* objects.

A *SimpleSoundSource* represent a sound source that emits sound at **one** timestamp, with **one** frequency, at **one** decibel level from a certain position. All *SimpleSoundSource* objects are collected into one list. This list is subsequently used to construct a geometry acceleration structure. It is constructed similar to the sonel acceleration structure. The AABBs of the custom primitives are constructed by using the position of the sound source and a radius r_{source} , which is a pre-configured radius for all sound sources. Like the sonels, the buffer for the SBT-records is allocated and afterwards the *SimpleSoundSource* objects are copied over.

With all geometry acceleration structures constructed, an instance acceleration structure is created and used as entry point to begin the ray traversal process for the sonel gathering step.

3.4 Sonel Gathering

The goal of the sonel gathering step is to calculate an estimate for the sound energy arriving at a certain receiver. A sound receiver is represented by a sphere of a given radius r_{rec} . The radius can be freely chosen but it will have an impact on the minimum amount of sonels that should be emitted, as seen in equation 2.3. It can receive sound from any direction.

The method for calculating the estimate for a receiver is explained in detail in section 2.3.2 but we will go over a summarized version. For each receiver, a pre-configured amount of **visibility rays** will be cast out. These visibility rays behave similar to how a sonel would. Meaning, each time an intersection is found there is a chance that the ray will be absorbed, reflected diffusely or reflected in a specular manner. When a ray is absorbed it will end the ray traversal process for the respective ray. Diffuse reflections however, is where the gathering of sonels happens. When a diffuse reflection takes place, a nearest neighbour search is started and the sonel map is used to retrieve nearby sonels. During the nearest neighbour search, only sonels of the same frequency as the visibility ray will be added to the list of nearby sonels. Unlike a sonel, the ray traversal is also concluded after a diffuse reflection. A specular reflection is treated like a continuation of the ray and thus no gathering of sonels takes place. After the ray tracing process is finished, the found sonels are used to add energy to an echogram.

As explained earlier, rays will have to be cast for each frequency and each receiver. Like in the tracing step, a choice can be made about which part of the loop to implement on the CPU and which part to implement on the GPU. In my implementation I have opted to move the frequency loop to the GPU and to have the CPU take care of the receiver loop. Iterating over the frequencies can be implemented as a loop within the ray generation program. However, we can make use of the optimisations within the OptiX API by using the launch dimensions. We can do this by setting one axis to be equal to the amount of frequencies that have to be iterated over. The other dimension being equal to the amount of visibility

rays to emit. With this configuration, the ray generation program will be launched once for each ray of each frequency.

During the gathering process the sonels found with the visibility rays have to be stored. Like the sonels during the tracing step, this is done by using a pre-allocated energy-buffer on the GPU. The buffer is made out of *GatherEntry* objects and has a size of $N_{buffergather}$ which is equal to:

$$N_{buffergather} = N_{freq} N_{vis} N_{near_{max}} N_{bytesgather} \quad (3.7)$$

- N_{freq} is the amount of distinct frequencies.
- N_{vis} is the amount of visibility rays to emit.
- $N_{near_{max}}$ is the maximum amount of sonels that are retrieved from the sonel map. Similar to the maximum depth during the sonel tracing step.
- $N_{bytesgather}$ is the size of the *GatherEntry* data structure.

Next to the energy-buffer, an integer hit-buffer is allocated of size $N_{vis} N_{freq}$. This hit-buffer keeps track of how many entries have been added to the energy-buffer for each ray.

It is possible to use the *Sonel* data structure to store the found sonels but it contains data that is not necessary for the construction of the echogram. The required data to add a value to the echogram is a frequency, a sound energy value and a timestamp. Seeing as the frequency can be inferred from the position of the data within the energy-buffer, a *GatherEntry* is composed out of only two values: **energy** and **time**.

Like mentioned in section 2.3.2, before the sonel energy can be added to the echogram it needs to be scaled to take the air attenuation into account. If the parameters for equation 2.8 are copied over to the GPU it is possible to scale the energy during the ray traversal process itself. Scaling the energy on the GPU itself is also recommended for performance reasons.

During the ray traversal it is possible that the visibility rays intersect with a sound source instead of a regular surface. If this intersection is made prior to the ray being specularly reflected, it is referred to as a **direct sound** contribution. In either case, direct sound or not, the sound energy that a sound source contributes is calculated by using equation 2.2. The decibel level of the *SimpleSoundSource* is used to calculate the sound energy levels which are then divided by the amount of visibility rays that were emitted N_{vis} . The contribution of the sound source is stored in the same buffer as the sonels. It can be thought of as if the nearest neighbour search found one sonel for this ray.

The OptiX pipeline used for the gathering step is different from the pipeline used in the tracing step. Next to the ray generation, closest-hit and miss programs, this pipeline also implements an intersection program and any-hit program. The way each program works will now be explained, starting with the ray generation program.

The ray generation programs is executed by OptiX once for each visibility ray, for each frequency. It initializes a ray data structure called *GatheringRayData*, the 64-bit pointer of which is stored in the first two 32-bit unsigned integers of the ray payload. Afterwards a uniformly distributed random direction is generated. Finally, the ray traversal process is initiated. The visibility mask of the ray is configured to only search for intersections in the scene acceleration structure and sound source acceleration structure. This is because we are not yet searching for sonels. Instead we are looking for either sound sources (for direct sound contributions) or for intersections with the scene in order to start the nearest neighbour search. The *GatheringRayData* contains the following fields:

- **Index (64-bit unsigned integer):** A ray index, similar to the rays during the sonel tracing stage.
- **CudaRandom:** A self-written wrapper around the random number generator API of CUDA.
- **Distance (float):** The distance that the ray has traveled.
- **Hits (16-bit unsigned int*):** This field points towards the entry in the hits-buffer for this ray.

The C++ description of the ray data can be found in listing 3.7.

Listing 3.7: C++ description of the ray data class used during the sonel gathering stage.

```

class GatheringRayData {
    uint64_t index;
    CudaRandom cudaRandom;
    float distance;
    uint16_t* hits;
}

```

The ray index represents a unique identifier for the ray and is also used to access the energy-buffer. It is calculated by using the frequency and visibility ray indices and it is incremented each time a value is stored in the buffer. The formula of calculating the ray-index is:

$$index_{ray} = N_{near_{max}} N_{vis} index_{freq} + N_{near_{max}} index_{vis} + N_{added} \quad (3.8)$$

- $N_{near_{max}}$ is the maximum amount of values that can be stored for a single ray.
- N_{vis} is the amount of visibility rays that are emitted for each frequency.
- $index_{freq}$ is the frequency index. It ranges from $[0, N_{freq}[$, with N_{freq} being the amount of distinct frequencies in the simulation.
- $index_{vis}$ is the visibility ray index. It ranges from $[0, N_{vis}[$.
- N_{added} is the amount of values that have been added to the energy-buffer.

Continuing with the *CudaRandom* field, this field is initiated by using a seed value. As previously mentioned, this seed value needs to be unique for each execution of ray generation program. The approach used here is the same as the one used for the *CudaRandom* field of the sonel tracing stage. A frame-index value is added to the pipeline which increments with the size of the energy-buffer after every execution of the pipeline. By adding the frame-index and ray-index together in the ray generation program, a unique value is calculated that can be used as a seed value.

The distance field is initialized with 0. After an intersection has been encountered, the value of this field is increased by the distance from the ray origin and location of the intersection. The field "hits" is initialized by constructing a pointer that points to the integer storing the hits for this ray. This is done by taking the starting pointer of the array $pointer_{hit_{start}}$ and adding an offset to it. This offset is equal to $index_{freq} N_{vis} + index_{vis}$. The equation for the pointer calculation is as follows:

$$pointer_{hit_{ray}} = pointer_{hit_{start}} + index_{freq} N_{vis} + index_{vis} \quad (3.9)$$

After the ray generation program, the programs that can be executed are the miss program, closest-hit program, intersection program and any-hit program. If the miss program is executed it means that the ray has missed every object within the scene. It is also possible that the ray distance of the closest hit was more than the configured maximum distance. However, in my implementation the maximum distance is set to 10^{20} so this case is unlikely to happen. When the miss program is executed it means that the ray traversal of this ray has ended.

When the closest-hit program is executed it means that an intersection was encountered with either the scene or a sound source. As explained in section 3.3, a primitive can be linked to an SBT-record. The data within the SBT-record is free to be chosen. In my implementation, it contains pointers to either model data, a *SimpleSoundSource* or *Sonel*, along with an enum representing the type of object. If the intersection is with a sound source we need to add a value to the energy-buffer. The sound energy is calculated by using equation 2.2 and data from the sound source. Before the energy can be added it needs to be scaled to account for air attenuation. This is done by employing equation 2.8. The value of the air absorption coefficient m is retrieved from the parameters that were copied to the GPU before the launch of the pipeline. The distance traveled through the air l is set to the distance between the ray origin and the intersection. After the energy is scaled it is added to the energy buffer and the hit counter is incremented.

If the intersection is with an object in the scene the ray can be absorbed, reflected diffusely or reflected specularly. If the ray is absorbed the ray traversal is ended and a *null* value is added to the energy-buffer. A specular reflections is treated like a continuation of the ray. The direction of the outgoing ray is calculated as explained in the sonel tracing implementation, see 3.2. As it is treated like a continuation of the ray, the visibility mask of the new ray is equal to the visibility mask of the ray generation program. This means that the new ray is also configured to be able to intersect with the scene and sound sources. Before starting the ray traversal of the new ray, the distance field of the ray data is updated by adding the distance between the current intersection and the ray origin.

In case of a diffuse reflection we need to start a nearest neighbour search. Like the closet-hit program in the tracing step, the location of the intersection is calculated by interpolating the vertices using barycentric coordinates provided by OptiX. Afterwards the surface normal is either retrieved directly from the SBT-record or it is calculated from the vertices. The surface is then used to initiate the ray traversal of a new ray. This ray is configured to only intersect with the sonels in the sonel map. Rays can also be configured with a minimum and maximum value. As we are only interested in sonels near the intersection position the distance can be very small. In my implementation the minimum value is set to 0.001 and the maximum value is set to 0.01. Before launching the ray traversal of the nearest neighbour search ray, the distance field of the ray data is updated. After starting the ray traversal process of the nearest neighbour search ray, the programs that can be executed are the intersection program, any-hit program and miss program.

An intersection program is called when the OptiX API needs to calculate whether or not a primitive intersects with a ray. For common primitives such as triangles, this program is built-in into the OptiX API. Because both sound sources and sonels are defined as custom primitives, an intersection program has to be added to the pipeline. Firstly, the implemented intersection program first uses the linked SBT-record to determine which type of object we are calculating the intersection for. For sound sources, it is important to calculate if the sphere representing the sound source intersects with the ray. The reason why it is important is because we need to provide OptiX with an accurate intersection location to determine if this intersection is actually the closet one or if it isn't. For sonels an accurate intersection is not required as we can simply check if the distance between the ray origin and the sonel is less than the search radius r_{vis} and if the frequency of the sonel is equal to the frequency of the ray. If a ray does intersect with an object, the any-hit program is launched. However, in my implementation the any-hit program is disabled for the scene and sound source acceleration structures as it is not used in these cases. In other words, it is only enabled for the sonel map.

As a reminder, the any-hit program is called whenever an intersection is found and it may or may not be the closest intersection. In an any-hit program, a call can be made to the OptiX API to ignore the current intersection. If this is done, the ray traversal continues. In my implementation, the task of the any-hit program is to perform the nearest neighbour search. When the any-hit program is executed for an intersection with a sonel, the energy of the sonel is attenuated and added to the energy-buffer. To calculate the timestamp of this entry, the total distance l_{total} is divided by the speed of sound v_{sound} , to which the sonel time offset t_{offset} is added. The total distance l_{total} is calculated by adding the distance the sonel has traveled l_{sonel} to the distance the ray has traveled l_{ray} .

$$t_{entry} = \frac{l_{sonel} + l_{ray}}{v_{sound}} + t_{offset} \quad (3.10)$$

After adding a value to the energy-buffer the ray index and the hit counter are increased by one. Finally, if the buffer is not full, the ray intersection is ignored and thus the ray traversal/nearest neighbour search continues. If the buffer is full, the ray traversal is ended for this ray. When the nearest neighbour ray is finished, the execution returns to the closet-hit program. With this, the closet-hit program is also finished. Once every ray is finished, execution is return to the CPU and the energy-buffer and hit-buffer are copied over from the GPU. After which, echogram construction is started.

3.5 Echogram construction

The construction of the echogram is the last step in the sonel mapping process. To recap, an echogram is a table of decibel levels per frequency over a certain time period. The x-axis of an echogram represents the time at which a receiver hears sound at a certain decibel level. Given a total simulation time of t_{max} ,

the amount of buckets is equal to $\frac{t_{max}}{t_{res}}$. The y-axis of an echogram indicates the frequency of a certain decibel level. Each value of the echogram is initialized to 0.

After the sonel gathering step, we are left with an energy-buffer. The energy buffer exists out of N_{freq} pages, one for each frequency. The page itself is constructed out of N_{vis} rows, one for each visibility ray. Each row contains at most $N_{near_{max}}$ values. Before the entries in each row can be added to the echogram, they are divided by $N_{near_{actual}}$ which represent the amount of sonels that were found during the nearest neighbour search of a single ray. As part of the density estimation, it is assumed that each sonel contributes to the same bucket.[4] After this division, the energy value can be added to the appropriate bucket. Each entry in the energy buffer has a time value t_{energy} . This timestamp is transformed into an index for the echogram by dividing it with the time resolution t_{res} as can be seen in equation 2.7. This index represents the column of the echogram that the energy has to be added to. The row of the echogram corresponds with the page that is currently being handled. The code of this construction can be found in listing 3.8.

Listing 3.8: C++ implementation of the echogram construction.

```
for (uint32_t frequency = 0; frequency < nFreq; frequency++) {
    for (uint32_t ray = 0; ray < nVis; ray++) {
        uint64_t rayStart = frequency * (nVis * nNearMax);

        for (uint32_t sonelIndex = 0; sonelIndex < nNearMax; sonelIndex++) {
            GatherEntry& entry = energyBuffer[rayStart + sonelIndex];

            uint32_t timeIndex = floor(entry.time / tRes);
            echogram[frequency][timeIndex] += entry.energy /
                                                static_cast<float>(rayHits[ray]);
        }
    }
}
```

The echogram at this point contains sound energy values. It can be transformed to a human-readable format by converting it back to decibel levels. This is done by taking the sound energy value and using the follow equation:

$$L = 10 \times \log_{10}\left(\frac{E_{sound}}{10^{-12}}\right) \quad (3.11)$$

Finally, the echogram is completed and with it the sonel mapping algorithm is finished as well.

3.6 Validation

3.6.1 i-Simpa

I-Simpa is an open source room acoustic simulation software. It provides two ways of generating results, a "classical theory or reverberation" and an "SPPS" mode. The SPPS mode geometric acoustics simulation making use of ray tracing. It does not use the sonel mapping technique. The software will be used in this section as a point of comparison. The results of i-Simpa will be used as the ground truth. It is also used as a point of comparison to determine the performance gains of my sonel mapping GPU implementation.

3.6.2 Test Setup

To make it possible to compare multiple evaluation scenarios, a baseline setup has been configured. All the configured parameters can be found in the list below. Each of these parameters have a listed **default** value. The default value is used in the upcoming tests unless otherwise specified. The tests were performed on a computer with an RTX 3080 for GPU and Ryzen 5900X for CPU.

- **Scene:** The 3D-model of the scene is a cube shaped room with a width, height and depth of 4 meters.
- **Sound sources:** In the simulation there is only one sound source which has 1 frequency. This frequency has only one decibel value to emit. The sound source is located at position (3.5, 3.5, 3.5). It is an omnidirectional sound emitter, meaning the sound emission is not directed and it emits sound in every direction. The sound source has a radius of 0.5m.
- **Sound receivers:** There is one sound receiver configured with a radius of 0.31m. This sound receiver is located at (0.5, 0.5, 0.5). Meaning, the sound source and receiver are in opposing corners. Like the sound source, the sound receiver is also omnidirectional and receives sound from every direction.
- **Frequencies:** As previously mentioned, there is only one frequency defined. The frequency influences the air absorption coefficient. The frequency is set to 250hz. Because there is only one frequency it means that the frequency size N_{freq} is equal to 1.
- **Sound Energy:** The frequency is configured to emit sound at a decibel level of 90dB.
- **Air absorption coefficient:** The air absorption coefficient is automatically determined by my implementation dependant on the frequency. As it is not clear which air absorption coefficients are used in i-Simpa, the air absorption coefficient is forcefully overwritten in i-Simpa to match the coefficient in my implementation. For 250hz at 50% humidity this is equal to 0.08dB/m.
- **Material properties:** The properties of each surface have been configured to have an absorption probability of 0.1. The probabilities of specular and diffuse reflections are set to 0.45.
- **Maximum sonel depth:** The maximal amount of bounces that a sonel can make is configured to be 32. With an absorption probability of 0.1 the average depth is equal to 10. The probability of a sonel of reaching a depth of 32 is equal to $(1 - 0.1)^{32} \approx 0.0343 \approx 3\%$.
- **Sonel search radius:** The search radius during the nearest neighbour search of the gathering step is set to 0.15m.
- **Sonel amount:** The amount of sonels to emit during the sonel tracing stage is set to 100 000.
- **Visibility rays:** The amount of visibility rays that are emitted during the sonel gathering step is configured to 5000.
- **The maximum amount of nearest neighbours:** During the nearest neighbour search, there is a maximum amount of sonels that can be retrieved per ray. This value is set to 50t.
- **Simulation time:** The maximum time duration of the simulation is set to 2s.
- **Time resolution:** The time resolution t_{res} is set to 5ms or 0.005s.

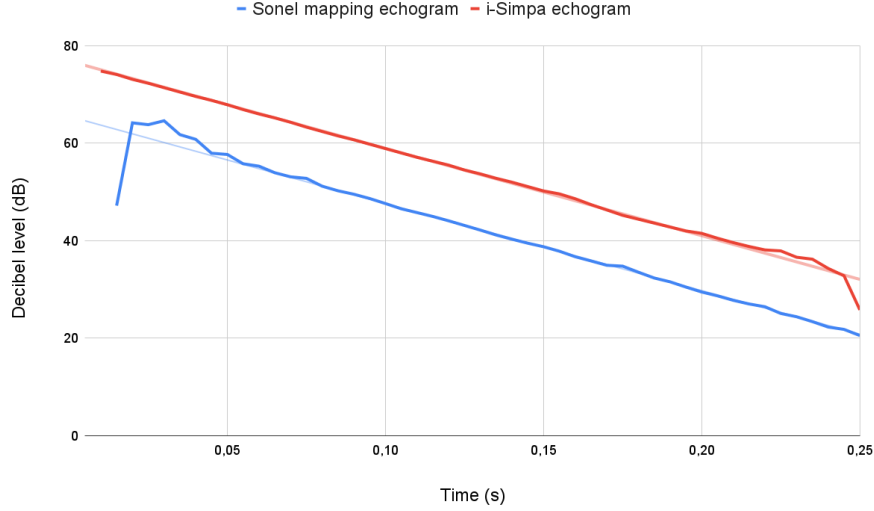


Figure 3.3: A graph comparing the decibel levels of i-Simpa against the sonel mapping implementation. The trend line indicates the decay rate. The first 0-values at the start were removed as they influenced the trend line.

3.6.3 Echogram Validation

The final step of the sonel mapping algorithm creates an echogram for each receiver. An echogram is a table of decibel values in relation to both the timestamp and frequency. To compare the accuracy of the implemented sonel mapping algorithm we can compare the generated echograms against the echograms generated by i-Simpa. The test setup for this comparison is a simplified setup so it is clear how each change in parameters influences the generated echogram. The configuration for these tests can be found in section 3.6.2.

With this configuration an echogram was generated for both i-Simpa and my implementation. As can be seen in figure 3.3, the decibel level between the two implementation do not match. After removing outliers, the mean difference between the two implementation is $11.3382dB$. This is a significant difference, especially accounting for the fact that the decibel scale is logarithmic and a decrease of $10dB$ corresponds with a tenfold decrease in energy.

Scaling the sonels

To better understand these findings, a multitude of simulations has been calculated. The first batch of simulations were part of the **sonel-scaling** test. The purpose of this test is to see what effect the amount of emitted sonels has on the echogram, with all other parameters remaining unchanged. Simulations were calculated with the sonel amount being equal to 5 000, 25 000, 50 000, 100 000, 500 000, 1 000 000. Each of these simulations were executed with 5000 visibility rays and the maximum amount of nearest neighbours was set to 50. When the echograms are plotted against the echograms of i-Simpa, as can be seen in figure 3.4, it appears that the difference between the ground truth increases as the amount of sonels increases. This is counter-intuitive as ray tracing algorithms should converge towards the correct answer. By increasing the amount of rays or sonels, the estimate of the algorithm should get more accurate. This is not the case for this implementation.

If we take a look at the graph in figure 3.5 it shows that the mean decibel difference increases alongside the amount of sonels that were emitted. This is an unwanted result. The origin of this issue could be due to a bug or a wrong interpretation of the sonel mapping algorithm. If we take a closer look at how the sonel map is used to estimate the energy level at a certain intersection location we can find the reason for this issue. In my understanding of the sonel mapping algorithm, when an intersection is encountered during the gathering step the energy of the sonel is divided by the amount of sonels. The amount of sonels found is capped at a maximum $N_{near_{max}}$. The energy added to the echogram by each sonel is

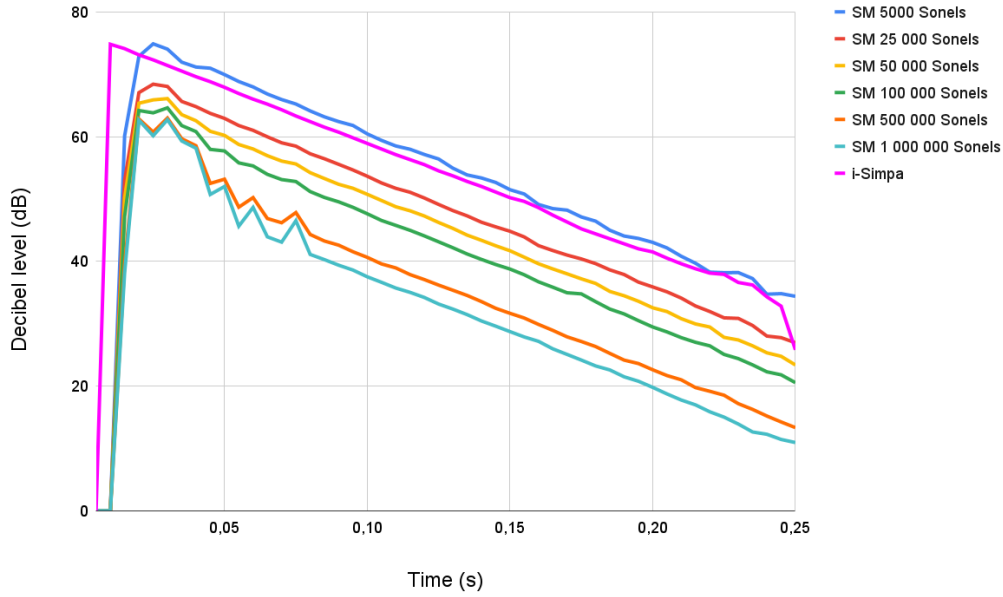


Figure 3.4: A graph plotting the decibel levels of several simulations against the echogram of i-Simpa. Each simulation has a different amount of emitted sonels as seen on the legend.

equal to:

$$E_{sonel_s} = \frac{E_{sonel_o} e^{-ml}}{N_{near}}$$

. If we assume that during each search, the maximum amount of sonels is found we can substitute N_{near} for $N_{near_{max}}$:

$$E_{sonel_s} = \frac{E_{sonel_o}}{N_{near_{max}}}$$

The energy with which the sonel is emitted depends on the decibel value and the amount of sonels as seen in equation 2.2. If we substitute E_{sonel_o} for this value the equation becomes:

$$E_{sonel_s} = \frac{(\frac{10^{L/10}}{N_{sonel}} \times 10^{-12}) e^{-ml}}{N_{near_{max}}}$$

If we assume that the value of l , the distance the sonel traveled until it reached the receiver, remains similar for each execution of the simulation. Then each parameters is constant and the only parameter having influence on the energy added to the echogram is the amount of emitted sonels N_{sonel} . As divisor of the decibel energy, if the sonel amount increases, the energy that is added to the echogram is decreased.

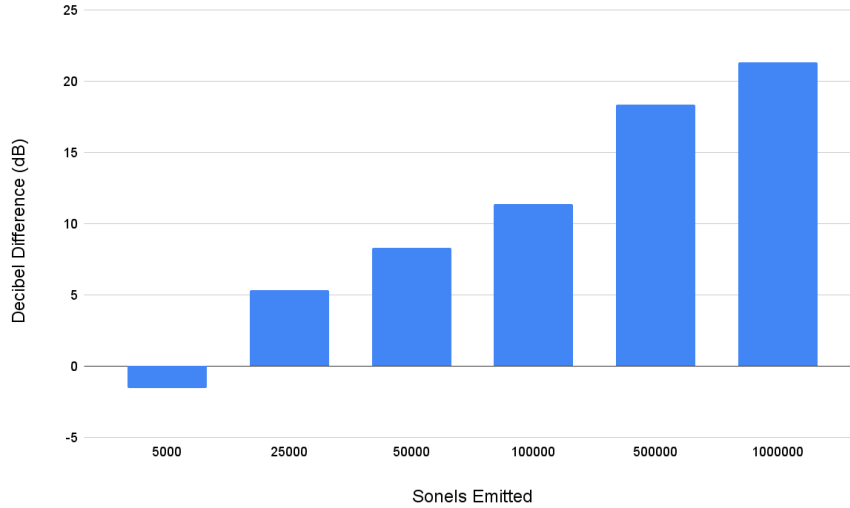


Figure 3.5: A bar-chart of mean decibel differences between each of the simulations and i-Simpa, plotted against the amount of sonels that were emitted.

Scaling the visibility rays

Another experiment we can do to better understand these results is by leaving all parameters static except for the amount of visibility rays that are emitted N_{vis} . A number of simulations were calculated with the amount of visibility rays being equal to 1000, 2500, 5000, 10 000, 50 000, 100 000, 500 000, 1 000 000 and 10 000 000. The amount of sonels emitted during these simulations remained at a constant 100 000. For the nearest neighbour search, the maximum amount of nearby sonels was configured to be 50. The comparison of echograms can be seen in figure 3.6. By scaling the amount of visibility rays it appears that we have the opposite problem as when scaled the amount of sonels. An increase in visibility rays corresponds with an increase of total energy. Again, increasing the amount of visibility rays does not seem to cause the algorithm to converge to the correct estimate. Even more, the peak of the echogram when 10 million visibility rays were used was $93.22dB$. This is 3 decibels higher than the input decibel value of $90dB$. The mean difference between the echogram of the test cases against the ground-truth of i-Simpa is plotted out in image 3.7.

The reason for this issue can again be due to a bug or a wrong interpretation of the sonel mapping algorithm. In the current implementation each visibility ray start a nearest neighbour search upon a diffuse reflection. The energy of these sonels is scaled by the amount of sonels found. It is not scaled in relation to any property of the receiver. If more visibility rays are emitted, it means more sonels are found, which in turn means more energy is added to the echogram.

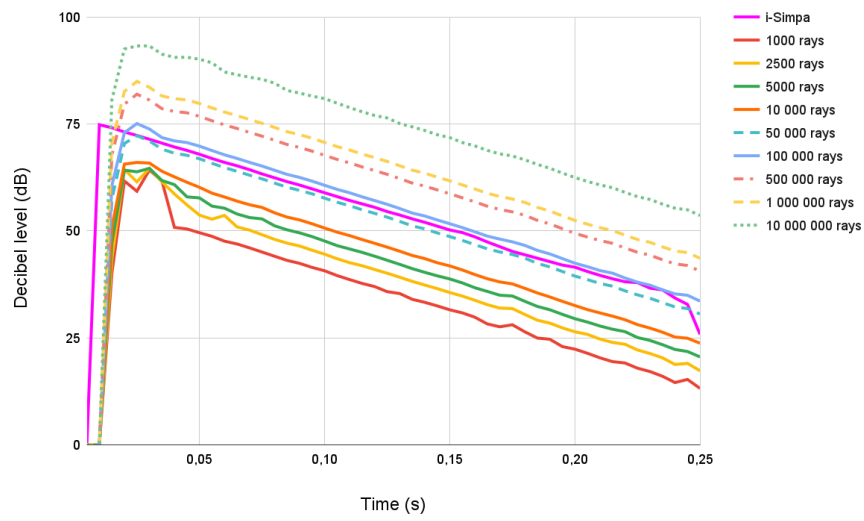


Figure 3.6: A graph containing all the echograms during the visibility ray scaling experiment.

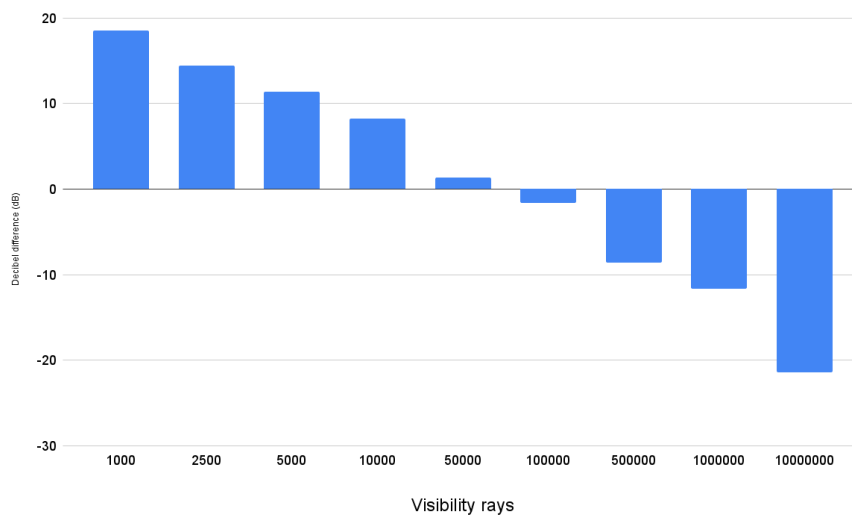


Figure 3.7: A bar-chart of mean decibel differences between each of the simulations and i-Simpa, plotted against the amount of visibility rays emitted.

Scaling the maximum amount of nearby sonels

The same experiment was repeated on the maximum amount of nearby sonels as it is a parameter that is used to scale the energy of a sonel during the gathering step. The values that were tested include 10, 25, 50, 100, 250, 500. Looking at the echogram of these simulations, the parameter does not seem to significantly influence the energy calculation. The mean decibel differences between the simulation and i-Simpa echogram can be seen in the following table.

$N_{near_{max}}$	10	25	50	100	250	500
Diff (dB)	11.363	11.349	11.338	11.320	11.309	11.311

3.6.4 Performance Analysis

One of the goals of this thesis is to improve the execution time of the sonel mapping algorithm so that it is sufficiently fast to be used for interactive purposes. The ray tracing parts of the algorithm were implemented by using the OptiX API, which runs on the GPU. Implementing an algorithm on a GPU has its own unique set of challenges. One of the biggest one being the use of memory. To ensure maximum performance it is important to limit the amount of dynamically allocated memory. Next to memory management, it is also important to limit the amount of branching paths the GPU code can take. Additionally, extra steps need to be taken to set up the GPU for computation and for exchanging data between the CPU and GPU. Therefore it is not evident to achieve a significant performance increase. To evaluate the implementation of the algorithm, both on CPU and GPU we can take a look at how the execution increases by changing the input. The setup for these tests can be found in section 3.6.2. Certain parameters have been changed from their defaults to provide more useful data.

Performance scaling with sonels

For this performance test we will look at how the performance of the software changes with an increase to the amount of sonels emitted. We will look both at the execution time and the amount of peak memory usage. For this test a variety of metrics have been collected as seen in the table below. There were a total of 15 time-related metrics collected for this test. To make it easier for the reader to read the data, a subset of metrics are used in this analysis. This subset is composed out of metrics that are the most relevant for the current test.

N_{sonel}	$Total(ms)$	$T_{up}(ms)$	$T_{sim}(ms)$	$T_{dl}(ms)$	$Map_c(ms)$	$G_{hr}(ms)$	$G_{sim}(ms)$	$Mem(GB)$
25000	144.89	2.48	0.38	14.13	40.51	78.51	4.05	1.5
50000	272.07	3.57	0.56	30.71	75.17	149.63	6.66	1.7
100000	535.88	5.96	0.88	54.11	144.71	309.28	13.17	2.0
250000	1298.80	13.00	1.72	143.32	348.68	748.29	31.18	2.9
500000	2575.03	28.50	4.03	270.67	738.35	1449.42	60.94	4.7
1000000	5188.12	48.74	6.54	591.75	1452.01	2921.37	120.49	8.1
2500000	13981.10	118.96	16.06	1394.13	4487.68	7557.08	282.79	18.1

- N_{sonel} Is the amount of sonels that were emitted from the sound source.
- T_{up} Is the amount of time spent preparing and uploading the required data to the GPU for the sonel tracing stage. This includes allocating the buffer to store the sonels in.
- T_{sim} Is the amount of time that was spent by the GPU during the sonel tracing step.
- T_{dl} Is the amount of time it took to copy the sonel data from the GPU to the CPU in order to construct the sonel map.
- Map_c Is the amount of time that was needed to upload the sonel data to the GPU and to prepare an AABB for each sonel.
- G_{hr} Is the amount of time it took to generate all the SBT hit-records for the gathering step.
- G_{sim} Is the amount of time that was spent by the GPU during the sonel gathering step.
- Mem Is the amount of peak memory consumption during the execution of the software.

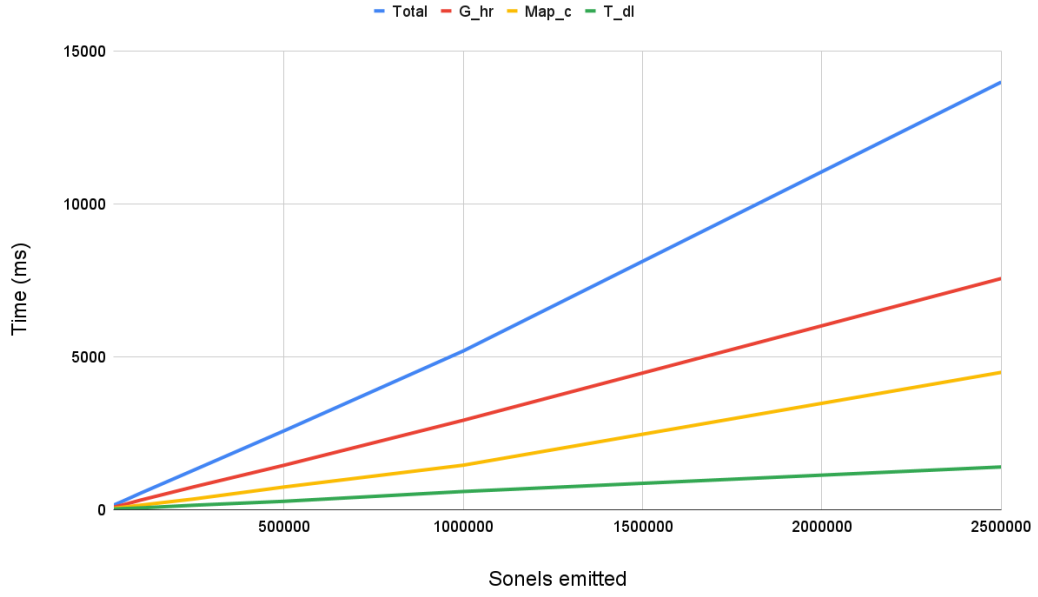


Figure 3.8: A graph plotting the execution times $Total$, G_{hr} , Map_c and T_{dl} against the amount of sonels emitted. With $Total$ being the total execution time of the implementation. G_{hr} being the time required to construct SBT hit-records for the gathering step. Map_c being the time required to construct the sonel map. This includes uploading the sonel data to the GPU and constructing an AABB for each sonel. Finally, T_{dl} is the time required to download the sonel data from the GPU to the CPU after the tracing step.

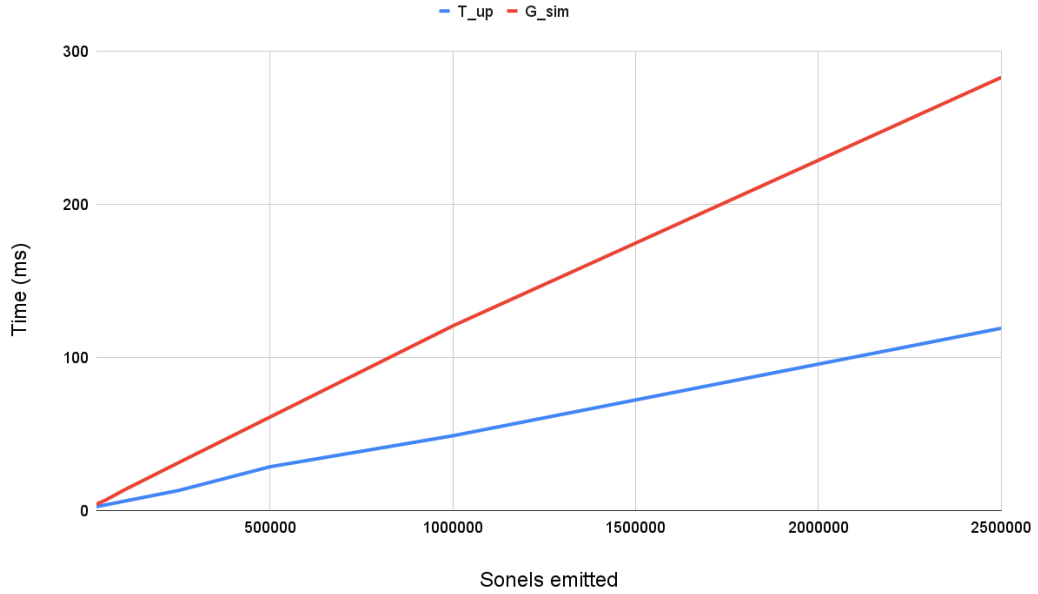


Figure 3.9: A graph plotting the execution times T_{up} and G_{sim} against the amount of sonels emitted. With T_{up} being the time required to allocate the sonel buffer on the GPU and to upload the required data for the sonel tracing step. G_{sim} is the time required to execute the gathering step on the GPU.

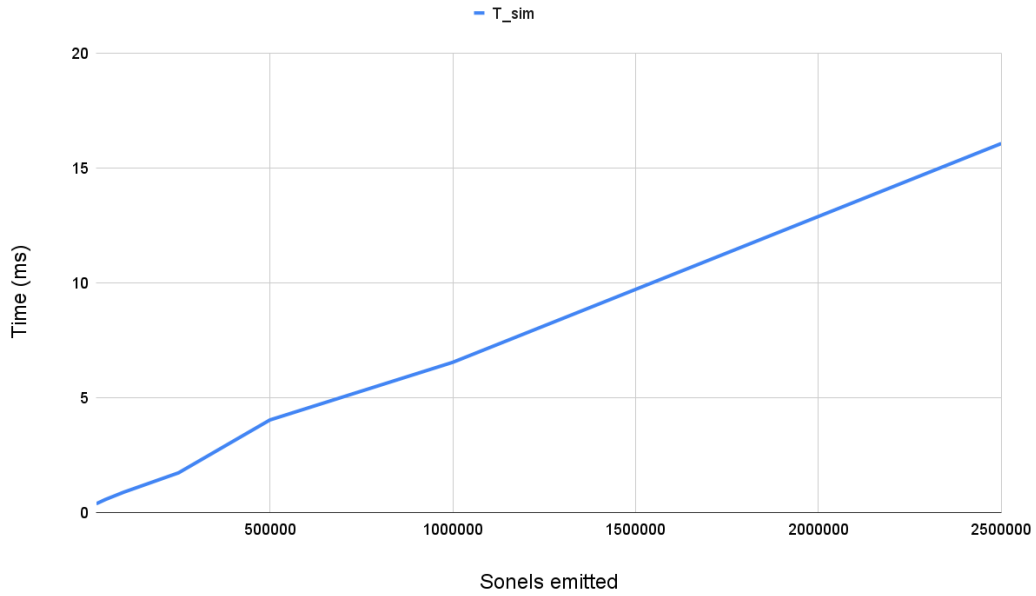


Figure 3.10: A graph plotting the execution time of T_{sim} against the amount of sonels emitted. With T_{sim} being the time required to execute the sonel tracing step on the GPU.

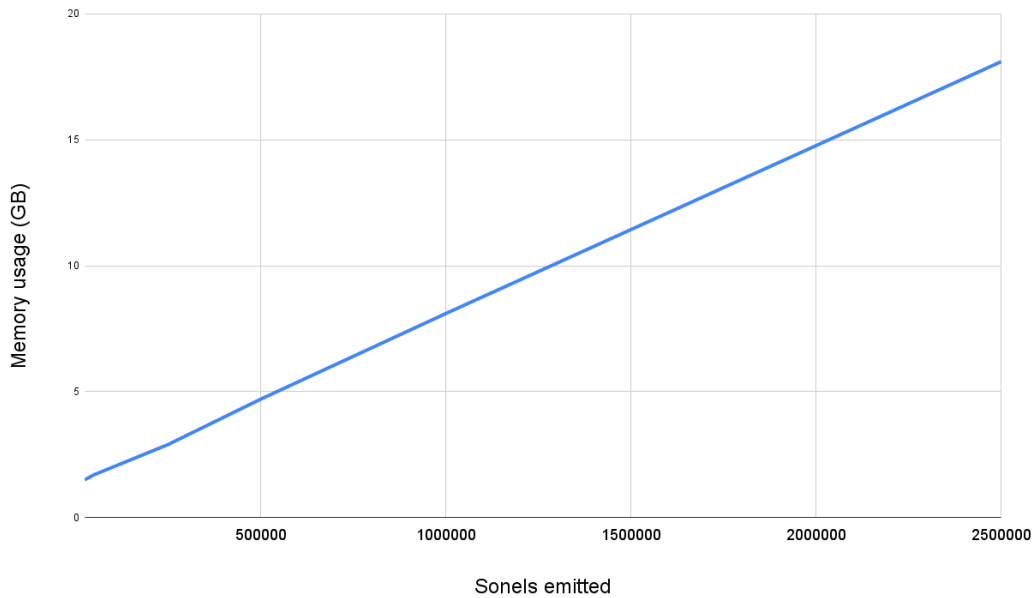


Figure 3.11: A graph plotting the amount of peak memory required against the amount of sonels emitted.

A few things can be gathered from these tests. Firstly, if we look at the graph in figure 3.8, which depicts the total execution time, G_{hr} , Map_c and T_{dl} in relation to the amount of sonels emitted, we can see that there is a linear relation. This is to be expected. Each additional sonel added to the amount of sonels emitted from a sound source corresponds with multiple additional hits. This increases the amount of data that has to be downloaded and thus we see a linear increase in T_{dl} . The same principle explains the increases to G_{hr} and Map_c . Each additional hit requires an extra item to be added during the construction of the sonel map, as well as an additional hit record. Again, this increase seems to be linear.

This linearity can be seen in the other metrics as well. By plotting out the values of T_{up} and G_{sim} we can see that both steps show a linear relation with N_{sonel} , as can be seen in the graph from image 3.9. The increase in T_{up} is somewhat surprising as the amount of data that is copied over from the CPU to the GPU does not increase. However, the increase in sonels corresponds with an increase to the required sonel buffer size. The sonel buffer is left uninitialized as the memory is not set to any value after allocation. Because of this it seems that allocating more memory takes additional time. For G_{sim} an increase in sonels results in an increase of sonel AABBs to check for the nearest neighbour lookup. As a result, an increase to N_{sonel} corresponds to a linear increase to G_{sim} .

Continuing with the graph in image 3.10. We can see that the execution time of the sonel tracing step on the GPU also increases linearly with N_{sonel} . This is nominal as an increase to the amount of emitted sonels means additional rays have to be traced. Additionally, we can take a look at the amount of memory required. Looking at image 3.11 we can see that the amount of required memory also increases linearly with N_{sonel} .

Lastly, from the data of these tests we can note that the steps G_{hr} , Map_c and T_{dl} account for 95% of the execution time. In other words, the majority of the execution time is spent copying data from the GPU or preparing data for use on the GPU.

Performance scaling with visibility rays

The purpose of this batch of tests is to see how the execution time changes in relation to the amount of visibility rays used. Explanation of the metrics used in the table below can be found in previous subsection 3.6.4. However, an additional metric G_{parse} has been added, which represents the time spent on the CPU for the parsing of the results of the gathering step and the construction of the echogram. The setup for these tests can be found in section 3.6.2.

N_{vis}	$Total(ms)$	$T_{sim}(ms)$	$T_{dl}(ms)$	$Map_c(ms)$	$G_{hr}(ms)$	$G_{sim}(ms)$	$G_{parse}(ms)$	$Mem(GB)$
1000	513.59	0.88	58.14	139.30	292.52	9.78	0.29	2.0
2500	553.83	0.87	54.24	148.87	324.61	12.41	0.80	2.0
5000	549.85	1.27	54.77	144.58	321.53	13.31	1.55	2.0
10000	534.45	0.87	55.95	158.05	289.46	13.38	2.48	2.0
25000	527.15	0.89	53.39	140.11	295.89	16.81	6.38	2.0
50000	581.52	0.88	59.90	156.34	309.49	30.03	12.07	2.0
100000	602.02	0.89	56.23	145.84	310.96	49.11	25.93	2.0

Seeing as the parameter N_{vis} is only used for the gathering step we should expect to see no relation between N_{vis} and the tracing step metrics. By looking at the graphs in image 3.13 and the data in the table above, we can confirm that the tracing steps T_{sim} and T_{dl} do not seem to increase alongside N_{vis} . The same goes for the sonel map steps Map_c and G_{hr} as can be seen in the graph from image 3.14. However, for the gathering step metrics G_{sim} and G_{parse} there does seem to be a linear relation as can be seen in image 3.13. This is to be expected as an increase to N_{vis} results in additional rays having to be traced. This in turn causes an increase to the amount of nearest neighbour searched that need to be executed. The addition of these rays also results in an increase to the amount of data that needs to be parsed to construct the echogram.

As previously mentioned, the steps G_{hr} and G_{sim} are only a fraction of the total execution time, therefore the impact of N_{vis} is negligible. However, for interactive applications targeting frame times less than 33ms, choosing an appropriate number of visibility rays will be important as we want the time of G_{sim} to be as low as possible. When looking at the memory usage, it can be noted that the amount used does not seem to be impacted heavily by an increase of visibility rays.

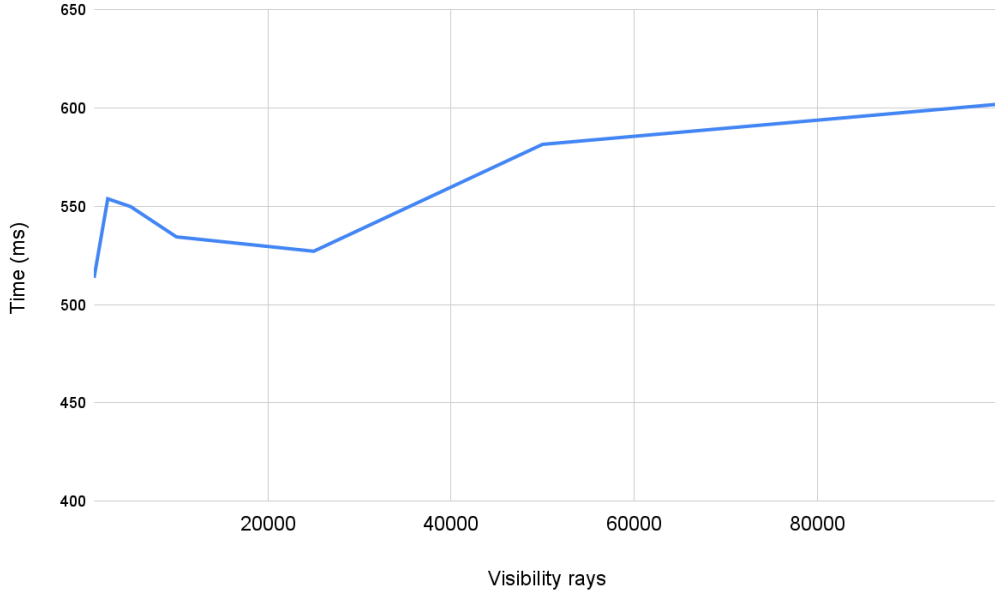


Figure 3.12: A graph plotting the total execution time of the sonel mapping implementation against the amount of visibility rays used during the gathering step.

The reason behind this is because the amount of data used per visibility is insignificant compared to the sonels. In the implementation a *Sonel* consists out of 40 bytes. Each sonel also requires a an entry for the sonel map acceleration structure and a hit record. Whilst an *GatherEntry* for the gathering steps requires only 12 bytes. During the sonel tracing stage, each sonel that is emitted will generate multiple hits that are stored a *Sonel* as well. This is not the case for the gathering step. With the setup used during these tests, the average amount of sonels to store equals 10 million (see equation 3.3). If we only assume 40 bytes per sonel, this equates to 400 million bytes used. Compared to the visibility rays, where each ray only consumes $N_{buffer} = N_{bytes} \times N_{near_{max}} = 12 \times 50 = 600$ bytes, it only takes 60 million bytes to store. In reality, the amount of bytes per sonel is more than 40 because of the aforementioned reasons.

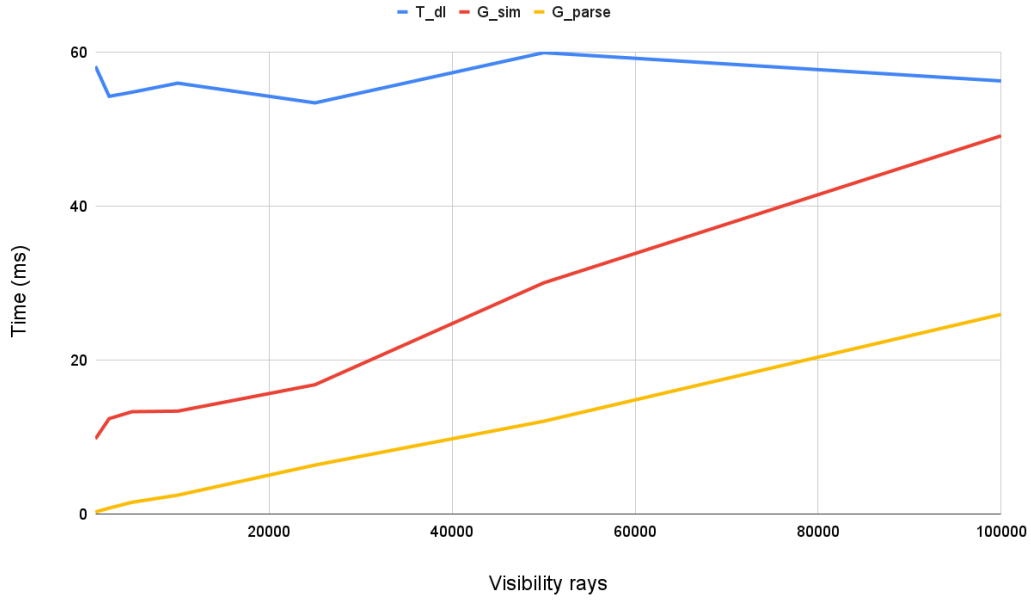


Figure 3.13: A graph plotting the execution times T_{dl} , G_{sim} and G_{parse} against the amount of visibility rays used during the gathering step. T_{dl} represents the time required to download the sonel data from the GPU after the sonel tracing stage. With G_{sim} being the execution time required to perform the sonel gathering step on the GPU itself. G_{parse} is the time spent by the CPU after the gathering step to parse the results and to create the echogram.

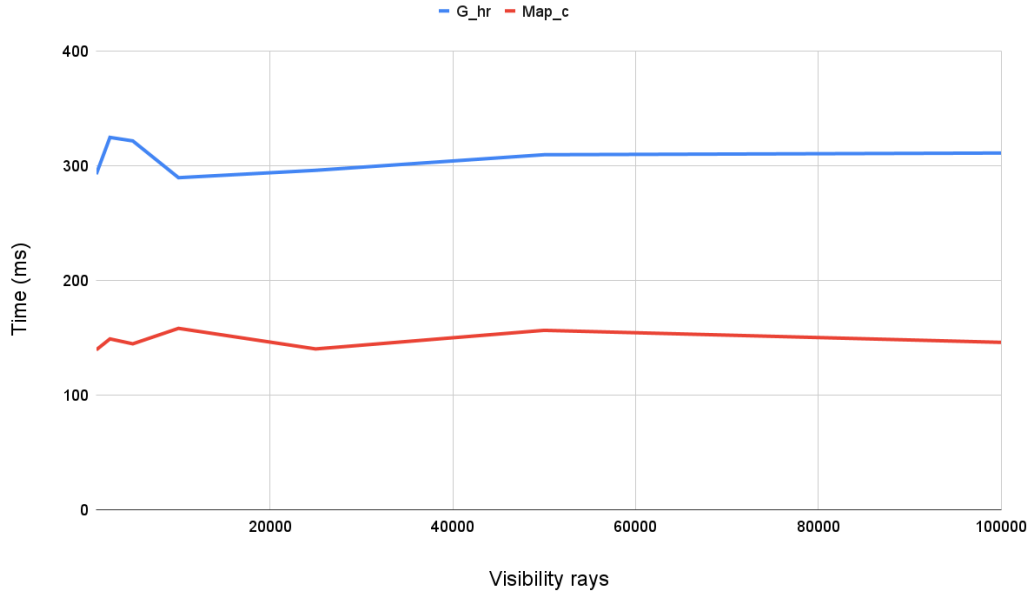


Figure 3.14: A graph plotting the execution times of G_{hr} and Map_c against the amount of visibility rays used during the gathering step. With G_{hr} being the time required to construct the SBT hit-records for the gathering step and Map_c being the time required to construct the sonel map after the sonel tracing step.

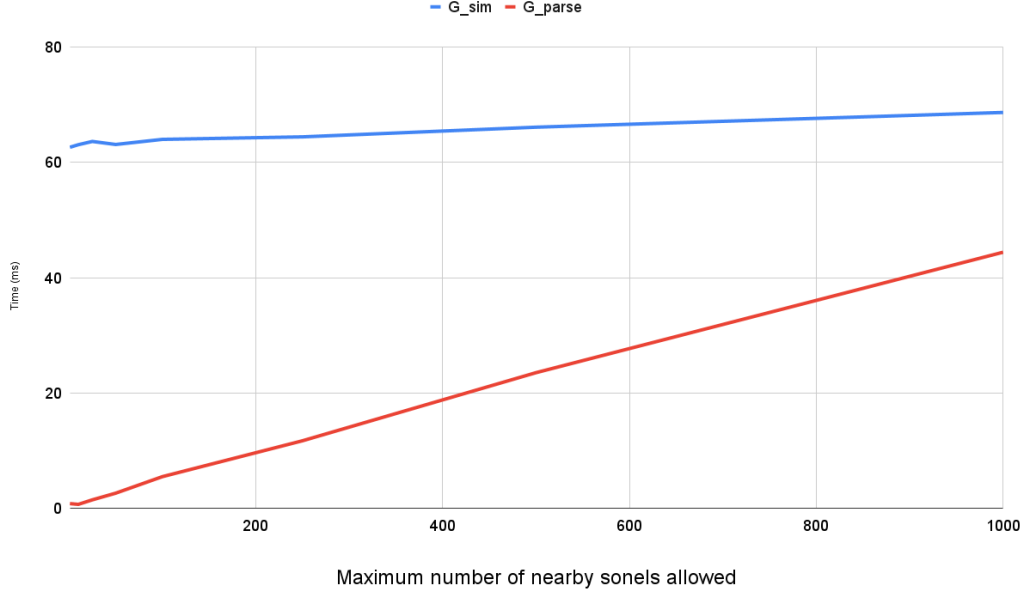


Figure 3.15: A graph plotting the execution times of G_{sim} and G_{parse} against the maximum amount of nearby sonels that may be found during the nearest neighbour search of the gathering step $N_{near_{max}}$. With G_{sim} being the time required to perform the gathering step on the GPU and G_{parse} being the time required to parse the results of the gathering step and to construct the echogram.

Performance scaling with maximum amount of nearest sonels

This sets of tests aim to provide an insight to the performance of the application in relation to the maximum amount of sonels that may be found during the nearest neighbour search or $N_{near_{max}}$. The tests were conducted with the setup described in section 3.6.2. The amount of sonels emitted was set to 500000 and the amount of visibility rays was set to 10000. Descriptions of the metrics used in the table below can be found in section 3.6.4. There is one new metric added to this table which is Ray_{sat} , it represents the percentage of saturated visibility rays. A saturated visibility ray is a ray where the amount of sonels found during the nearest neighbour search N_{near} is equal to $N_{near_{max}}$. This metric was added because an increase in $N_{near_{max}}$ can result in a nearest neighbour search where $N_{near} < N_{near_{max}}$, and thus a ray that requires less work than a saturated ray. A comparison between the performance of each test can only be done if the amount of saturated rays is close to each other.

$N_{near_{max}}$	$Total(ms)$	$T_{sim}(ms)$	$T_{dl}(ms)$	$G_{sim}(ms)$	$G_{parse}(ms)$	$Ray_{sat}(\%)$	$Mem(GB)$
1	2610.64	4.03	275.31	62.59	0.84	73.04	4.6
10	2541.00	3.19	277.12	63.03	0.70	72.17	4.7
25	2616.74	3.46	272.90	63.60	1.49	72.16	4.6
50	2637.68	3.51	293.25	63.07	2.64	72.14	4.6
100	2598.75	3.94	300.03	63.96	5.51	71.80	4.7
250	2575.49	4.16	269.62	64.39	11.73	72.03	4.6
500	2534.82	3.44	266.59	66.07	23.53	71.84	4.6
1000	2627.27	4.31	278.66	68.62	44.40	71.15	4.7

As can be seen from the table, the ray saturation levels are within 1% of each other which means a performance comparison can be made. The term $N_{near_{max}}$ is only used during the gathering step therefore we expect to see no correlation between timings during the tracing step and $N_{near_{max}}$. This is indeed the case as T_{sim} and T_{dl} do not seem impacted. Furthermore, from the graph in image 3.15 we can see a linear relation between G_{parse} and $N_{near_{max}}$. Given that the ray saturation stay consistent this result is to be expected. The parsing of the results of the gathering step happens in a single-threaded function on the CPU. As a result, given that the amount of rays reaching $N_{near_{max}}$ stays the same and that this function has to iterate over every value, a linear increase based on $N_{near_{max}}$ is within expectations. Furthermore, there is a correlation between G_{sim} and $N_{near_{max}}$. An increase in $N_{near_{max}}$ seems to cause

a minor increase in G_{sim} . The majority of the work performed during G_{sim} is composed of ray tracing the visibility rays and performing the nearest neighbour search. The time these tasks take can be calculated by setting $N_{near_{max}}$ equal to 1. With $N_{near_{max}}$ set to 1, the time G_{sim} took was $62.59ms$. If this baseline time is subtracted from the other tests, a linear correlation does appear for $N_{near_{max}} \geq 250$. Overall the impact of $N_{near_{max}}$ on the execution time is minor. For interactive applications a value between $[50, 250]$ seems advisable as it provides the most amount of samples for very little performance cost.

3.6.5 Performance comparison with i-Simpa

Next to an accuracy comparison with i-Simpa we can also perform a performance comparison. As explained in the previous sections, increasing the sonel amount has the biggest impact on performance. The setup used for these tests can be found in section 3.6.2. All parameters, other than the amount of sonels initially emitted, will stay the same during these tests. The parameters that can be matched between i-Simpa and the sonel mapping implementation will be set to the same value. A set of most relevant metrics have been collected for this data set.

- N_{sonel} : The amount of sonels emitted from the sound source. For i-Simpa this is configured as "particles per sound source".
- I_{calc} : This is a timing value provided by the log of i-Simpa. What part of the process is part of the provided "calculation time" is not explained but it seems to represent the time spent ray tracing particles.
- I_{total} : This is the total time spent calculating the echogram by i-Simpa. After the ray tracing part it appears $600ms$ is needed to parse the results.
- T_{sim} : This is the time spent on the GPU to ray-trace the sonel tracing step of the sonel mapping implementation.
- G_{sim} : This is the time spent on the GPU to ray-trace the sonel gathering step of the sonel mapping implementation.
- S_{total} : This is the time spent by the sonel mapping implementation in total. The time it takes to load in the 3D model of the scene is not included as it is not included in I_{total} either.

N_{sonels}	$I_{calc}(ms)$	$I_{total}(ms)$	$T_{sim}(ms)$	$G_{sim}(ms)$	$S_{total}(ms)$
50000	218	849	0.56	6.66	272.07
100000	358	1000	0.88	13.17	535.88
250000	639	1307	1.72	31.18	1298.80
500000	956	1571	4.03	60.94	2575.03
1000000	1043	1675	6.54	120.49	5188.12
2500000	1957	2612	16.06	282.79	13981.10

Taking a look at the I_{total} we can see that the implementation of i-Simpa scales well with an increase to N_{sonels} . This is reflected when the value of I_{total} is compared to S_{total} as can be seen in graph 3.16. The total execution times of both applications scale linearly with the sonel mapping implementation starting out faster. However, because the implementation of i-Simpa scales better we can see that the performance of i-Simpa catches up to the sonel mapping implementation around $N_{sonels} = 250\,000$. The early advantage of the sonel mapping implementation seems to be due to a static overhead in the i-Simpa implementation. As mentioned earlier, the implementation of i-Simpa seems to add $600ms$ after the ray-tracer has finished.

The sonel mapping implementation makes use of the parallelism of a GPU to simulate multiple rays at the same time. However, a significant overhead is present in the sonel mapping implementation. The most significant of which being transferring data between the CPU and GPU and constructing the sonel map for use on the GPU. Both of these actions are costly and also increase linearly which results in the sonel mapping implementation being slower than i-Simpa overall. However, when comparing the ray tracing parts of both programs there is a significant improvement to the execution times in favour of the sonel mapping implementation. As can be seen in graph 3.17, both implementations scale linearly but the sonel mapping ray tracing is significantly faster and scales better. Looking at the table of data above, we can even see that the gathering step G_{sim} reaching interactive timings for $N_{sonels} < 250\,000$.

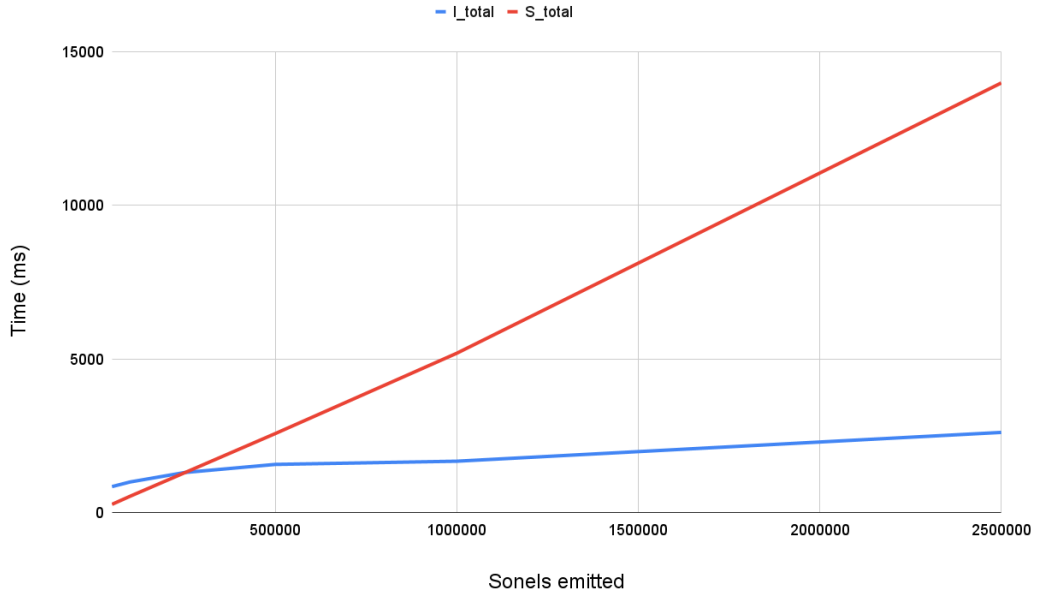


Figure 3.16: A graph plotting the total execution times of i-Simpa and the sonel mapping implementation against the amount of sonels emitted. With I_{total} being the total execution time of i-Simpa and S_{total} being the total execution time of the sonel mapping implementation.

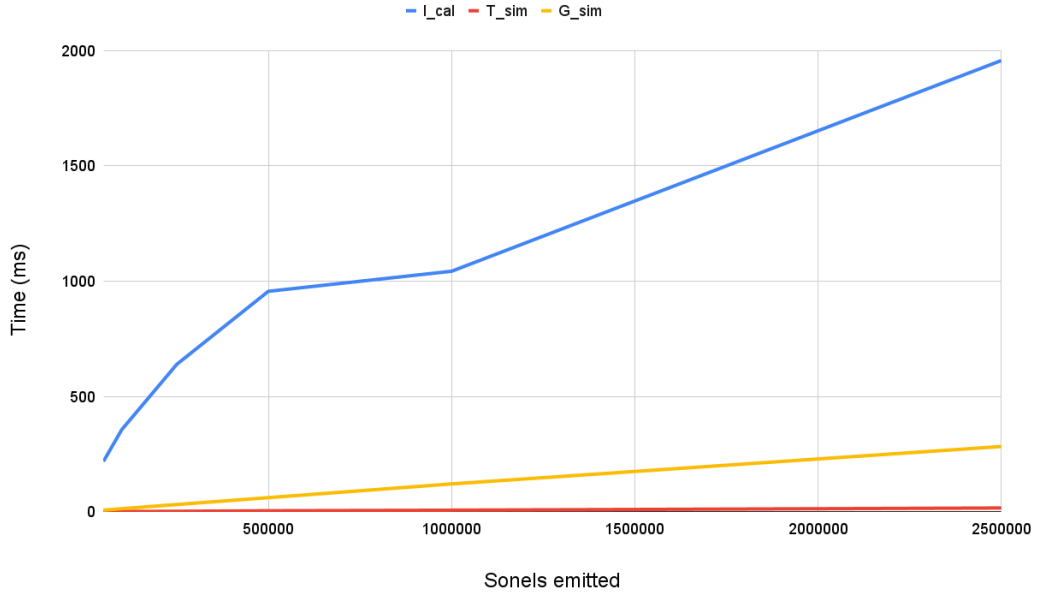


Figure 3.17: A graph plotting the ray tracing execution times of i-Simpa and the sonel mapping implementation against the amount of sonels emitted. With I_{calc} being the execution time of the calculation stage of i-Simpa. While T_{sim} and G_{sim} are the execution times of the tracing step and gathering step on the GPU of the sonel mapping implementation respectively.

Chapter 4

Conclusion

From the echogram analysis performed in section 3.6.3 we can conclude that the implementation of the sonel mapping does not converge towards a solution. Because the solution does not converge it also means it is not converging towards the correct solution. However, the shape and decay of the echogram does seem to correspond with the echogram of i-Simpa. From the analysis that was performed and by looking at similar algorithms such as SPPS[9] and photon mapping[8], it seems to be an issue with the gathering step. These algorithms scale the energy of particles that were emitted to ensure that the estimate converges to a solution. A similar scaling appears to be lacking in the sonel mapping algorithm. As future work, inspiration can be taken from the aforementioned algorithms to ensure the solution of the sonel mapping algorithm converges towards a correct solution. This is an unfortunate result but the performance goals of the thesis can still be achieved.

In the data provided in section 3.6.4 the total execution time of the implementation exceeds the interactive time requirements of 16-33ms. However, the gathering step and construction of the echogram is shown to be less than 33ms for $N_{sonel} < 250\,000$ which does meet the interactive timing requirement. The benefit of the sonel map approach is that the tracing and gathering steps can be separated. If the sound sources are known in advance and do not change dynamically, for instance during a cutscene or scripted sequence. The tracing step can be calculated beforehand and the resulting sonel map can be stored for later use. Currently, the biggest bottleneck of the sonel mapping implementation is the construction of SBT hit-records which does happen after the construction of the sonel map. However, these records only have to be constructed once for each sonel map that was loaded into memory. This can be done during the start of the interactive software or can be done in the background before a simulated sound source is reached. Afterwards the reconstruction of sound energy levels can be done in less than 33ms. Seeing as the location of the receiver is only needed during the gathering step, and the gathering step can be executed at interactive speeds, it possible to use the sonel mapping technique for interactive purposes.

Continuing with performance, the ray-traced parts of the sonel mapping implementation are significantly faster than the ray tracing parts of i-Simpa. However, the overall performance is hampered by GPU memory management and hit-record construction. This is apparent in the data in section 3.6.4. On average, 10% of the execution time is spent on downloading the sonel data from the GPU, 27% of execution time is spent on creating the acceleration structure for the sonel map and 58% is spent on creating hit records for each sonel. These 3 items account for 95% of the total execution time. This was an unexpected result during the analysis as effort was put in optimising other parts of the implementation. This is another example of optimising before profiling an implementation. It is advisable for future implementations to properly profile the application before adding optimisations.

For future implementations it can also be interesting to explore the use of alternative nearest neighbour search methods. The method used in this implementation formulates the nearest neighbour search as a part of the ray tracing process. The technique has been shown to outperform other GPU nearest neighbour applications[11] but the preparation cost of this approach might outweigh the benefits. As previously mentioned, constructing the SBT hit-records for the gathering step takes 58% of execution time. If the construction of these hit-records can be avoided it would result in a significant increase in performance. The benefit of utilising this nearest neighbour search technique needs to be weighed against the cost of preparing all the SBT hit-records. After such an evaluation, the selection of nearest neighbour

search can be selected depending on the use-case.

If the nearest neighbour search algorithm stays the same there is another optimisations that can cut down on the time required to construct the acceleration structure. Currently, the construction of the AABB for the sonels is executed purely on the CPU. However, this is a task that can be solved in parallel. Before downloading the sonels from the GPU, the GPU could be instructed to construct the AABBs of the sonels.

Furthermore, the memory usage of the application is significant. The amount of available video memory is usually less than the amount of available CPU memory therefore, it is valuable to optimise the memory consumption of the implementation. Furthermore, the tracing step in the implementation is executed in parts on the GPU to save on the amount of memory required. However, for the gathering step, each sonel needs to be present on the GPU. This puts an upper limit on the amount of sonels that can be simulated. Inspiration can be taken from techniques such as progressive photon mapping to eliminate this limit. The progressive photon mapping technique has eliminated the need for the whole photon map to be present during the gathering step, because of this it is well suited as a solution to the memory problem.

Additionally, the original sonel mapping algorithm had support for diffraction. This phenomena was not implemented for this thesis. For future work, adding diffraction support to the GPU implementation of sonel mapping would be an interesting avenue to explore.

Finally, with writing and implementing this thesis I have learned a lot about GPU ray tracing, OptiX, CUDA and about geometric acoustics. GPU ray tracing went from something I knew about to something I am familiar with. A lot of struggles went into understanding OptiX but by the end I was comfortable working with it.

Chapter 5

Appendix

5.1 Code

Listing 5.1: Psuedo-code of my implementation of the ray generation program during the sonel tracing stage.

```
--raygen--tracing() {
    sonelIndex = optixLaunchIndex.x
    decibelIndex = optixLaunchIndex.y

    soundFrequency = globalParameters.soundFrequency

    decibelStride = soundFrequency.decibelSize *
                    soundFrequency.sonelMaxDepth *
                    soundFrequency.sonelAmount

    rayIndex = decibelStride * decibelIndex +
                sonelIndex * soundFrequency.sonelMaxDepth

    decibels = soundFrequency.decibels[decibelIndex]

    rayData = TracingRayData()
    rayData.random = CudaRandom(rayIndex)
    rayData.index = rayIndex
    rayData.distance = 0
    rayData.timeOffset = decibelIndex * globalParameters.timestep

    // It should be 10e-12 but this can cause precision issues.
    rayData.energy = ((powf(10.0f, decibels / 10.0f) /
                      (soundFrequency.sonelAmount * 1.0))) * 10e-6;

    pointer1 = 0
    pointer2 = 0
    packPointer(pointer1, pointer2, &rayData)

    rayDirection = prd.random.generatePointOnSphere()
    optixTrace(
        globalParameters.soundSourcePosition,
        rayDirection,
        pointer1,
        pointer2
    )
}
```

Listing 5.2: Psuedo-code of my implementation of the closest-hit program during the sonel tracing stage.

```
--closesthit__tracing() {
    rayDirection = optixGetWorldRayDirection()
    rayOrigin = optixGetWorldRayOrigin()
    rayData = unpackOptixPointers()

    u = optixGetTriangleBarycentrics().x
    v = optixGetTriangleBarycentrics().y

    vertexIndex = indices[optixGetPrimitiveIndex()]

    hitPosition = (1.f - u - v) * vertex[index.x]
                  + u * vertex[index.y]
                  + v * vertex[index.z]

    surfaceNormal = normal
    if (surfaceNormal == nullptr) {
        surfaceNormal = cross(
            vertex[index.y] - vertex[index.x],
            vertex[index.z] - vertex[index.x]
        )
    }

    sonel = globalParams.sonelBuffer[rayData.index]

    randomNumber = rayData.cudaRandom.uniformFloat()
    // Check if sonel is absorbed
    if (randomNumber > (diffuseProb + specularProb)
        || rayData.depth + 1 == globalParams.maxDepth) {
        sonel.frequency = 0
        return
    }

    rayDistance = length(hitPosition - rayOrigin)
    rayData.distance += rayDistance
    rayData.depth += 1
    rayData.index += 1

    newDirection = vector3f()
    // Check if sonel is reflected diffusely
    if (randomNumber <= diffuseProb) {
        sonel.position = hitPosition
        sonel.incidence = rayDirection
        sonel.energy = rayData.energy
        sonel.time = rayData.distance / globalParams.soundSpeed + rayData.timeOffset
        sonel.distance = rayData.distance

        sonel.frequency = globalParams.soundFrequency.frequency
        sonel.frequencyIndex = globalParams.frequencyIndex

        newDirection = rayData.cudaRandom.uniformVec3fHemisphere(surfaceNormal)
    }
    // Sonel is reflected specularly
    else {
        newDirection = 2 * dot(rayDirection, shadingNormal)
                       * surfaceNormal
    }
}
```

```
                                - rayDirection
    }

    pointer1 = 0
    pointer2 = 0
    packPointer(pointer1, pointer2, &rayData)

    optixTrace(
        hitPosition,
        newDirection,
        pointer1,
        pointer2
    )
}
```

Bibliography

- [1] James T. Kajiya. “The rendering equation”. In: *ACM SIGGRAPH Computer Graphics*. Vol. 20. 4. New York, NY, USA: Association for Computing Machinery, Aug. 1986, pp. 143–150. DOI: 10.1145/15922.15902.
- [2] H. E. Bass, L. C. Sutherland, and A. J. Zuckerwar. “Atmospheric absorption of sound: Update”. In: *Acoustical Society of America Journal* 88.4 (Oct. 1990), pp. 2019–2021. ISSN: 0001-4966. DOI: 10.1121/1.400176.
- [3] Martin Bertram et al. *Phonon Tracing for Auralization and Visualization of Sound*. Vol. 0. Oct. 2005. ISBN: 978-0-7803-9462. DOI: 10.1109/VIS.2005.78.
- [4] B. Kapralos, M. Jenkin, and E. Milios. “Sonel Mapping: A Stochastic Acoustical Modeling System”. In: *Acoustics, Speech, and Signal Processing, 1988. ICASSP-88., 1988 International Conference on* 5 (June 2006), p. V. ISSN: 1520-6149. DOI: 10.1109/ICASSP.2006.1661302.
- [5] B. Fabianowski and J. Dingliana. “Interactive Global Photon Mapping”. In: *Comput. Graphics Forum* 28.4 (June 2009), pp. 1151–1159. ISSN: 0167-7055. DOI: 10.1111/j.1467-8659.2009.01492.x.
- [6] Toshiya Hachisuka and Henrik Wann Jensen. “Stochastic progressive photon mapping”. In: *SIGGRAPH Asia ’09: ACM SIGGRAPH Asia 2009 papers*. New York, NY, USA: Association for Computing Machinery, Dec. 2009, pp. 1–8. ISBN: 978-1-60558858-2. DOI: 10.1145/1661412.1618487.
- [7] Toshiya Hachisuka and Henrik Wann Jensen. “Parallel progressive photon mapping on GPUs”. In: *SA ’10: ACM SIGGRAPH ASIA 2010 Sketches*. New York, NY, USA: Association for Computing Machinery, Dec. 2010, p. 1. ISBN: 978-1-45030523-5. DOI: 10.1145/1899950.1900004.
- [8] Henrik Wann Jensen. “Global Illumination using Photon Maps”. In: *Rendering Techniques ’96*. Wien, Austria: Springer, Vienna, Dec. 2011, pp. 21–30. DOI: 10.1007/978-3-7091-7484-5_3.
- [9] Judicaël Picaut and Nicolas Fortin. “SPPS, a particle-tracing numerical code for indoor and outdoor sound propagation prediction”. In: *ResearchGate* (Apr. 2012). URL: https://www.researchgate.net/publication/278806110_SPPS_a_particle-tracing_numerical_code_for_indoor_and_outdoor_sound_propagation_prediction.
- [10] Iordanis Evangelou et al. “Fast Radius Search Exploiting Ray-Tracing Frameworks, JCGT”. In: *ResearchGate* 10.1 (Feb. 2021), p. 2021. URL: https://www.researchgate.net/publication/349551320_Fast_Radius_Search_Exploiting_Ray-Tracing_Frameworks_JCGT.
- [11] Yuhao Zhu. “RTNN: Accelerating Neighbor Search Using Hardware Ray Tracing”. In: *arXiv* (Jan. 2022). DOI: 10.48550/arXiv.2201.01366. eprint: 2201.01366.