Sondre Slåttedal Havellen

Acoustic Simulation in 2D using CUDA

Master's thesis in Computer Science
Supervisor: Magnus Lie Hetland
July 2019

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Kunnskap for en bedre verden

Sondre Slåttedal Havellen

Acoustic Simulation in 2D using CUDA

Master's thesis in Computer Science
Supervisor: Magnus Lie Hetland
July 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Abstract

This thesis explores how the GPU can be employed in order to simulate acoustical reverberation in virtual spaces. This thesis presents mathematical theory of acoustics, several FDTD methods as well as implementation and evaluation of them. Two applications are implemented; one for simulating acoustical waves on the GPU using CUDA and one VST for convolving the resulting impulse responses from the simulator with an incoming audio signal. Experimental results show that this approach to a physically based VST plug-in might be viable, although artifacts such as numerical dispersion and low computational efficiency might prevent it from reaching its potential.

Sammendrag

Denne oppgaven undersøker hvordan GPU kan brukes for å simulere akustikk i virtuelle rom. Denne oppgaven presenterer matematisk teori om akustikk, flere FDTD-metoder, samt implementering og evaluering av metodene. Videre er to applikasjoner implementert; en for å simulere akustiske bølger på GPU ved hjelp av CUDA og en VST for å anvende de resulterende impuls responsene fra simulatoren med et innkommende lydsignal. Eksperimentelle resultater viser at denne tilnærmingen til en fysisk basert VST-plugin kan ha potensiale, selv om artefakter som numerisk spredning (numerical dispersion) og lav ytelse fortsatt forhindrer praktisk bruk av applikasjonen.

Acknowledgements

While the writing of this thesis has been hard, and at times demotivating but in the end rewarding, this thesis could not have been possible without the help of my advisor Magnus Lie Hetland. So thanks to him! Further, thanks to my father Vidar Havellen, mother Sissel Slåttedal and brother Vegard Slåttedal Havellen for proof reading and motivation. A big thanks to the open source projects that have contributed to this thesis as well.

Contents

List of Figures

1 Introduction

This master thesis is dealing with the theory, design and implementation of an application for simulating audio reverberation and propagation in a virtual space. More generally, I will explore how the GPU can be employed for acceleration of audio application software. In the end I hope to have a usable software application for simulating realistic acoustical effects in virtual spaces.

In short the software will consist of a simulation part where the user should be able to draw geometry on an image. Ideally, the user should be able to move around in a 2D space and experience different reverberation effects depending on the position of both the source and the listener. The implementation is partly based on [1] and [2], and several ways of simulating and using the results will be explored. Specifically, my implementation is implemented with NVIDIA CUDA for acceleration on the GPU using a Finite Difference Time Domain method. After the user has drawn geometry on a canvas, it should be possible to run a simulation from a given audio file and generate reverberated sound as output.

Although convincing audio reverberation effects can be made using existing algorithms and software, there does not exist any publicly available software for simulating virtual geometry effectively on the GPU, at least not any consumer product for audio production as far as I know. Exploring how the geometry of a virtual space affects an output sound is not something I have seen in any audio production applications, or more specifically VSTs, which are applications for applying sound effects to incoming audio. Extensive research for visual computing and rendering has been done in relation to for example video games, but similar research for digital audio simulation is less extensive. So the end goal for this project is to develop an application for doing such simulation, and hopefully also an effect plug-in (VST) for usage in Digital Audio Workstations. Ideally these two parts should be one and the same, however there are a lot of different technologies that have to work together here, so this thesis will primarily focus on the simulation part.

A large part of this thesis will explore different numerical simulation methods and evaluate them against each other. Effects like diffraction (sound bends around objects), diffusion (decay in time) and reflection (echo) will be explored. Artifacts such as numerical dispersion, geometry constraints, unwanted reflections and time stepping constraints / numerical stability will be discussed. This project takes some inspiration from acoustical echo chambers where hollow enclosures was used to produce audio reverberations. The idea is to be able to design such rooms in an application. As we will see in the result section, the actual results were less than ideal and more work will have to be done in order for this to be an actual usable reverberation simulation application.

A few research questions will be explored in this thesis as well, most notably

- What exists in terms of physically accurate acoustic simulators (ie. software) aimed at audio engineering?
- What algorithms exists for generating reverberations in an audio signal?
- Given low latency requirements for real-time audio effects, could the GPU be utilized for real-time physically accurate acoustical effects?

1.1 Approach

There exists several ways to simulate acoustics in a space, several of which will be explored in this thesis. Most common is the Finite Difference Time Domain methods. In this thesis I will start with partial differential equations describing the physics of sound propagation, derive numerical integration schemes and then implement said methods and evaluate them against each other. Common to all of the methods tested here is that they use a FDTD scheme. As a result of such a simulation it is possible to capture an impulse response. By using short term convolutions one can create realistic sounding acoustical effects by convolving an input signal with said impulse response. Thus, a simple VST based on mathematical convolutions is implemented for testing real-time effect in audio applications. An attempt to combine the simulation part and the convolver part is made, but the main focus in this project is on the simulation. Pros and cons of the different FDTD methods will be discussed and evaluated. In the implementation part I will be implementing several of the methods in Python for evaluation and then port them to C++ and CUDA for maximum performance. I will also be discussing how this problem can be extended into three dimensions.

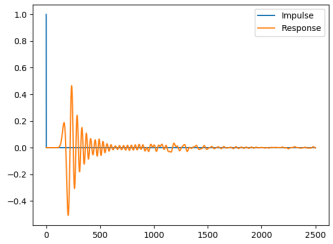

Figure 1: An impulse and it's response.

Convolutional reverberation effects work by applying an impulse response to an incoming signal to produce the illusion that the signal has been recorded in (for example) a large hall. This works because acoustics of spaces can be described by linear time-invariant systems, and as such they can be described by an impulse given a brief input signal, called an impulse (as shown in figure 45). If we apply such an impulse to our system, we can capture the response as this energy travels and reflects and diffracts around objects in a space. Using this captured signal we can then convolve a given source sound with this impulse response and get an accurate representation of the actual room acoustics for this scene. This is the approach taken by [1]. However, problems present themself in terms if the input

parameters (position of source and destination etc.). If one were to use this approach one would have to simulate each source-to-destination pair for the enitire room, requiring $O(t(m * n)^2)$ timesteps and space requirements in the naive case (m and n being width and height while t is the number of timesteps). The upshot is great performance for the real-time processing part, but a huge memory usage. Essentially, we will have to restrict our simulation to static source and destination positions.

There is also possible to plug a source sound directly into the simulation and then pick up the result directly at another location in the virtual space. Advantages include processing time and memory requirement decreases (we only have to simulate one source-to-destination) and a simpler implementation (no need for convolutions). However, large requirements in regards to latency and performance might become hard to satisfy. As this type of simulations is largely only feasible on the GPU, steps has to be taken to avoid memory latency when transferring data from and to the GPU. Audio processing has some fairly limiting requirements in respect to latency. Further, the simulation efficiency has to be sufficient. For example a common sampling frequency used in audio applications is 44100 samples per seconds, as humans are only really able to hear frequencies up to 22kHz (requiring the double of this to store in memory due to the Nyquist sampling theorem). Thus we need simulation steps of $\frac{1.0}{44.1kHz} \approx 2.27 * 10^{-5}$ seconds, and possibly even smaller depending on the stability constraints of the different numerical methods. Couple this with $m * n$ cells in the simulation and we quickly get huge requirements for efficiency. For example, a 256 * 256 simulation space will require about $\sim 2.89 * 10^9$ calculations in addition to the numbers of calculations involved in an iteration for a single cell. And this is not even accounting for the required spacial step size, which should be so small that the smallest wavelength could propagate around the virtual space. This smallest step size is constrained by $\lambda = \frac{c}{f}$ where c is the speed of sound, $343\frac{m}{s}$, and f is the maximum frequency, which in our case is 44.1 kHz. Plugging in these numbers we get a smallest step size of 7.7 mm. So for a 256 * 256 grid we are only able to simulate a $\sim 2m^2$ space. The original goal for this thesis was to be able to such simulation in real-time. However, it quickly became apparent that it was infeasible due to the simulation speed obtained and the numerical constraints hindered adjusting the time steps.

As stated, the application will consist of two parts. A simulation part and a real-time processing part, which contain both a GUI and audio processing. The simulation part consists of editing a 2D scene in the form of drawing walls and placing sources and destinations on an image. After that a simulation will be run from a source position to a destination position. The real-time processing part consists of a VST plugin [3] made with the JUCE framework [4]. By the end, work will be started combining these two parts, and as we shall see the way the different technologies work together can be quite cumbersome.

1.2 Motivation

The idea for doing this project came from the fact that a lot of audio application software is not utilizing the GPU to it's fullest, at least not using it for physical simulation. In fact, physical simulations in VSTs is a rare sight and I wanted to explore ways to generate physically based effects based on real physical laws. Several different ideas was explored; simulating strings and capture audio from the resulting vibrations, simulating wind instruments as seen in [2], generating wave-tables for morphing in a wave-table synthesizer, but I finally arrived at simulating acoustical waves as this seemed more realizable. My initial goal was to implement a system that could simulate wave propagation in real time using the GPU as a hardware accelerator. Using the GPU directly for a real-time task like acoustical simulation proved impossible, but convolutions seemed to work well. Whether or not I was successful in the final goal is debatable, as detailed in section 6. I ended up with a fairly good convolutional reverberation VST (as the resulting audio sounded was good), but the simulator was not really usable in a real scenario.

1.3 Code repositories

The code ended up being a little scattered throughout several repositories. This is because a lot of the development ended up following a trial-and-error approach. In the end a final repository was created and the code can be found here: `https://github.com/sondrehav/master`.

- **ConvolverVST**: The final VST responsible for applying an impulse response to an input signal.
- **SimulatorApplication**: The final simulator application responsible for generating impulse responses.
- **EchoSimPython**: The Python code used for testing the different methods.
- **Old CUDA VST**: One of the early VST implementations.
- **Old OpenCL VST**: One of the early VST implementations.

2 Background

Today, computers has become one of the main tools for any audio engineers. Virtually all newer music has been made using computers and modern recording techniques. Audio production software have largely replaced large analog gear such as mixing boards and effect units, and nearly anyone with access to a reasonable powerful computer can produce music and audio. While computers are good at recreating effects and tools based on electronics, physical instruments is a much different story. Electronic circuits contains a relatively small domain, while real physical instruments require simulating a whole lot of real-world phenomenons and interactions. This chapter contains some background on what the context of this problem is, a few of the central concepts (ie. Digital Audio Workstations, plug-ins etc.) and a description of some similar projects.

2.1 History and a brief introduction to audio production software

In this section I will briefly give some background on how we are where we are today in terms of audio processing software and describe some of the common tools used in music production and audio engineering. As one of the end-goals of this project is to make a VST plug-in it is necessary to give some background on what they are and in what context they are used.

2.1.1 Effect units and synthesizers

An effect unit is an electronic device that alters the incoming sound in some way to produce a different output. In the context of guitars these units often somes in the form of small pedals, while in the mixing process these units can be incorporated into mixing boards. Very early effect units were only really practical in studios. It was in around 1940 recording engineers and musicians began experimenting with manipulation of reel-to-reel tape recordings to create echo effects and other unusual sounds. The first commercially available stand-alone effect unit, called Trem-Trol, was released by DeArmond in 1948, and produced a tremolo effect. It wasn't until the late 1960s the stand-alone effect units became popular, as the previous units often were large, impractical and bulky, requiring large transformers and high voltages. Common effects units include distortion, dynamic effects such as volume pedals and compressors, filters, pitch effects, modulation effects such as chorus and phasers and time effects such as delay and reverbs. Effect units like these could be implemented in a small box, taking one input source and producing an altered output sound. Today, these units are often the inspiration for audio plug-ins implemented in software.

Echo chambers is hollow spaces specifically designed to create acoustical reverbrations, often for recording purposes. Echo chambers could be employed by playing a recording of a sound, for example a conversation, and then capturing the recording with a microphone to record the reverbrations. These enclosures have

largely been replaced by effect units, but one interesting technique for making real enclosures like these practical today is the usage of captured impulse responses and then applying these IRs to sounds using a process involving convolutions. This is the ideal way this project can be implemented in a effect plug-in.

2.1.2 Sound recording and Digital Audio Workstations

Digital Audio Workstations, or commonly DAWs, is software for editing, recording and producing music and audio. DAWs are applicable to any situation where complex audio and editing is required, such as music, speech, television, soundtracks, podcasts, radio and sound effects etc. There exists a wide variety of Digital Audio Workstations ranging in both price and complexity. Some of the most notable commercial DAWs includes FL Studio, Logic Pro, Ableton and Cubase.

Before DAWs were the standard, audio engineers were limited to analog recording onto tape recorders and mixing using large mixing boards. This was the standard up until the 1970. Some argue that this process sounded better due to the sound of recording to a tape. But this had it's limitations due to artifacts such as noise and distortion. Further it was not possible to automate sliders (ie. the gain knob on a distortion pedal or the filter cutoff for a lowpass filter). All this had to be done manually, limiting the creative freedom and essentially requiring help from other people when "automating" several parameters (gain sliders, faders etc.). Computers made it possible to automatically move any slider without the help of humans, which greatly increases the creative freedom for musicians. When computers made it possible to incorporate the entire process into one application, this became the new standard. Today, most music is produced using digital audio workstations, as opposed to analog mixing boards and recording tapes.

2.1.3 Plug-ins, VSTs, AU, RTAS

DAWs is powered by a large market for plug-ins which is small applications, sometimes standalone but mostly able to run inside a DAW. For this to be possible, the applications has to implement a common interface. There exists many such interfaces, where the most common ones are listed below. Audio plug-in software interfaces allows different DAWs to host any plug-in implementing these interfaces, which has allowed for a rich and vast market of software effects and synthesizers. These audio interfaces enables plugins to process, generate, recieve and manipulate streams of audio in near-realtime with as little as possible latency. Common interface features include audio stream input and output, MIDI input and output, parameters (knobs, sliders, values etc.) and GUI features.

- **Steinberg's Virtual Studio Technology** (VST) was released in 1996 and is today one of the most commonly used audio plug-in software interfaces, supporting a wide variety of DAWs. A VST plug-in can either be an effect or an instrument. Effects receive digital audio from the host and process it through to their outputs. Instruments receive MIDI input from the host and generated sounds that the plug-in passes on back to the host.

- **Apple's Audio Unit** (AU) is the proprietary system-level audio interface for Apple computers, and is fairly similar to VST.
- **AAX** is a format for audio-plugins for Pro Tools LE, developed by Avid Audio. It is the replacement for another popular interface called Real-Time AudioSuite (RTAS) which is still somewhat used today.

All these different format is somewhat similar to having to developing for different platforms like Linux and Windows, and thus people have begun writing wrappers for these interfaces. The JUCE framework is an example of one such wrapper, which is able to generate ports to any of these formats.

2.1.4 General-purpose Computing on Graphics Processing Units

GPGPU is the use of a graphic processing unit for performing computation on tasks historically performed by CPUs. GPUs offers greater throughput than a CPU which typically favors a single instruction at a time. In general, GPUs can perform a single instruction on multiple data in parallel at a lower clock frequency, as they contain many more cores compared to a CPU.

General purpose GPU-computing became practical and popular around 2001 when programmable shaders and floating point support on graphic processors were introduced. The first GPU with programmable pixel shader was the Nvidia GeForce 3 (NV20) released in 2000, and as the power of shaders quickly became apparent vertex shaders was also introduced. Programmable pixel shaders in particular allowed developers to write shaders to perform some computation on the GPU, then write back the data. It was thus possible to run general computations by "hacking" APIs suck as OpenGL and DirectX.

In 2007 Nvidia released the parallel computing platform and API CUDA. It allows for developers to use CUDA-enabled GPUs for general purpose processing. CUDA is a software layer giving direct access to the instruction set of the GPU. It allows the programmer to write C/C++ code with a few additions, making it relatively easy to use. However, extensive knowledge of the CUDA hardware architecture is required in order to write efficient code. Another computing API targeting similar usecases as CUDA is OpenCL. It was first released in 2009, and it is the main competiting API for writing general purpose computations on the GPU. In contrast to CUDA, which only supports CUDA-enabled GPUs, OpenCL is designed to be executed across any heterogeneous platforms consisting of CPUs, GPUs, DSPs, FPGAs and other hardware accelerators. OpenCL was originally developed by Apple (which still holds trademark rights), but is today maintained by the non-profit technology consortium Khronos Group, which also maintains OpenGL among other standards.

General purpose graphic-card computing enables a wide variety of applications, ranging from physic simulations, machine learning, video processing, medical imaging, computer vision and many other important tasks. GPUs however do have some

drawbacks. GPUs are generally optimized for throughput and not latency (as opposed to CPUs), which can make some real-time tasks with low latency requirements harder or infeasible. Real-time audio processing for example may require a certain delay in the audio stream as memory transfers from and to the GPU can be slow. GPUs are generally suited for tasks where you would require the same operation for a large amount of data. The equivalent code on a CPU, compared to a GPU, would most likely involve a loop which iterates over several data elements and executes some instruction. While CPUs are optimized for arbitrary computation, and importantly branching, GPUs can't really handle branching very well. This is because the GPU units executes the same instructions in lockstep and if any of the computing units have to branch (for example a divergent `if`-statement), the other units essentially have to wait for that divergent unit to complete. This is important to keep in mind when writing efficient code on both CUDA, OpenCL and OpenGL.

2.2 Similar projects

Below some articles and projects are listed which this project takes inspiration from. One fairly large problem encountered in this project is that some of the most advanced VST projects are products from commercial companies. This implies thatDSP methods and algorithms employed for a given plug-in can be hard or impossible to deduce. Such information is usually kept within a company and not openly available as research articles or publications. However, for this project the necessary tools was readily available on the web (ie. a convolutional audio signal method).

2.2.1 The NESS Group

The Next Generation Sound Synthesis project was an exploratory project based in the University of Edinburgh. It was a join project between the Acoustics and Audio Group and the Edinburgh Parallel Computing Centre. The project is concerned with synthetic sound and numerical simulation techniques for physically accurate sound synthesis. In particular, FDTD methods for simulating a set of instruments is explored and of particular importance is the implementation on parallel hardware such as GPUs for maximum performance. Of particular importance to this thesis is the 2018 paper **Higher-order Accurate Two-step Finite Difference Schemes for the Many-dimensional Wave Equation** [5] and the 2014 paper **Hexagonal vs. rectilinear grids for explicit finite difference schemes for the two-dimensional wave equation** [6].

2.2.2 Aerophones in Flatland - Article

This article and project concerns the simulation of 2D virtual wind instruments. It includes interactive geometry modification, full bandwidth sounds (ie. able to simulate sounds up to a prescribed frequency of 128 kHz) and runs in real-time. The key challenge is simulating geometric features in the range of a few millimeters

and microseconds, this requiring a extreme amount of resources. Other challenges include the dynamic modification of geometry, as hard editing of the geometry during simulation can result in clicking artifacts in the final sound.

2.2.3 Efficient and Accurate Sound Propagation Using Adaptive Rectangular Decomposition

This paper concerns physically accurate offline simulation of impulses in a 3D environment and a run-time environment capable of interpolating the different impulse responses. The system exploits the analytic solution of the wave equation on a rectangular grid, and is capable of simulating million cells with few artifacts. Key challenges include the decomposition of computational domain into disjoint rectangular regions suitable for analytic simulation, interface handling and the amount of data generated from such simulation. In regular FDTD the simulation has to be carried out for every point-to-point source and destination and the results has to be stored in memory, requiring space upwards of a few gigabytes. This is one of the main problems this article tries to solve. Originally, this paper first served as the main inspiration for this project, however the methods employed proved too difficult to replicate here. There was however some progress made when trying to solve this problem, which will be discussed here.

3 Theory

The partial differential equations (PDE) covering acoustical wave propagation looks relatively simple, as seen in equation (1). There is however a vast amount of theory, considerations, constraints and other concepts that is involved with this equation, and generally other partial differential equations. An entire field of study is devoted to acoustical engineering, and of particular importance is the electro acoustic branch which deals with sound reproduction and recording among other things. As this is an entire field of study, this thesis will only deal with a simplified domain, and modeling of the wave equation describing acoustical phenomenons will be of little importance here.

3.1 Mathematical modeling and framework

The wave equation can be described by

$$\frac{\partial^2 u}{\partial t^2} - c^2 \nabla^2 u = f, \tag{1}$$

where u is the pressure at a given location, t is time, c is the speed of propagation and f is the time dependent force acting on the system. ∇^2 is the Laplacian of the pressure field which means that the pressure is dependent on the curvature of the pressure field. c is defined as the speed of sound in air, ie. 343.0 m/s. The constant comes from mass density and elasticity of air. The derivation of the PDE is not the concern of this thesis, but can be found with more detail in [7].

It is also possible to formulate the wave equation as a system of PDEs. The following equations is the same system as above using several simultaneous PDEs. One for the pressure values and one for each velocity dimension. $\nabla\cdot$ is the divergence of the velocity field and ∇ is the gradient of the velocity field. These operators will be described in more detail below.

$$\frac{\partial u}{\partial t} = c\nabla \cdot \vec{\boldsymbol{v}} \tag{2}$$

$$\frac{\partial \vec{\boldsymbol{v}}}{\partial t} = c\nabla(u + f) \tag{3}$$

3.1.1 Solution

The wave equation has a well known analytic solution for rectangular grids. It is possible to exploit for acoustical simulation, but problems present them self when geometry is introduced. [1] used this as the basis for their simulator, where they solved the analytic equation using Fast Fourier Transforms at each time step and then exchanged the boundary values over an interface in the forcing term component. They did this by decomposing a scene into rectangular regions, running a simulation step over each region and then exchanging border values. This approach is fairly hard to implement, as I describe in the implementation section 5.

3.2 Numerical integration schemes and approaches

In order to solve the wave equation we set up a scheme for integrating the equation numerically. This is usually done by devising an approximation for the derivative terms in the equation. We start by looking at the definition of the derivative.

$$\frac{d}{dx}f(x) = \lim_{\delta \to 0} \frac{f(x+h) - f(x)}{\delta} \tag{4}$$

This definition arises from looking at an indefinitely small portion of a function, $f(x)$, and then finding the rate of change, or slope, for that x-value. We know that the second derivative simply is the derivative of the derivative. Plugging that in to (4) we get

$$\frac{d^2}{dx^2}f(x) = \lim_{\delta \to 0} \frac{\frac{d}{dx}f(x+\delta) - \frac{d}{dx}f(x)}{\delta} \tag{5}$$

$$= \lim_{\delta \to 0} \frac{f(x-\delta) - 2f(x) + f(x+\delta)}{\delta^2} \tag{6}$$

Since we want to discretize this definition we replace the limit by a small value, h, which will be our step size.

$$\frac{d^2}{dx^2}f(x) \approx \frac{f(x-h) - 2f(x) + f(x+h)}{h^2} \tag{7}$$

In the original wave equation we have the Laplace operator. It is defined as the divergence of the gradient, which will become more relevant later. For now the Laplace operator can be defined as the sum of all second partial derivatives in the Cartesian coordinates x_i.

$$\nabla^2 = \sum_{i=1}^{n} \frac{\partial^2}{\partial x_i^2}$$

As we are working in two dimensions for this project, I will be restricting the rest of the discussion to two dimensions. So we get

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$$

Using the function u which is the pressure over the space in (1) and discretizing yields

$$\nabla^2 u \approx \frac{u(x-h,y) + u(x+h,y) + u(x,y-h) + u(x,y+h) - 4u(x,y)}{h^2}$$

This is commonly known as a five-point stencil since we are using five points to determine the value for the given point, as seen in figure 3. We now have a discretization of the second term in (1).

3.2.1 Simple forward Euler

The time differential is discretized in the same manner as above, with a time step of k.

$$\frac{d^2}{dt^2}u(x,y,t) \approx \frac{u(x,y,t+k) - 2u(x,y,t) + u(x,y,t-k)}{k^2}$$

Writing the different variables in terms of indices can be useful from here on. That is we discretize $f(x,y)$ and u in terms of a grid with step size h, as seen in figure 2. Combining this with the original wave equation (1) and introducing the superscript t for the current time step, we get

$$\frac{1}{k^2}(u_{m,n}^{t+1} - 2u_{m,n}^{t} + u_{m,n}^{t-1}) - \frac{c^2}{h^2}(\nabla^2 u_{m,n}^{t}) = f_{m,n}^{t} \tag{8}$$

Rearranging terms in (8) yields a fairly simple explicit method.

$$u_{m,n}^{t+1} = (\frac{kc}{h})^2(\nabla^2 u_{m,n}^{t}) + f_{m,n}^{t} + 2u_{m,n}^{t} - u_{m,n}^{t-1} \tag{9}$$

This is now in a suitable form for implementation, but still has some large problems associated with stability. Instability occurs when errors in the derivatives accumulate over time. If the stability constraint of a numerical method is violated, the errors typically will increase exponentially resulting in floating point overflows on computers and unusable results. This constraint is called the Courant-Friedrichs-Lewy condition and is defined as

$$C = \frac{c\Delta t}{\Delta x} + \frac{c\Delta t}{\Delta y} = \frac{2ck}{h} \leq C_{max}$$

C is called the Courant number. For most explicit time stepping methods, $C_{max} = 1$. As humans typically can hear up to 22050 Hz the time step needs to be at least $\frac{1.0}{2*44100}$ seconds due to the Nyquist theorem. Further, if we want the geometry to affect all frequencies we also need a smallest step size of $h = \frac{c}{f_{max}} = \frac{343.0}{44100.0} = 7.78 * 10^{-3}$. The Courant number for this situation is then 2. So we will need a time step of $\frac{1}{44100*2}$ seconds in order to avoid numerical instability. This integration scheme is simple to implement, but not good in practice as it generates quite large numerical errors and equally simple (in terms of computation) but better schemes can be devised.

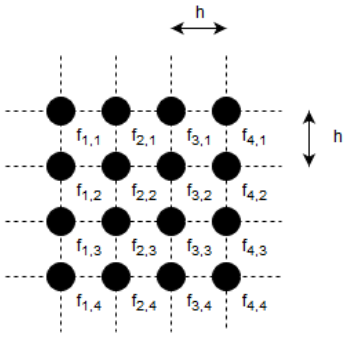

Figure 2: $f(x, y)$ mapped to a grid

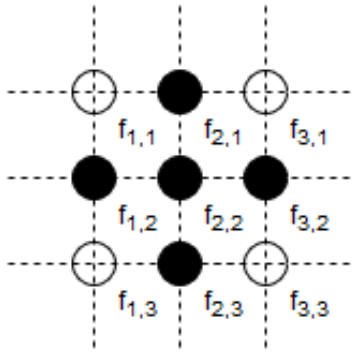

Figure 3: The five-point stencil

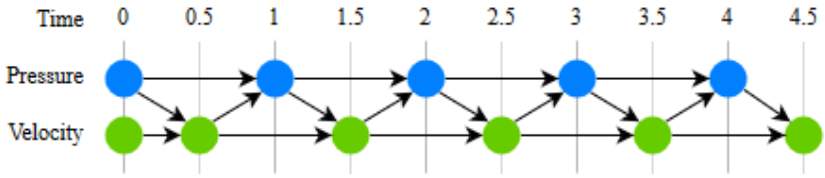

Figure 5: An illustration of a two-step leapfrog scheme

3.2.2 Two-step leapfrog

The slope of a chord between two points on a function (x_0, f_0) and (x_1, f_1) is a much better approximation of the derivative at the midpoint $f_{1/2}$, rather than at the endpoints as can be seen in figure 4. We can use this fact to devise a better integration method for the wave equation. This method is often called the Leapfrog method and works better in practice then the standard forward Euler method at virtually no extra computational cost. An illustration of how this method works compared to the standard forward Euler method can be seen in figure 5. We start with the wave equation (1) as a system of two PDEs as in (2) and (3).

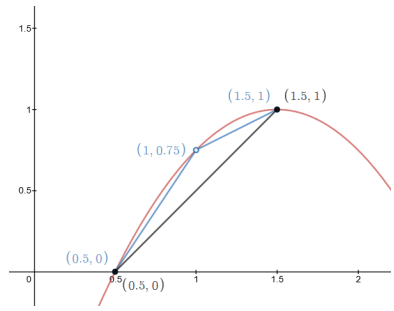

Figure 4: The derivative at $(1, f(1))$ is approximated better at the two points $(0.5, f(0.5))$ and $(1.5, f(1.5))$ compared with $(0.5, f(0.5))$ to $(1, f(1))$ or $(1, f(1))$ to $(1.5, f(1.5))$.

The forward Euler used above essentially approximates the wave equations by

$$u_{t+1} = u_t + \Delta_t v_t$$
$$v_{t+1} = v_t + \Delta_t f_t$$

where v is the velocity acting on a particle and f is the force acting on a particle (looking at the equivalent system of PDEs in (2) and (3)). As the midpoint of the function is a better approximation for the derivative, we can use that fact to devise a better integration scheme.

$$\begin{aligned} u_{t+1} &= u_t + \Delta_t v_{t+1/2} \\ v_{t+3/2} &= v_{t+1/2} + \Delta_t f_t \end{aligned} \tag{10}$$

In the case of the wave equation the force in (10) is dependent on the pressure as in (3). Looking at this scheme it is not immediately apparent how we get $v_{1/2}$. However, by utilizing the forward Euler for half a time step we can easily get this value. As the leapfrog method uses the midpoint rule as opposed to an endpoint, the error of this method grows with $\sim h^2$ over time. Thus he method is a second order approximation. However, more accurate methods could be devised, for example the widely used family of methods called Runge-Kutta.

3.3 Perfectly Matching Layers

In the discrete scheme above we have implicitly assumed perfectly reflecting walls along the boundaries. This is not realistic if we wish to simulate acoustics, as the sound waves will continually bounce of the walls forever. We will have to do some tricks in order to get rid of these reflections. A Perfectly Matched Layer is a solution to such a problem. It works by introducing an artificial layer around the area of interest, as in figure 6. PMLs was first formulated by Berenger in 1994 for use with Maxwell's equations for electromagnetism, but works just as well for the wave equation as they are quite similar. There are several different formulations such as split-field PML (which is the one we will use), uniaxial PML and stretched-coordinate PML. In short, whenever a spacial derivative occurs in an equation, we replace it with

$$\frac{\partial}{\partial x} \to \frac{1}{1 + \frac{i\sigma_x(x)}{w}} \frac{\partial}{\partial x} \tag{11}$$

$\sigma_x(x)$ can be defined in several ways. The actual definition isn't that important until we get to the implementation part. However the general idea is that absorption should gradually increase in the PML region. As such, if we define W as the width of the domain and p as the PML width, one definition could look like

$$\sigma_x(x) = \left\{ W + 2p > x \geq 0 : \max\left(\frac{-x}{p} + 1,\ 0,\ \frac{x - W - p}{p} \right) \right\}$$

We start by doing the transformation for one dimension, x, then the other, y. As we will see this can get quite messy, and we will have to introduce two extra partial differential equations into our system. Further, we will have to use the Fourier transform. The Fourier transform is a way of representing a function in terms of frequency components. Most importantly, differentiation and integration in the spacial domain reduces to just a complex multiplication in the frequency domain, as seen in (12). A complete description of Fourier transforms and complex numbers can be found in [8], and in this section it is only used symbolically. The only important aspect for our case is the fact that we can do a Fourier transform, differentiate or integrate (eq. (13)) and then transform back. No actual transformation has to be carried out.

$$\begin{aligned} \mathscr{F}\{f(x)\}(\omega) &= \hat{f}(\omega) \\ \mathscr{F}^{-1}\{\hat{f}(\omega)\}(x) &= f(x) \end{aligned} \tag{12}$$

$$\begin{aligned} \mathscr{F}\{\frac{\partial f(x)}{\partial x}\}(\omega) &= -i\omega\hat{f}(\omega) \\ \mathscr{F}\{\int f(x)dx\}(\omega) &= \frac{1}{-i\omega}\hat{f}(\omega) \end{aligned} \tag{13}$$

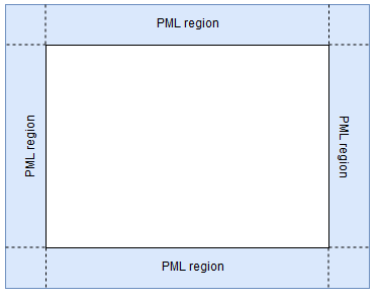

Figure 6: A Perfectly Matching Layer

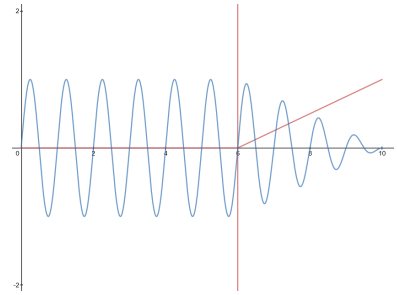

Figure 7: Gradual absorption

A PML works by exponentially dissipating energy along the PML region. In figure 7 we can see the σ_x function for a single direction in one dimension. Here we have a PML width of 4 and we see that the sinusoid is gradually decreased until it reaches zero. What we are essentially doing is evaluating an analytic function along along a deformed contour in the complex plane. We deform the axis to increase along the imaginary axis for $x \geq 6$. The solution of the equation e^{ikx} does not

change for $x \leq 6$ but is exponentially decaying for $x \geq 6$. In the context of the wave equation we have to analytically extend the solution in the PML region. All this essentially boils down to replacing the derivatives, as in (11), so I will not go into detail for that. Extra material can be found in [9].

We start with our system of equations from (2) and (3) and replace the x-derivative with the transformed derivative as in (11) and doing a Fourier transform over the time parameter. We have to do this wherever the differential operator appears. We get

$$\frac{\partial v_x}{\partial t} = c\frac{\partial(u+f)}{\partial x}$$

$$\Rightarrow -iwv_x = c\frac{1}{1+\frac{i\sigma_x(x)}{w}}\frac{\partial(u+f)}{\partial x}$$

Multiplying both sides with $1+\frac{i\sigma_x(x)}{w}$ and rearranging we get

$$-iwv_x = c\frac{\partial(u+f)}{\partial x} - v_x\sigma_x(x)$$

Now we'll do the inverse Fourier transform and get our first transformed PDE.

$$\frac{\partial v_x}{\partial t} = c\frac{\partial(u+f)}{\partial x} - v_x\sigma_x(x) \tag{14}$$

The velocity in the y-direction in this case is trivial and results in no change as the derivative for the x-direction does not appear. For the pressure equation we will have to do some extra tricks. We start with replacing the differential operator.

$$\frac{\partial}{\partial t}u = c\nabla\cdot\vec{\boldsymbol{v}} = c\cdot\left(\frac{\partial v_x}{\partial x}+\frac{\partial v_y}{\partial y}\right)$$

$$\Rightarrow -iwu = c\cdot\left(\frac{1}{1+\frac{i\sigma_x(x)}{w}}\frac{\partial v_x}{\partial x}+\frac{\partial v_y}{\partial y}\right)$$

Again, multiplying both sides with $1+\frac{i\sigma_x(x)}{w}$ and rearranging we get

$$-iwu = c\nabla\cdot\vec{\boldsymbol{v}} + \frac{ci\sigma_x(x)}{w}\frac{\partial v_y}{\partial y} - u\sigma_x(x) \tag{15}$$

However, now we can't get rid of the complex part of the equation as the term is still there. If we recognize that the $\frac{i}{w}$ part of the equation is an integration in the frequency domain we can introduce an auxiliary equation, ψ, and rewrite the original equation.

$$-iwu = c\nabla\cdot\vec{\boldsymbol{v}} + \psi - u\sigma_x(x) \tag{16}$$

$$\psi = \frac{ci\sigma_x(x)}{w}\frac{\partial v_y}{\partial y} \tag{17}$$

Doing the inverse Fourier-transform on (16) and (17) yields the two next transformed PDEs.

$$\frac{\partial u}{\partial t} = c\nabla \cdot \vec{\boldsymbol{v}} + \psi - u\sigma_x(x) \tag{18}$$

$$\frac{\partial \psi}{\partial t} = c\sigma_x(x)\frac{\partial v_y}{\partial y} \tag{19}$$

Now we have successfully introduced a PML in the x-direction by introducing an auxiliary PDE. We repeat the process for the y-direction. From (11) we get

$$\frac{\partial}{\partial y} \rightarrow \frac{1}{1 + \frac{i\sigma_y(y)}{w}}\frac{\partial}{\partial y} \tag{20}$$

The first equation, (18), gives the following equation after the transformation (20).

$$\frac{\partial u}{\partial t} = c \cdot \left(\frac{\partial v_x}{\partial x} + \frac{1}{1 + \frac{i\sigma_y(y)}{w}}\frac{\partial v_y}{\partial y}\right) + \psi - u\sigma_x(x)$$

Multiplying both sides with $1 + \frac{i\sigma_y(y)}{w}$, taking the Fourier transform and rearranging we get

$$-iwu = \left(c\frac{\partial v_x}{\partial x} + \psi - u\sigma_x(x)\right)\left(1 + \frac{i\sigma_y(y)}{w}\right) + c\frac{\partial v_y}{\partial y} - u\sigma_y(y)$$

We now face the same problem as in (15), where we cant directly get rid of the complex term. The solution is to introduce another auxiliary partial differential equation ϕ.

$$-iwu = c\frac{\partial v_x}{\partial x} + c\frac{\partial v_y}{\partial y} + \psi + \phi - u\sigma_y(y) - u\sigma_x(x) \tag{21}$$

$$\phi = \left(c\frac{\partial v_x}{\partial x} + \psi - u\sigma_x(x)\right)\frac{i\sigma_y(y)}{w} \tag{22}$$

Taking the inverse transform of (21) and (22) we get two new equations.

$$\frac{\partial u}{\partial t} = c\frac{\partial v_x}{\partial x} + c\frac{\partial v_y}{\partial y} + \psi + \phi - u\sigma_y(y) - u\sigma_x(x) \tag{23}$$

$$\frac{\partial \phi}{\partial t} = \left(c\frac{\partial v_x}{\partial x} + \psi - u\sigma_x(x)\right)\sigma_y(y) \tag{24}$$

The transformation on the velocity component in the y-direction from (3) gives almost the same result as for the x component in (14).

$$\frac{\partial v_y}{\partial t} = c * \frac{\partial(u + f)}{\partial y}$$

$$\Rightarrow -iwv_x = c\frac{1}{1+\frac{i\sigma_x(x)}{w}}\frac{\partial(u+f)}{\partial x}$$

$$\frac{\partial v_y}{\partial t} = c * \frac{\partial(u+f)}{\partial y} - \sigma_y(y) \tag{25}$$

The final equation we have to transform is (19). Applying the transformation and doing a Fourier transform we get

$$-iw\psi = c\sigma_x(x)\frac{1}{1+\frac{i\sigma_y(y)}{w}}\frac{\partial v_y}{\partial y}$$

Multiplying both sides by $1+\frac{i\sigma_y(y)}{w}$ and rearranging gives

$$-iw\psi = c\sigma_x(x)\frac{\partial v_y}{\partial y} - \sigma_y(y)\psi$$

$$\frac{\partial \psi}{\partial t} = c\sigma_x(x)\frac{\partial v_y}{\partial y} - \sigma_y(y)\psi \tag{26}$$

Finally, putting it all together we get a modified system with exponential absorption in the PML region.

$$\begin{aligned}
\frac{\partial u}{\partial t} &= c\nabla\cdot\vec{\boldsymbol{v}} + \psi + \phi - u(\sigma_y(y)+\sigma_x(x)) \\
\frac{\partial \vec{\boldsymbol{v}}}{\partial t} &= c\nabla(u+f) - \vec{\boldsymbol{\sigma}}\cdot\vec{\boldsymbol{v}} \\
\frac{\partial \psi}{\partial t} &= c\sigma_x(x)\frac{\partial v_y}{\partial y} - \sigma_y(y)\psi \\
\frac{\partial \phi}{\partial t} &= (c\frac{\partial v_x}{\partial x} + \psi - u\sigma_x(x))\sigma_y(y)
\end{aligned} \tag{27}$$

with

$$\vec{\boldsymbol{\sigma}} = \begin{bmatrix}\sigma_x(x)\\ \sigma_y(y)\end{bmatrix}$$

Note that PMLs are only reflectionless for the exact continuous wave equation (ie. the analytic version). In any computer simulation such as FDTD, there will be numerical reflections. They are however exponentially absorbed in the PML region, which gives suitable results for this project. Having too high absorption coefficient can result in reflections at the start of the PML region, while too low absorption results in that the waves travels through the PML region and reflects back.

3.3.1 Geometry

Obviously we want to include geometry which obstructs the propagation of sound waves and reflect them back. This is very easy to implement, but we might have to take some care depending on the grid system we choose to use. A wall can be included in the PDE by modifying the velocity at which a sound wave travels. For hard walls we set the velocity to 0 and we can even vary the absorption of the wall if we desire. As with the PML examples above, absorption is simply implemented by varying the velocity gradually over an area. As the PML functions determine the absorption of the PML layers it is natural to just add the geometry to the σ-functions. We define the geometry of the virtual space as g. For areas where we want the sound wave to propagate (ie. open space) $g = 0$, while for areas where we want the sound to absorb or reflect $0 \leq g < 1$. The g-values for the velocity directions also have to be calculated and is done using the gradient.

$$\overrightarrow{\boldsymbol{g_v}} = \nabla \cdot g \tag{28}$$

Modifying the original PDE (27) by multiplying the sound velocity with the geometry yields

$$\frac{\partial u}{\partial t} = c(1-g)\nabla \cdot \vec{\boldsymbol{v}} + \psi + \phi - u(\sigma_y(y) + \sigma_x(x)) \tag{29}$$

$$\frac{\partial \vec{\boldsymbol{v}}}{\partial t} = c\nabla(1-g) \cdot \nabla(u+f) - \vec{\boldsymbol{\sigma}} \cdot \vec{\boldsymbol{v}} \tag{30}$$

$$\frac{\partial \psi}{\partial t} = c(1-g)\sigma_x(x)\frac{\partial v_y}{\partial y} - \sigma_y(y)\psi \tag{31}$$

$$\frac{\partial \phi}{\partial t} = (c(1-g)\frac{\partial v_x}{\partial x} + \psi - u\sigma_x(x))\sigma_y(y) \tag{32}$$

3.3.2 Discretization of the modified system

As in section 3.2.2 we discretize the above equations in a leap-frog system. ψ and ϕ is computed at the half-steps as with the velocity.

$$u^{t+1} = (c(1-g)\nabla \cdot \vec{\boldsymbol{v}}^{t+1/2} + \psi^{t+1/2} + \phi^{t+1/2} - (\sigma_x + \sigma_y)u^t)\Delta_t + u^t \tag{33}$$

$$\vec{\boldsymbol{v}}^{t+3/2} = (c\nabla(1-g) \cdot \nabla(u^{t+1} + f) - \vec{\boldsymbol{v}}^{t+1/2} \cdot \vec{\boldsymbol{\sigma}})\Delta_t + \vec{\boldsymbol{v}}^{t+1/2} \tag{34}$$

$$\psi^{t+3/2} = (c(1-g)\sigma_x \cdot (\nabla \cdot \vec{\boldsymbol{v}}^{t+1/2})_y - \psi^{t+1/2}\sigma_y)\Delta_t + \psi^{t+1/2} \tag{35}$$

$$\phi^{t+3/2} = \sigma_y(c(1-g)(\nabla \cdot \vec{\boldsymbol{v}}^{t+1/2})_x + \psi^{t+1/2} - \sigma_x u^{t+1})\Delta_t + \phi^{t+1/2} \tag{36}$$

It looks a little messy, but it is suitable for implementation. Things to keep in mind is that σ_x, σ_y and their corresponding sum is constant. Further the divergence calculation is the same in (33), (35) and (36), so reuse of results is possible here. We can also possibly assume that the geometry is constant over the duration of the simulation. I will go into more detail in the methods (5.4.8) section.

3.4 Grid system - Staggered grid vs. Collated grid

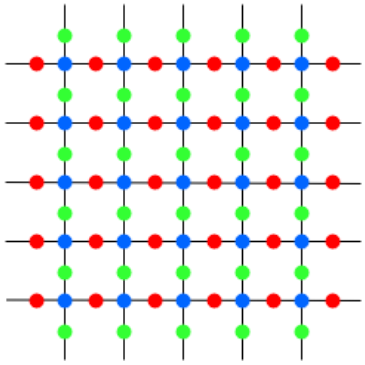

Figure 8: An example of a staggered grid. The blue dots represent pressure, the red dots represent velocity in the x direction and the green dots represent velocity in the y direction.

For now we have sort of deferred a discussion of the grid layout. The divergence and gradient operators as well as all the values are lacking coordinates, except for the simple forward Euler method. A method for giving second order accurate spatial derivatives "for free" is by placing the center of the pressure values in a different location than the center of velocity values. It is also more obvious what the derivatives are when using this system as, for example, the gradient of the pressure field is simply the right minus the left pressure value for the x-direction and similar for the y-direction. A staggered grid is illustrated in figure8.

Implementation-wise it gets more complicated using a grid like this, as we will have to be very careful when calculating the indices for the different fields. For example, the location of the pressure value is not the same as the location of the corresponding velocity value in, say, the x-direction even tho they have the same indices. This is a concern in the gradient calculation, the divergence calculation and the PML functions. In figure 9 we can see how the m and n indices can be mapped to the staggered grid. The transparent cells on the bottom and right are "dead" cells which we will have to take special care of in the implementation.

3.4.1 Derivative on a grid

By simply using the difference between a left and right value in the staggered grid we get second order accurate spatial derivatives. However, in an effort to avoid numerical dispersion we will devise a higher order derivative. Numerical dispersion occurs when some wave frequencies is travelling faster than other frequencies. This is problem in many FDTD methods for certain PDEs, and can be mostly eliminated by increasing the

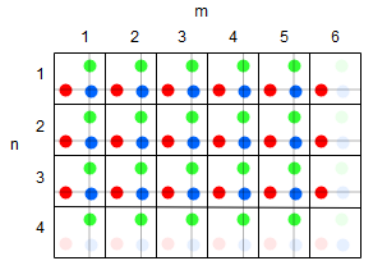

Figure 9: An example of index mapping for the staggered grid.

computation accuracy. Numerical dispersion can be seen in effect in the result section 6, and in this context it sounds like a sine sweeping up in frequency. It is a highly undesirable effect.

The coefficients of a derivative stencil can be found by solving a system of linear equations. In 3.2 we used a stencil with the coefficients $[1, -2, 1]$ for approximating the second derivative around a point. Here we wish to find the 6th order approximation around a point in a staggered grid. This gets a little complicated when the velocities are not aligned with the pressure field.

This section describes a method to get any derivative stencils around any point of any order. In order to do this we will have to go over some background theory concerning derivatives and approximations. What we have is an arbitrary function we wish to differentiate. A Taylor series is a representation of a function as an infinite sum of derivatives. It is defined by

$$\sum_{n=0}^{\infty} \frac{f^n(x_0)}{n!}(x - x_0)^n \tag{37}$$

x_0 is the point where we expand around. In our case we wish to find a stencil that can approximate the first derivative around x_n with the points $x_{n-5/2}$, $x_{n-3/2}$, $x_{n-1/2}$, $x_{n+1/2}$, $x_{n+3/2}$ and $x_{n+5/2}$. We start by writing the derivative as a linear combination of the Taylor expansions for each point.

$$D_6 f(x_0) = a_0 f(x_0 - \frac{5h}{2}) + a_1 f(x_0 - \frac{3h}{2}) + \cdots + a_5 f(x_0 + \frac{5h}{2}) \tag{38}$$

Each f can be approximated by a Taylor series around point x_0. For example the first term expands to

$$\begin{aligned} f(x_0 - \frac{5h}{2}) = &\frac{f^0(x_0)}{0!} + \frac{f^1(x_0)}{1!}(-\frac{5h}{2} - x_0)^1 + \frac{f^2(x_0)}{2!}(-\frac{5h}{2} - x_0)^2 \\ &+ \cdots + \frac{f^5(x_0)}{5!}(-\frac{5h}{2} - x_0)^5 \end{aligned}$$

Expanding all these functions, putting it back into (38) and solving for the second derivative (ie. why the right hand side looks like it does) turns this into a simple linear system we have to solve.

$$\begin{bmatrix} -(\frac{5}{2})^0 & -(\frac{3}{2})^0 & \cdots & (\frac{5}{2})^0 \\ -(\frac{5}{2})^1 & -(\frac{3}{2})^1 & \cdots & (\frac{5}{2})^1 \\ \vdots & \vdots & \vdots & \vdots \\ -(\frac{5}{2})^5 & -(\frac{3}{2})^5 & \cdots & (\frac{5}{2})^5 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_5 \end{bmatrix} = \begin{bmatrix} 0 \\ \frac{1}{h} \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{39}$$

We need the rows on the right hand side to be 0 except for row 1 as we which to find the first derivative. h is also moved from the matrix on the left hand side

to the right hand side. Solving this system yields a first order differential stencil with the values

$$\vec{\boldsymbol{a}} = \begin{bmatrix} 0.00469 & -0.06510 & 1.17187 & -1.17187 & 0.06510 & -0.00469 \end{bmatrix}^T \tag{40}$$

Note that the values for h is set to 1 in this case. Modifying for h is a matter of dividing this vector by h. This method is of course extendable to any differential and order, although we want $d \leq N$ where d is the order of derivatives and N is the number of stencil points. This system only requires a few lines of Python code to solve. Using the code in listing 1 we can easily experiment with different stencil sizes.

Listing 1: A function for generating differential stencils

```
import numpy as np
from scipy.linalg import solve

def calcDifferential(xs, order):
        A = np.array([np.power(xs, n) for n in range(0, len(xs))])
        b = np.zeros_like(xs)
        b[order] = -1
        return solve(A, b)
```

Putting this all together, the grid system and the differential stencil, we now have a system for calculating the divergence as well as the gradient. The divergence equation becomes

$$\nabla \cdot \vec{\boldsymbol{v}}_{m,n} = \vec{\boldsymbol{v}}_{m-2:m+3:,n} \cdot \vec{\boldsymbol{a}} + \vec{\boldsymbol{v}}_{m,n-2:n+3} \cdot \vec{\boldsymbol{a}} \tag{41}$$

In the implementation section this operation is further split into divergence in one direction and then in the other, as the differential operator also appears in (31) and (32). The gradient calculation becomes

$$\nabla u = \begin{bmatrix} u_{m-3:m+2:,n} \\ u_{m,n-3:n+2} \end{bmatrix} \cdot \vec{\boldsymbol{a}} \tag{42}$$

Looking at this equation, we can see that it is a 6×2 matrix multiplied by a vector of length 6 (we look at $u_{m-3:m+2:,n}$ and $u_{m,n-3:n+2}$ as a long row-vectors stacked on top of each other). Note the difference in indexing between (41) and (42). This is due to the staggered grid. Both of these operations can be calculated using a convolution over the different fields.

3.5 Hexagonal grid

A third approach to the grid layout is tested out in this project. As numerical dispersion is a huge problem in FDTD simulation and this project specifically, an idea to use a hexagonal grid was evaluated. An hexagonal layout can be embedded in a 2d array by mapping indices as seen in figure 11 and 10. As dispersion in a regular rectilinear grid is the worst along the diagonal (as is discussed in the result section), a hexgrid might be able to reduce some of these artifacts. This is also supported by [10].

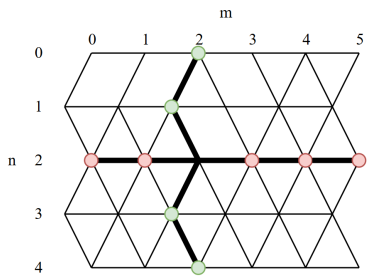

Figure 10: Hexagonal layout

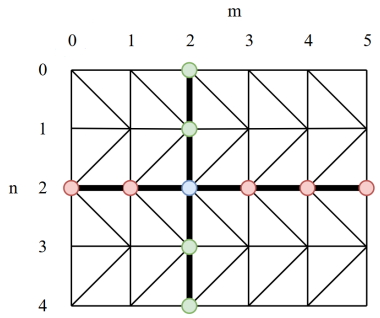

Figure 11: Hexagonal mapping of indices

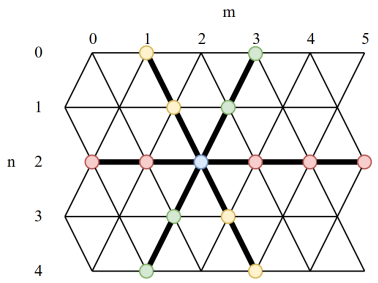

Figure 12: Derivative directions for the hexagonal grid

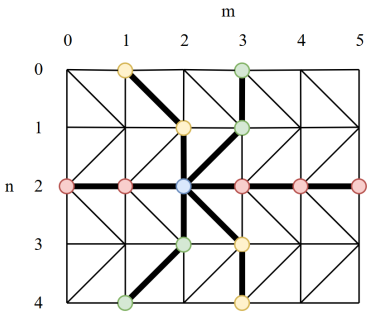

Figure 13: Mapping of the derivative to indices

Essentially, all we have to to is transform the indexing of the original rectilinear grid into a hexagonal indexing scheme and differentiate in three directions, as opposed to only y and x, seen in 12. The indexing used gets more complicated when realizing we have to change the indexing for each row. Ie. odd rows produces different indices than even rows. This can be seen in figure 13. The derivative in x-direction is the same as before, whereas in the y direction we get two convolutions with the indices

$$m_{tl} = \lceil \frac{i}{2} \rceil \tag{43}$$

$$m_{tr} = \lceil \frac{-i}{2} \rceil \tag{44}$$

and

$$m_{tl} = \lfloor \frac{i}{2} \rfloor \tag{45}$$

$$m_{tr} = \lfloor \frac{-i}{2} \rfloor \tag{46}$$

where i is the current index of the convolution, n is the same as before and tl and tr refers to the starting point of the convolution (top left and top right). An implementation and evaluation can be seen in section 5 and 6 respectively.

4 Technology

Having devised the necessary mathematical tools for this project, it is now possible to look into what technologies we could use in order to implement a simulator for the PDEs. During the duration of this project a lot of different technologies was considered. Some of them are listed in this chapter and described below, although more technologies, frameworks and libraries was evaluated.

Two programming languages was primarily used; C++ and Python. C++ is a high-level multi-paradigm programming language, developed as an extension of C. It includes features such as object oriented programming, generic programming (through templates) and functional programming. C++ compiles down to machine code and is thus able to generate very efficient code, at the cost of great complexity. Setting up projects and environments can be quite cumbersome (as experienced in this project) and developing for different platforms can present even bigger challenges. Python on the other hand is quick and easy to set up, but can't match C++ in performance on account of that it is an interpreted language. It does however work very well as a prototyping language, and it emphasizes code readability and ease of usage. It contains the same paradigms as C++, which makes translating Python code to C++ code fairly straight forward. Python is in this project used in conjunction with Numpy for doing linear algebra and Matplotlib to produce visualization of the results.

4.1 The audio processing pipeline

Audio applications in music production (Digital Audio Workstations, or more commonly DAW) typically is comprised of a host program and several plug-ins, usually implemented through a common interface (for example VST, AU, RTAS). Examples of DAWs include FL Studio 20, Ableton Live, Reason, Logic etc. Common to all these applications (and many others) is that they implement VST hosting (or possibly some of the other interface types). This means that any audio processing plug-in implementing the VST interface can be run in any of these hosts. This makes developing plug-ins (VSTs from here on) for any host possible through a simple interface. This is similar to how it is possible to develop games for "any" machine through one common API specification, like OpenGL.

There does not seem to be that much usage of *general-purpose computing on graphics processing units* (GPGPU) in modern VST development, most likely because the CPU is capable of handling most tasks related to real-time audio processing. Further, for an application where real-time processing is key, it is crucial that the processing has as little as possible of latency. This will make real-time processing on the GPU less feasible as memory transfers to and from the GPU is slow. Most importantly, the general purpose GPU computing APIs today are often vendor locked. The two most popular competing APIs are CUDA from NVIDIA and OpenCL. CUDA is fairly mature, but is limited only to NVIDIA graphic cards. OpenCL can run on any OpenCL-capable device, but has somewhat bad support

among certain vendors (NVIDIA supports OpenCL up to version 1.2, while the newest version is 2.2). So, a fairly large tradeoff has to be made regarding which platforms to support. If cross platform usage is a requirement, developers is pretty much limited to OpenCL 1.2. This is however not a problem in this project. In this project a *proof-of-concept* VST will be developed. Cross platform support is not a concern. As we will see in section 5 the application in itself is split into a standalone pre-processing part and one runtime-part using the results obtained from the pre-processing. The ideal result would however be simulation and runtime in the same application, and attempts to integrate the two applications together is carried out.

VSTs are usually split into two categories; synths (usually distincted by "VSTi") and effects. VSTi's takes as input MIDI commands and returns audio. Further, they contain a set of parameters which the host can automate (ie. change over time) and these parameters can be stored as presets. One preset usually corresponds to one "sound". More generally a preset is the state of all parameter values saved in some file. Examples of VSTi's can be "a general synth" (using oscillators and effect chains and so on), a recreation of a known synthesizer (for example Voyager Plug SE which is a recreation of the Moog synth) or an audio sampler which plays sounds recorded from real instruments. In this way realistic recreations of real instruments can be made. Sampling is suitable for instruments like piano or organs, but less suitable to instruments where the tone is largely dependent on how a player plays them. For example, Wikipedia lists 16 different techniques to play a trumpet. A complete audio sampler would have to capture each of these techniques for each possible pitch with different velocities. Pianos really only have the pitch and velocity, making them much easier to sample.

A VST effect plug-in usually takes in audio (possibly multiple channels, but most commonly only a left and right channel) and sometimes MIDI commands, does some processing on that audio and returns the resulting audio back to the host. Examples include distortion, filtering, dynamic range compression or reverberation. Multiple effects can be stacked on top of each other producing complex sounds and effect chains. An important function class in many of these effects is linear functions which can be modeled by convolutions, as described in section 3. Filters, delay and reverberation are examples of such effects.

As mentioned, an impulse response can accurately model the acoustical properties of a given room. However, generating such impulse responses from "real" spaces is hard and requires a lot of computational power. There is alternative methods to make reverberation plug-ins. For example feedback delay circuits create a large, decaying series of echoes. This method is relatively easy to implement, but is not that realistic. Methods to improve this can be adding filtering the the delays like low-pass and high-pass. Commonly, reverberation effects are also split into ER/LR parts (early reflections and late reflections). The early reflection can be simulated with a FDTD solver, while the late reflection, or tail as it is called, can

be produced through more algorithmic approaches such as feedback delay circuits. This have the effect of greatly reducing simulation time, while preserving realistic effects, as the tail typically is not that different from impulse response to impulse response. One such algorithm is the Schroeder's algorithm [11] which makes use of tapped delay lines, comb filters, and allpass filters.

4.2 JUCE

JUCE s a framework for desktop and mobile application development. It is particularly used for it's GUI and plug-in libraries which can target several of the most important audio plug-in interfaces. JUCE allows applications to be written cross platform and supports several environments and compilers. It is partially open-source, and it's license states that it can be used for free given non-commercial purposes.

JUCE has a lot of different features, ranging from GUI elements, threading, graphics, audio as well as digital signal processing modules such as FFTs and delays. Developers needing a lot of third party libraries might be able to only stick to JUCE due to it's many modules. In addition to the framework, a tool called the "Projucer" is supplies. It is an IDE for managing and creating JUCE projects, and even includes an integrated GUI editor. For the most part, Projucer was avoided in this project (apart from project creation) as Visual studio and Projucer did not work well together.

Some alternatives to JUCE includes iPlug which is a C++ framework for developing cross platform audio plugins. While possibly usable for this project, the first version is obsolete and the second version is only in pre-release. That is JUCE seems like a more mature framework. Another option would be to use the VST interface directly and any compatible GUI framework, however this did not seem to be worth the hassle.

4.3 CUDA

CUDA is a parallel computing platform made by NVIDIA for heterogeneous computing on their CUDA-enabled graphic cards. CUDA was launched in 2007 and is widely used for accelerating applications with heavy computation demand. CUDA is particularly suited to simulations as a lot of calculation can be done in parallel. This is also the case for this project. Importantly, CUDA is a proprietary API made by NVIDIA for NVIDIA graphics cards. This means consumers with an AMD card would not be able to run the code in this project. For this project a NVIDIA Quadro P2000 was used, which contains a Pascal architecture (relevant to what computing architectures I'm able to target for this project).

4.3.1 CUDA APIs

The CUDA API is divided into several layers of abstraction, as seen in figure 14. The highest level abstraction is libraries implemented with CUDA such as cuBLAS, cuDNN, cuFFT, CUDA Math Library etc. A complete list can be found on the NVIDIA website [12]. These libraries can easily be integrated into existing application for hardware acceleration. The next level is the CUDA runtime API, which contains convenient methods for copying, using and allocating memory, and launching kernels. The CUDA runtime API makes it possible to link CUDA kernels into executables, which has a number of benefits like not having to distribute `.cubin` or `.ptx` files with the application (these are the resulting extensions of the compiler). At the lowest level there is the CUDA driver API, which gives the programmer more control over the execution at a higher complexity cost. The programmer has to manually load and initialize module. The CUDA runtime API is implemented on top of the driver API, and they are not mutually exclusive.

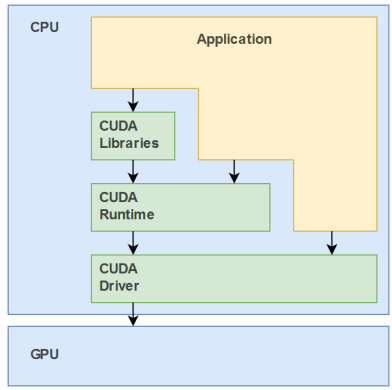

Figure 14: CUDA API abstractions

The NVIDIA CUDA toolkit also has it's own compiler, called *nvcc*. It serves as an intermediary step of transforming the CUDA source code into executable CUDA binaries. It requires a general purpose C++ host compiler as the standard C++ code is forwarded to this compiler. The CUDA compiler accepts a set of conventional compiler options, such as macros, include/library paths, optimization flags and target flags. Common targets include `.ptx`, `.cubin`, `.gpu`, hybrid object file and hybrid `.c` files. *ptx* is the one used for this project, and is compiled to a low-level parallel thread execution virtual machine and instruction set architecture. These *ptx*-files is then loaded manually through the CUDA driver API. One important point to stress is that the programmer has to compile for specific ar-

chitectures. By specifying the `-arch` flag we set the architecture we are compiling for. This can either be a real computing architecture (`sm_XX`) or a virtual architecture (`compute_XX`). For this project we are only compiling for the real architecture `sm_61`, which corresponds to the Pascal architecture. A more complete description can be found on the NVIDIA websites [1].

4.3.2 CUDA memory model

The memory model for CUDA is one of the most important parts of the CUDA architecture. There exists a couple of different memory types in CUDA, and all have their specific use cases depending on the problem at hand. In general, the more general the memory is, the slower it is. For example global memory is one of the slowest types, but also the most general as it can easily be both written and read from by different threads in execution across kernel execution. Register memory on the other hand is the fastest, but only exists per thread execution, making communicating this memory to other threads only feasible through some memory higher in the hierarchy. Below is the available memory types available for usage. Selecting the correct memory type / location can be crucial for performance as memory typically is the bottleneck for GPU programming.

Global memory Accessible from all threads, both readable and writable from device and host.

- **Constant memory** does not change during the kernel execution, writable from host, but only readable from device. Faster than global memory as the contents can be efficiently cached.
- **Texture memory** is similar to constant memory, main difference being the layout of the memory. When adjacent memory locations (meaning adjacent on a 2D or 3D grid) from threads in a warp is being read from, the GPU can cache this more efficiently.

Shared memory is both readable and writable from all threads in a block and lasts for the duration of that block. Shared memory performs very fast, and when applicable it can boost performance of applications ten-folds.

Register memory is visible only to the thread that owns it, and lasts for the duration of the thread.

Local memory has the same scope rules as registers, but performs slower. It's usage is for storing data that does not completely fit in the register. When this happens, it is called a *register spill* and should be avoided.

Behind the scenes the memory architecture uses registers and caching for better performance. This is important to keep in mind when aiming for efficient code. If

[1] For example at `https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#gpu-feature-list`. A lot of details have been glossed over, a more complete description can be found here: `https://stackoverflow.com/questions/35656294/cuda-how-to-use-arch-and-code-and-sm-vs-compute`

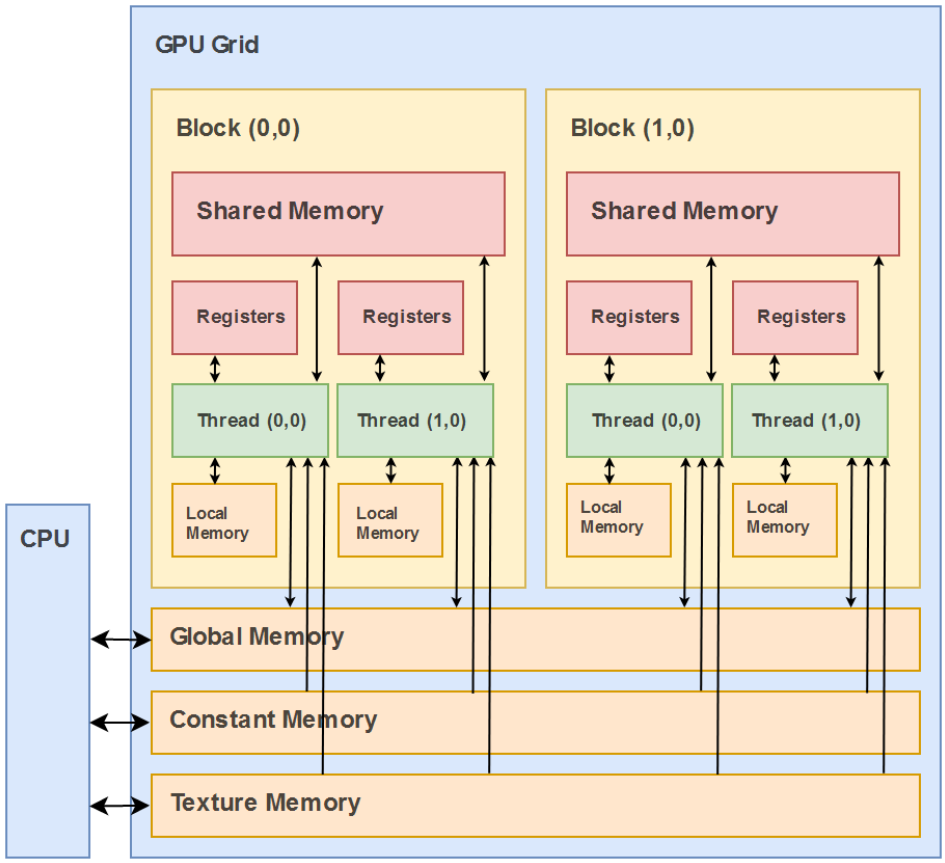

Figure 15: CUDA memory architecture

the entire contents of the kernel memory can be fit into the registers (and shared memory depending on the problem), we can get a huge performance boost. A diagram of how the different memories interact and is logically located can be seen in figure 15.

4.3.3 CUDA programming model

The CUDA C/C++ programming model enables the use of a set of extension to the C/C++ languages for enabling heterogeneous computing. It is designed in such a way that already existing applications can easily be modified to work in parallel on the GPU. Programs is divided up into serial and parallel parts where the parallel parts are run on the GPU, or *device*, and the serial part is run on the CPU, or *host*. When writing efficient CUDA code one has to be aware of some of the architectural characteristics of the GPUs. For example the different memory

types is crucial for enabling fast computations as memory transfers between host and device is typically a bottleneck for heterogeneous computing. Further a lot of the parameters when invoking *kernels* (functions executed by threads on the GPU) is bound by hardware restrictions and how these values play together is important to be aware of. In CUDA GPU kernels are declared by the keyword `__device__` or `__global__`. Global functions are kernels that can be launched from both the host code and device code, while functions declared with *device* can only be launched from a kernel. Kernels are launched by specifying parameters for how many blocks and threads are required. Further if dynamic shared memory is used, we'll have to specify that as well. An example of a CUDA kernel launch can be seen in listing 2. Here we are launching n blocks with m threads each and each block has `100 * sizeof(float)` (400 bytes) of shared memory at their disposal.

Listing 2: Launching a CUDA kernel

```
__global__ void foo() {}

foo<<<n,m,sizeof(float)*100>>>();
```

Execution of a kernel on the GPU is done on a *grid*. A grid contains a set of blocks, each block contains several threads and these threads is executed concurrently[2], as shown in figure 16. The block and thread configurations can be either 1D, 2D or 3D but the total number of blocks and threads is limited. Executing kernels beyond these limits will result in errors. For example, in CUDA computing model 6.1, which is the one used for this project, the maximum number of threads is 1024 in total (so possible configurations is (32 x 32), (1 x 1024 x 1), (4 x 4 x 64) etc.) and the maximum number of blocks is 2^{32} - 1 in the x-dimension, and at most 65535 blocks in the y and z dimensions. When a kernel grid is invoked, the blocks of the grid are distributed among a set of *Streaming Multiprocessors*. Each streaming multiprocessor executes a single thread warp. Each instruction in the threads are executed in lockstep, meaning each thread executes the same instruction over different data. A SM contains 8 CUDA cores. As a warp only can hold 32 threads at a time, care has to be taken when requiring synchronization among blocks containing more than 32 threads. Such explicit synchronization can be accomplished by the built in function `__syncthreads()`.

4.4 OpenGL

OpenGL is a graphics API specification for interfacing with the GPU and drawing 2D and 3D graphics to a window. It is cross-platform and should work on Windows, Linux and MacOS, as well as on mobiles. OpenGL is typically described as a huge finite state machine.

[2]All the threads in a block is not strictly speaking executed in parallel. Threads is executed in thread warps of size 32 at a time. This has consequences for how you write and read from the shared memory in a block.

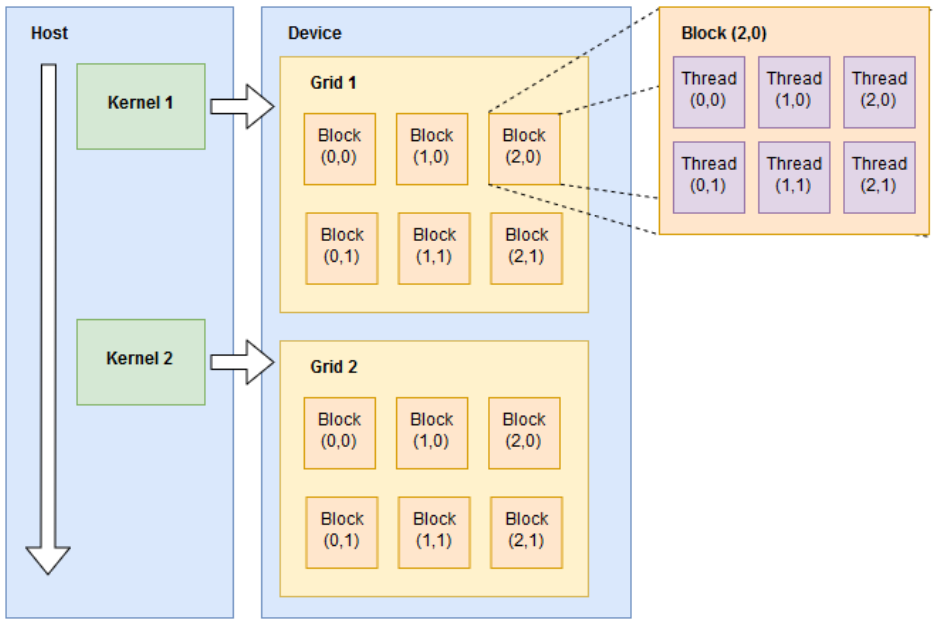

Figure 16: The CUDA execution model

In order for OpenGL to work on a system, it needs a context. A context stores all state information associated with an instance of OpenGL, and typically includes a window. The way a programmer goes from here is typically to "ask" the GPU what capabilities it has. The programmer usually sets the screen pixel format and related information during initialization. Once this is done the programmer has to load function pointers to the different functions contained within OpenGL. All this initialization is usually abstracted away with libraries, and is typically not something the average programmer has to deal with. For creating a window and context the library *GLFW* can be used, while the function pointers can be loaded with an extension loader like *GLAD*. Going to the GLAD website [3] shows all the different versions that a programmer can target, which is a lot. The version selection typically depends on the consumer GPUs a programmer might want to target.

4.4.1 Programming with OpenGL

Programming with OpenGL involves calling functions to interface with the GPU. Common resources include textures, buffers and shaders. Textures are objects (memory really) that contains one or more images that all have the same image format. These objects is used in samplers to read values in a *shader* or can be used as *render targets*. There are different texture types; 1D, 2D, 3D and cube

[3] `https://glad.dav1d.de/`

maps. Buffers typically stores vertex data or index data. In the context of 3D, a mesh contains indices and vertices, which is typically comprised of positions, colors, texture coordinates etc. and the indices is a long list of integers that defines triangles from the vertex list. Ie. three indices might define one triangle comprised of three vertices. Shaders are small programs that define how a pixel ends up looking. Different shader types are usually coupled together to make a shader program. Shader types include vertex and fragment shaders, geometry shaders and tesselation shaders. Vertex shaders is used for transforming vertices while fragment shaders is used for calculating lightning, color etc. A geometry shader is able to generate new primitives on the fly. A fragment has a window space position, a few other values, and it contains interpolated per-vertex output from the vertex processing stage. These fragments may or may not end up on the screen. In this project vertex, geometry and fragment shaders are used in conjunction.

The most common way to use OpenGL is to write wrappers that encapsulates all the details of interfacing with OpenGL. This is also what I have done for this project. There are several areas where OpenGL is relevant in this project. Most importantly where I'm drawing the simulation in the simulator prototype. This is also what has generated all the result images. Further, when drawing the waveform in the convolver, OpenGL is used.

4.5 Other alternative technologies and APIs considered

4.5.1 OpenCL

OpenCL is another API for doing general purpose programming on the GPU. Initially, this API was chosen for the project and the early version used this API. However, OpenCL is old and not very well supported on NVIDIA GPUs. Only version 1.2 is supported for the graphics card used in this project, while the current version of OpenCL is 2.1. This makes programming with it hard, as information found online most likely concerns version 2. The main benefit with OpenCL is that it's not proprietary and is supported on a wide range of hardware. It is also not specifically made for use with GPUs, but rather across heterogeneous platforms. OpenCL works in much the same way that OpenGL does, where you would write compute shaders, upload the source code to the device and having it compiled there.

4.5.2 NVIDIA GVDB Voxels

GVDB Voxels is a library for simulation, computing and rendering of sparse voxels on the GPU. The library abstracts the complexity of sparseness in order to support computing and rendering of high resolution volumes. The main idea for this project was to use the library for voxelization and then use the methods described in [1] for simulation. It however turned out that using the library was quite complicated with little documentation, and a fair portion of this project was spent trying to get this library to work properly. The main problem was that it

is a relatively new library and therefore not much documentation and information exists on the web. There is however a programming guide [4], but debugging errors and problems was hard.

4.5.3 Vulkan and Direct3D

Both of these APIs are alternatives to OpenGL. Whereas Vulkan is cross-platform, Direct3D is proprietary to Microsoft. Vulkan is a relatively recent API (released in February of 2016) intended to offer more balanced CPU versus GPU usage and higher performance. It is considered a successor to OpenGL (while OpenGL is still largely supported across all platforms and that likely wont change any time soon [5]), but shares more with Direct3D 12, Metal and Mantle in terms of concept and feature set (ie. it is object oriented as opposed to state machine based). OpenGL was primarily chosen for this project because of prior knowledge and that JUCE supports OpenGL out of the box (although in a very convoluted way as we will see in 5).

4.5.4 BLAS and clBLAST

As this problem involves a lot of linear algebra, it makes sense that linear algebra libraries could be useful here. BLAS stands for Basic Linear Algebra Subprograms and is a specification for linear algebra routines. There exists a lot of implementations for different use-cases and hardware, the most popular being ATLAS, Eigen BLAS, cuBLAS (from NVIDIA for CUDA-enabled GPUs) and clBLAST (for OpenCL). For the first version of the application (when OpenCL was used), I used the BLAS specification (clBLAST as implementation) for doing the matrix multiplications. As the methods in the superior later version was based around convolutions, rather than matrix multiplications, BLAS became obsolete for this project.

[4] Found here: `https://developer.nvidia.com/gvdb`

[5] For example NVIDIA still recommends OpenGL in a lot of usecases (`https://developer.nvidia.com/transitioning-opengl-vulkan`) as it is simpler to maintain and program for due to it's maturity and lower complexity.

5 Method and Implementation

This section describes the overall workflow, methods and algorithms employed and implementation details. Due to a very steep learning curve, involving learning the mathematical basics of acoustical simulations, more advanced CUDA programming, a new framework (JUCE) as well as learning C++ in more detail, the project became sort of scattered among several repositories and code bases. However, by the end an effort to bring them together was carried out. A rough overview of the different code bases and repositories can be seen in figure 17. In this section the final simulator refers to the GLFW application capable of loading an impulse and generating a response. The final VST refers to the run-time convolver with an integrated simulator. The convolver is capable of loading an impulse response generated from the final simulator and convolve an incoming signal. There was two very early versions of the VSTs. One used OpenCL and BLAS in order to simulate waves, while the other used the CUDA run-time API. Both of these was discontinued as the methods and solutions there proved insufficient.

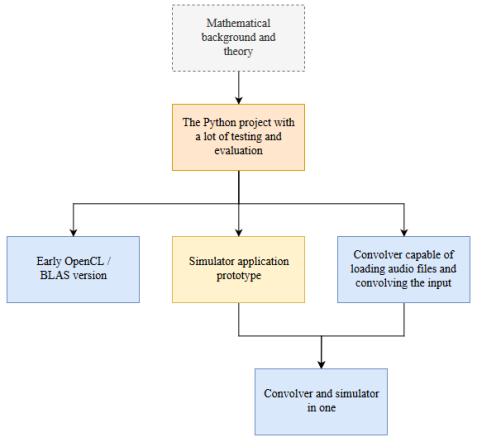

Figure 17: Overview of the different code bases and how they relate. The blue boxes is VST projects, the yellow box represents the GLFW project and the orange box represents the Python project.

5.1 Workflow

My general workflow consisted of reading papers and evaluating their methods, implementing in Python and then possibly implementing the methods in CUDA if the Python phase worked out. Several methods was tested out, as described in the theory section 3.

- Regular Forward Euler integration on a rectilinear grid

- Two-step Leapfrog on a rectilinear grid
- Two-step Leapfrog on a hexagonal grid
- Two-step Leapfrog on a staggered grid
- Analytic scheme with time steps as in Forward Euler and exchanging borders

Further, several attempts to bring this problem into 3D was carried out, although none was particularly successful. While extending these methods to three dimensions is fairly straight forward mathematically, the pre-processing steps necessary is more involved. For example, generating a *filled* voxelized mesh from a polygonal mesh was not as easy as it sounds. A shell voxelization is as easy as voxelizing each triangle individually, while filling said mesh is harder. While there probably exists straight forward methods, like flood filling with some special handling of whats "inside" and "outside", all special cases was harder to take care of. What I ended up trying [6] is creating a binary space partitioning of the scene and then check each voxel if it was inside or outside of the mesh. While this proved successful of some meshes, the algorithm broke for more complicated meshes and I could not figure out why this was the case.

5.1.1 Regular forward Euler integration on a rectilinear grid

This method was described in section 3.2.1. A very basic implementation can be found in the file `wave_2d_ex.py`. This implementation uses a matrix multiplication instead of convolution and is only using a second order spacial derivative. The matrix multiplication is implemented as two matrix multiplication and an addition.

$$\mathbf{D} = \begin{bmatrix} -2 & 1 & 0 & \dots & 0 \\ 1 & -2 & 1 & \dots & 0 \\ 0 & 1 & -2 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & -2 \end{bmatrix} \tag{47}$$

$$\mathbf{U} = \begin{bmatrix} u_{0,0} & u_{1,0} & \dots & u_{m,0} \\ u_{0,1} & u_{1,1} & \dots & u_{m,1} \\ \vdots & \vdots & \ddots & \vdots \\ u_{0,n} & u_{1,n} & \dots & u_{m,n} \end{bmatrix} \tag{48}$$

$$r = \frac{c * \Delta_t}{h} \tag{49}$$

$$\mathbf{U}^{t+1} = r^2 \cdot (\mathbf{U}^t \cdot \mathbf{D} + \mathbf{D} \cdot \mathbf{U}^t) + 2 \cdot \mathbf{U}^t - \mathbf{U}^{t-1} \tag{50}$$

There are obvious problems with this approach, like only being able to have square domains and only being a second order method (which, admittedly, is easily fixable). However, the implementation is *really* simple. The update function simply looks like listing 4.

[6] Based on `https://stackoverflow.com/a/4293454`

Listing 3: Update for the wave equation

```
Un = pow(r,2) * (U.dot(D) + D.dot(U)) + 2 * U - Ul
Ul = U
U = Un
```

This was further implemented in C++ with the library clBLAST which is a matrix multiplication specification (BLAS) implemented with OpenCL. The code in C++ with clBLAST can be seen below.

Listing 4: Update for the wave equation with clBLAST

```
void compute::Solver::step()
{
        // determine which buffers are the new ones, current and old.
        cl_mem newBuf = bufs_U[(iteration + 2) % 3];
        cl_mem currentBuf = bufs_U[(iteration + 1) % 3];
        cl_mem oldBuf = bufs_U[(iteration + 0) % 3];

        // events are for making sure things happens in the correct order
        float r_sq = pow(c * k / h, 2);
        cl_event e_1;
        cl_event e_2;
        cl_event e_3;
        cl_event e_f;

        // c := alpha*a*b + beta*c for side = 'L'or'l'
        // c := alpha*b*a + beta*c for side = 'R' or 'r'

        // we need to flip the old buffer
        CL(clWaitForEvents(1, &previous_cycle));
        CL(clReleaseEvent(previous_cycle));
        CLBLAST(clblast::Axpy<float>(dimension * dimension, -2, oldBuf, 0, 1,
             oldBuf, 0, 1, &ctx.getQueue(), &e_f));
        CLBLAST(clblast::Symm<float>(clblast::Layout::kRowMajor, clblast::Side::
             kLeft, clblast::Triangle::kLower, dimension, dimension, 1, buf_D,
             0, dimension, currentBuf, 0, dimension, 0, newBuf, 0, dimension, &
             ctx.getQueue(), &e_1));
        CLBLAST(clblast::Symm<float>(clblast::Layout::kRowMajor, clblast::Side::
             kRight, clblast::Triangle::kLower, dimension, dimension, r_sq,
             buf_D, 0, dimension, currentBuf, 0, dimension, r_sq, newBuf, 0,
             dimension, &ctx.getQueue(), &e_2));

        // y := y+alpha*x
        CLBLAST(clblast::Axpy<float>(dimension * dimension, 2, currentBuf, 0, 1,
              newBuf, 0, 1, &ctx.getQueue(), &e_3));

        // last sub
        CL(clWaitForEvents(1, &e_3));
        CL(clWaitForEvents(1, &e_f));
        CLBLAST(clblast::Axpy<float>(dimension * dimension, 1, oldBuf, 0, 1,
             newBuf, 0, 1, &ctx.getQueue(), &previous_cycle));

        CL(clReleaseEvent(e_1));
        CL(clReleaseEvent(e_2));
        CL(clReleaseEvent(e_3));
        CL(clReleaseEvent(e_f));
        // at this point the new U is in newBuf...

        iteration++;
}
```

It is worth noting that the function names for matrix multiplications are sort of cryptic. `Axpy` stands for **a** multiplied by **x** plus (ie. p) **y**. `Symm` is a symmetric matrix multiplication. The rest of the BLAS specification follows a similar structure,

and can be found in more detail in the official specification [7]. The full implementation can be found in `ReverbSimulator` in the file `WaveEq.cpp`, and this function is from the function `compute::Solver::step()`.

5.1.2 Two-step Leapfrog on a rectilinear grid

This is the version that ended up in the final application. It will be described in more detail in the next section.

5.1.3 Two-step Leapfrog on a hexagonal grid

This method was implemented in Python. The idea was that a hexagonal grid would give less numerical dispersion. While most of the standard leap-frog integration stayed the same, the differentials was different as we had to account for a different grid layout. In fact, we had to do a differentiation in three directions. While this probably wont matter to much in a CUDA implementation in terms of performance (as the texture lookups wouldn't be that different), the Python version ran slow as we could not directly use the numpy convolution routines. The code can be found in in the Python project under `laplacian.py`. The difference in complexity of the hexagonal Laplacian is quite large compared to the standard one. We can see in figure 18 that the difference is not too large in terms of numerical dispersion. This example uses a 6th order spacial differential. Over the diagonal in the regular implementation there seems to be greater artifacts than in the hexagonal implementation, but the difference is not that large that it is worth a full C++ implementation.

5.1.4 Two-step Leapfrog on a staggered grid

This method seems quite good on paper, and the initial simulations ran in Python seemed to be working fairly decent. However, when using a grid where the pressure and velocity components are decoupled (described in section 3.2.2), the field tends to look like a checker board. This is a problem that arises due to the nature of the central difference scheme when applied to the divergence operator and the pressure gradient operator. [2] does not mention how they solved this, and for the simulation carried out here it seemed better to just go with the regular two-step Leapfrog on a rectilinear grid. It did give the best results and easiest implementation anyway. The checkerboard problem is apparent in figure 19, which is a result from the `twostepleapfrog.py` file.

5.1.5 Analytic scheme with time steps as in Forward Euler and exchanging borders

This is by far the most advanced method. The time step simulation itself is fairly straight forward and is implemented in the same manner as the regular forward

[7] `http://www.netlib.org/blas/`

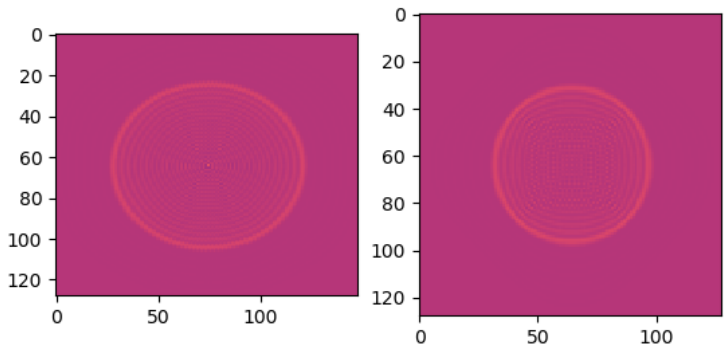

Figure 18: The hexagonal grid layout (left) vs. regular layout (right)

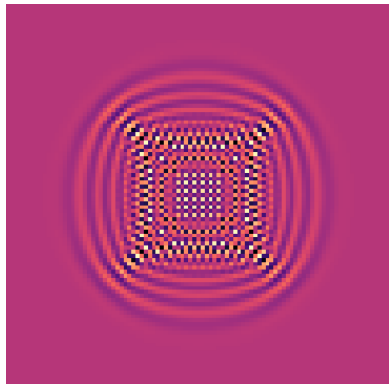

Figure 19: The checkerboard problem.

Euler described in 5.1.1. The spacial values are however represented as cosine coefficients, and given by

$$p(x,y,t) = \sum_{i=(i_x,i_y)} \hat{p}_i(t)\Phi_i(x,y) \tag{51}$$

where $\hat{p}_i(t)$ is time-varying coefficients of the pressure field and Φ_i is the eigenfunctions of the Laplacian. These values are given by

$$\Phi_i(x,y) = cos(\frac{\pi i_x x}{l_x})cos(\frac{\pi i_y y}{l_y}) \tag{52}$$

where l_x and l_y are the domain dimensions and i extends from $(0,0)$ to (l_x,l_y). Transferring to frequency domain and back is now simply a discrete cosine transform. In total the transformed wave equation becomes

$$\frac{\partial^2 M_i}{\partial^2 t} + c^2 k_i^2 M_i = DCT(F(t)), k_i^2 = \pi^2(\frac{i_x^2}{l_x^2} + \frac{i_y^2}{l_y^2}) \tag{53}$$

where c is the speed of sound. In order to transform this to a numerical integration scheme, we use a simple forward Euler integration step, as described in 3.2.1. This gives us the following update rule;

$$M_i^{t+1} = 2M_i^t cos(\omega_i \delta_t) - M_i^{t-1} + \frac{2 * DCT(F^t)}{\omega_i^2}(1 - cos(\omega_i \delta_t)) \tag{54}$$

where $\omega_i = ck_i$. This is now suitable for implementation. One small detail that is important to keep in mind is that the ω_i^2 term in the denominator of the forcing term is 0 for $i_{(0,0)}$. The following term $(1-cos(\omega_i \delta_t))$ is however also 0 for $i_{(0,0)}$, so for $i_{(0,0)}$ we simply use the first forcing tern, ie. $2 * DCT(F^t)_{(0,0)}$. The preceding formulations can be found in more detail in [1].

This basic formulation is implemented in `analytic3.py`, and the update step looks like 5.

Listing 5: Update step for the analytic solution

```
pressure[(current + 1) % 3] = 2 * pressure[current] * multipliers - pressure[(
    current - 1) % 3] + 2 * dct2(forces) * np.divide((1.0 - multipliers), np.
    power(T, 2), where=T!=0)
```

The result seems promising as seen in 20, although this is only for a simple rectangle with no border exchange and constant velocities, which makes is sort of useless.

The problems start to present them self when trying to incorporate geometry. In order to be able to exchange information between to adjacent domains, the pressure values has to be transformed back to regular spacial domain and then out into the forcing term of the neighbour.

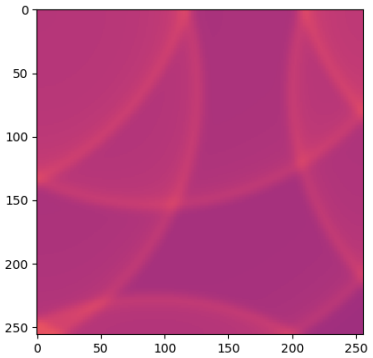

Figure 20: Analytic timestepping

5.2 Implementation of the simulator

The implementation of the final simulator is partly based on an example of a CUDA OpenGL interopability example [13]. The prototype uses Immediate Mode Graphical User interface (ImGuI) for drawing of user interface and GLFW for window handling. The simulator application can be seen in figure 21.

This method was first implemented in Python in the file `twostepleapfrog.py` and then ported to C++. It it this version that ended up in the final simulator application, with the exception that the PML calculation has been simplified. [2] used an approach where σ_x and σ_y were constrained to $\sigma_x = \sigma_y = \sigma$, which allows for a much simpler implementation involving no auxiliary PDEs. The modified equations is thus

$$\frac{\partial u}{\partial t} = c\nabla \cdot \vec{\boldsymbol{v}} - \sigma u \tag{55}$$

$$\frac{\partial \vec{\boldsymbol{v}}}{\partial t} = c\nabla(u + f) - \sigma \vec{\boldsymbol{v}} \tag{56}$$

This absorbed the outgoing waves slightly worse, thus requiring more layers than the Python version, but the implementation became much simpler and efficiency became greater. It did not appear to have any audible artifacts because of this. As both OpenGL and CUDA lives on the GPU it makes sense that they should be able to operate together. This project is heavily dependent on 2D textures, and as such, I have written wrappers that easily connects OpenGL textures with CUDA textures. Usage of CUDA textures in OpenGL rendering loop can be done with almost no extra overhead of memory transfers. This works as long as CUDA and OpenGL belongs to the same context. Some implications of this fact can be seen in section 5.4.6.

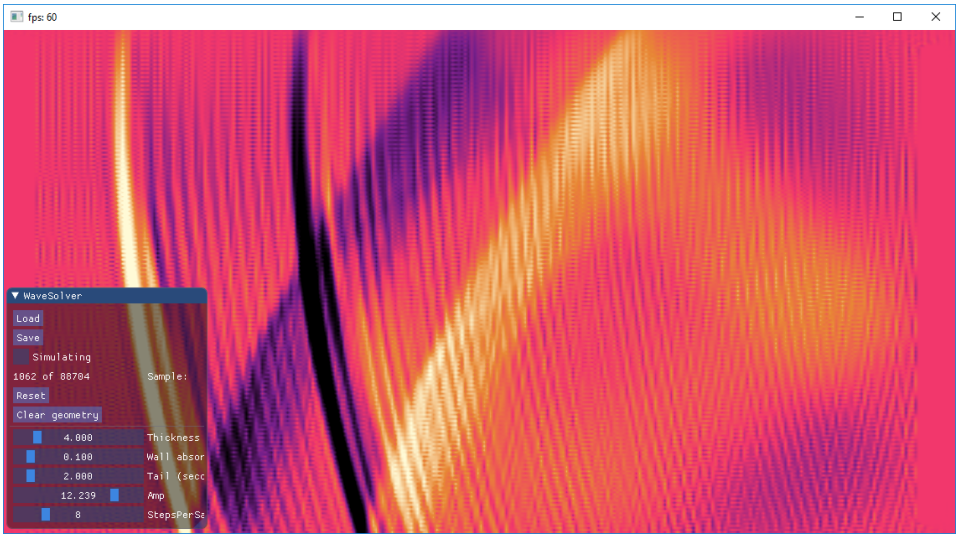

Figure 21: The final simulator application

5.2.1 The main program

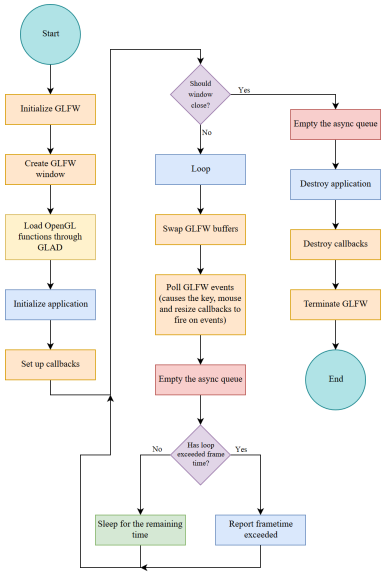

Figure 22: Program loop

Program The main simulation application is written as a single-threaded application. Because of this, the CUDA context and OpenGL context live on the same thread. The main class `WaveSolver` is extending a base class `Program` which hides the details of the GLFW interfacing. It also makes it possible to port the functionality to other applications without too much effort (as was originally intended). The `Program` class provides methods for mouse handling, keyboard handling, resizing events, asynchronous execution from other potential threads (through the class `AsyncQueue`) and three main methods that defines what the application should do; `init()`, `loop(float dt)` and `destroy()`. It runs at a fixed controllable frame rate and reports back when this frame rate is exceeded. This is useful for analyzing how fast the main simu-

lation thread is running. A diagram of how the lifetime of the application can be seen in figure 22. The blue squares are steps which can be overridden by the class using this base class (ie. `WaveSolver`), orange are all the steps dealing with GLFW, yellow is the step dealing with GLAD, red are the points where the execution queue is emptied and green is simply responsible for ensuring a fixed frame rate.

WaveSolver This is the main class of the application. It contains most of the functionality for controlling the simulation. For further work, it probably should be split up into smaller classes responsible for smaller functionality, but for this prototype it does not matter too much. As the `WaveSolver` extends from `Program` it was a couple of virtual methods implemented. In `init()` we are initializing CUDA and ImGui. This is done in their own separate functions. The initialization of CUDA is quite involved. It starts by creating a device (which should be the one we are running OpenGL on), creating a CUDA context linked with our already existing OpenGL context (through `cuGLCtxCreate(...)`), loading the compiled CUDA ptx module, getting the six kernels used in the simulation (more on that in section 5.2.2), creating all the textures used in the simulation, setting up surface refs and texture refs (what these are are described in more detail in section 5.4.3), setting up filter and borders (again explained later), and creating two renderers; one for the pressure texture and one for the editing of the geometry. As we can see this function should be split up into smaller modules. Reuse of code between the two final projects was something that should have been done from the start, as it would have saved a ton of complexity and made the code easier to read and use.

In order for a simulation to be done, we need some input audio to be loaded. This can be a simple wave file with a pulse similar to figure 45 or a complete audio sample. The function responsible for setting up all the parameters related to this simulation is called `initializeSimulation()`. It assumes that an initial impulse (or audio file) has been loaded and has the sample rate 44100. Here we allocate space for the input signal and output results. The program contains a tail parameter, which tells us how long the simulation should be run for after the input signal is done. As the buffers have two channels the final length of the buffers can be calculated using the following lines.

```
 inputBufferSize = inputFile.numSamples * numChannels = inputFile.numSamples * 2
outputBufferSize = inputFile.numSamples * numChannels + tail * sampleRate =
    inputFile.numSamples * 2 + tail * 44100
```

When the simulation is started it will run for `outputBufferSize * iterationsPerSample` number of iterations. The `iterationsPerSample` parameter is used for controlling how many iterations should be run before we write a sample to the output buffer. It is a result of the base simulation not being able to generate frequencies up to 22050 Hz, described in more details in the result section 6. As of writing, the simulator application "hardcodes" the source and destinations to specific locations, as seen in figure 23. The initialization is done whenever a new

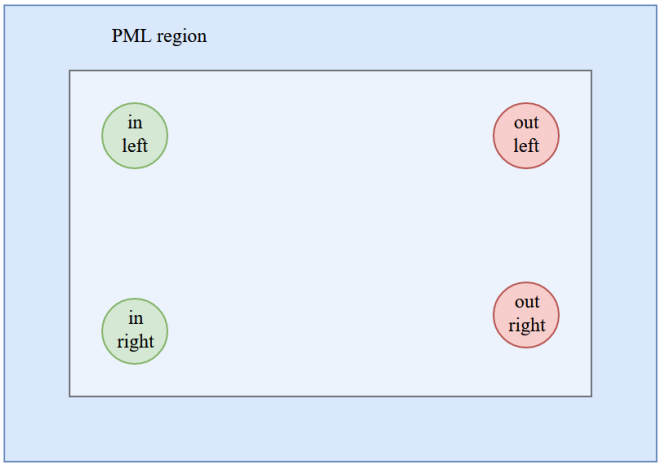

Figure 23: Source and destinations of the simulation.

input file is loaded. So, as it is right now, whenever an user wants to change the tail length or number of iterations per sample, the user has to reload the input file.

The `WaveSolver` class is also responsible for editing the geometry. It works by the user holding down the left *ctrl*-button and dragging the mouse while holding the right mouse button down. The user is then able to draw straight lines in the geometry field. While holding *ctrl* the view also changes to geometry view.

The application also contains an ImGui panel. The following UI elements are present.

- **Load**: loads an input impulse or audio file
- **Save**: saves the simulated reverberated audio to a file
- **Simulating**: toggles whether the simulation is running or not
- **Reset**: resets the simulation to the start
- **Clear geometry**: removes the currently drawn on geometry
- **Thickness**: controls how thick the walls that are being drawn should be. This is also controllable with the mouse wheel.
- **Wall absorption**: controls how much the walls absorb or reflect incoming waves. Every geometry piece has a certain falloff. With 0 wall absorption this falloff is essentially removed.
- **Tail**: controls how long the simulator should continue after the input is done

- **Amp**: amplification of the incoming signal (calculated as 10^{amp})
- **Steps per sample**: controls how many iterations should be done per sample

TextureRenderer and EditorRenderer The **TextureRenderer** is simply responsible of outputting the pressure texture to the screen. It does this by drawing a quad covering the screen and mapping the pressure values to a color lookup table. It is also reused for drawing the geometry. The **EditorRenderer** is responsible for drawing some items related to the editing of the geometry; the line that is currently being drawn and a brush.

As the application is single threaded, the simulation has to be interleaved with the drawing. This causes issues such as having to calculate the number of iterations possible per frame. A dynamic approach to this was explored, but what seemed to work best was setting the number of iterations per loop to a fixed number; 50 in this case.

5.2.2 The CUDA module

The CUDA module contains six kernels.

- **iterateVelocity**: Responsible for iterating the velocity component of the leap-frog scheme described in 3.2.2. The input impulse is baked into the force texture.
- **firstIterationVelocity**: Calculates the first timestep; $t^{0.5}$.
- **iteratePressure**: Responsible for iterating the pressure values.
- **drawLine**: This kernel draws a line to the geometry texture with a given thickness and falloff. In hindsight this functionality could have been implemented on the CPU, as was done in the convolver VST application.
- **sampleAt**: This is the kernel responsible for taking the output at some location and placing it in the output buffer.
- **sampleIn**: This kernel takes a sample input and places it onto the force texture.

Note that both `sampleAt` and `sampleIn` is only called with two threads (ie. the number of input channels and output channels) and one block. Writing from global memory is here done in a somewhat redundant way as each thread has to read from $2 * (n + 1) - 1$ cells, n being the order of the differentiation (6th order in this case). By using shared memory this can be significantly reduced by caching the neighbouring values. An example of this can be seen in section 5.4.8. The convolution is defined as a macro define. Changing the convolution order should be a matter of simply replacing the convolution defines.

5.3 Implementation of the convolver VST

The runtime convolver is implemented with the JUCE framework and is capable for running both standalone and inside a DAW as a VST. The convolver plug-in uses an open source library, made by user HiFi-LoFi [14], for doing real time audio convolutions. The final application is able to load an impulse response generated by the simulator prototype application and convolve incoming sounds with these IRs. It has a live audio level display meter and a display of the IR. It further includes a dry and a wet knob. Dry is how much of the original sound is passed to the output, while wet is how much of the processed sound is passed to the output. These signals is simply added together to produce an illusion that we are in a different space, and it works great. Most of the code is GUI related. The audio processing part is quite short, as we are using a library for applying the convolution to the incoming signal. However, here I'll highlight a few key areas of the code.

A class diagram of the final VST application can be seen in figure 24. The blue boxes are classes and components not written by me (ie. they were supplied by JUCE or from other places), yellow and orange are the wrappers around CUDA and OpenGL respectively and the green boxes are the various classes making up the plug-in. In general, on the left hand side the functionality related to simulation and audio processing lies, while on the right hand side functionality related to the graphical user interface lies.

For future work, the simulator from the final simulator application should replace the simulator in this VST version as it worked far better. In order to do so, work has to be done in regards to the threading and CUDA-OpenGL interopability, as we shall see in section 5.4.6.

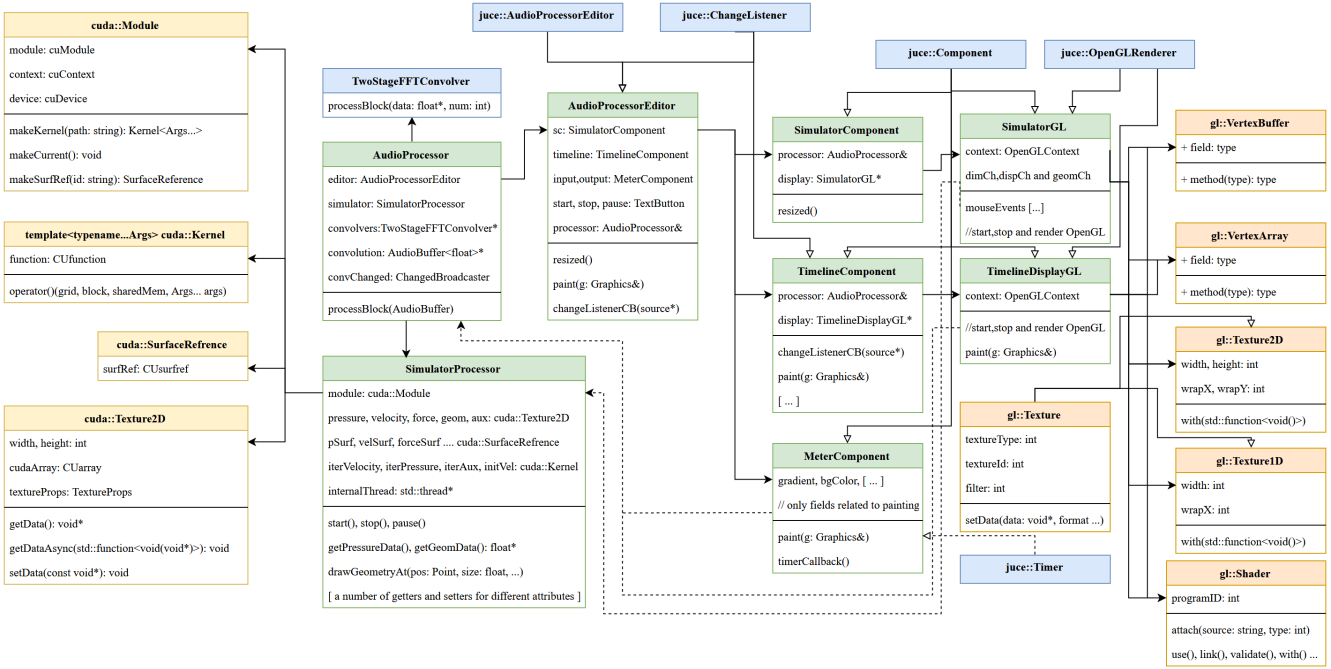

Figure 24: Class diagram of the VST.

5.3.1 JUCE overview

JUCE plug-ins typically consists of a processing part (PluginProcessor.cpp) and a GUI part (PluginEditor.cpp). The processor is passed certain data from the host, like audio buffers, MIDI buffers, parameter values etc. The plug-in then uses these values and processes the incoming audio. It is in the function `processBlock(...)` this takes place. The processor part of the plug-in is also responsible for handling the state. There is functions for both loading and saving the state, `getStateInformation(...)` and `setStateInformation(...)` respectively. Calls to the function `createEditor()`, `prepareToPlay(...)` and `releaseResources(...)` are also important in order for the VST to work correctly.

The `PluginEditor`, or rather a GUI, is not strictly required for an audio plug-in. If no editor is supplied by the plug-in, DAWs typically creates them themself based on the plug-in parameters. That is, as long as all the parameters and processing is contained within the processor-part of the plug-in, a GUI is not strictly necessary. This is also why we can't rely on a GUI being present when coding the processor. This is relevant as the editor part of the application must registers listeners to the processor, and the processor can't be guaranteed that an editor exists. JUCE GUIs consists of components in a hierarchy. All custom components inherits from a `Component`-class which handles painting, resizing, mouse and keyboard events and more. Components can consist of other components, and can include pre-built components like `TextButton`, `Slider`, `Label` etc. We, as the developer, has to manually specify the size of these components, typically done in a `resized`-callback function.

A plug-in is "shipped" by packaging it in a dynamic link library (`.dll` for VST2 and below or `.vst3` for VST3). These files are then possible to load using a DAW. In my case I am using the DAW Image-Line's FL Studio 20. The plug-in developed here is evaluated against a similar convolution plug-in from Image-Line called "Fruity Convolver". I did not bother skinning and making the application look good, as it is really only an evaluation tool. A image of the final plug-in can be seen in figure 25 and 26.

The convolver VST with the integrated simulator was not 100% completed by the end. The actual simulation and painting was completed, but it did not have any functionality for capturing the impulse responses from the simulation. However, this is fairly straight forward to implement and will be similar to what was implemented in the final simulator application.

5.4 Description of the various classes and their usage

The VST plug-in ended up having a lot of different classes with complex and subtle interactions. Here I'll list some of the classes and how they were used, as well as restrictions and demands.

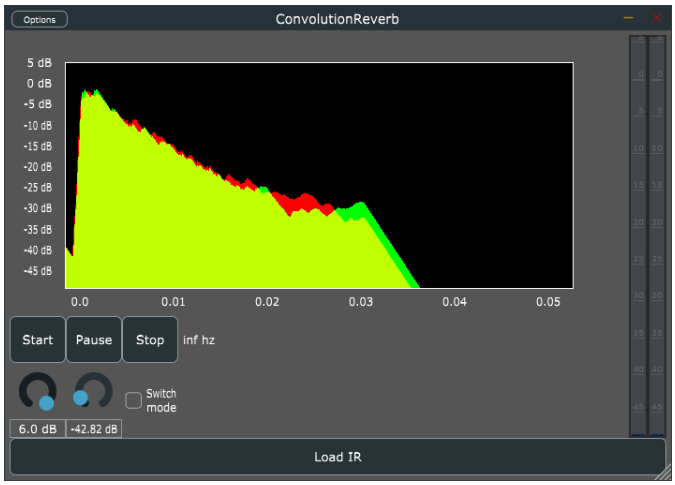

Figure 25: An image of the convolver VST plug-in with the timeline display active

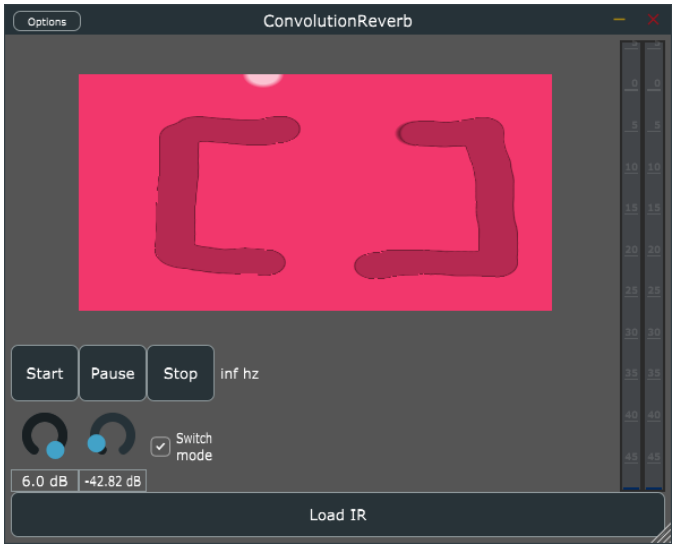

Figure 26: An image of the convolver VST plug-in with the simulation display active

5.4.1 AudioProcessor

Listing 6 shows how the convolution is applied to the incoming signal. In JUCE, buffers are passed to the plug-in with a certain number of samples. These sizes is determined by the DAW and we just have to use whatever it gives us. We can't even rely on that the buffer size stays constant. Thus we have to to a manual allocation of memory as seen in the listing. This is of course bad as it allocates and frees memory on each iteration. Further work could improve on this by allocate a certain buffer size that is guaranteed to stay larger than the input buffer when the object is initialized.

Listing 6: Processing the incoming audio

```
void ConvolutionReverbAudioProcessor::processBlock (AudioBuffer<float>& buffer,
    MidiBuffer& midiMessages) {
    [...]
        lock.enter();
    int numSamples = buffer.getNumSamples();
        float* buf = new float[numSamples];
        for (int channel = 0; channel < totalNumInputChannels; ++channel)
        {
                auto* readPointer = buffer.getReadPointer(channel);
                if (convolvers != nullptr)
                {
                        convolvers[channel].process(readPointer, buf, buffer.
                            getNumSamples());
                        buffer.addFrom(channel, 0, buf, numSamples);
                }
        }
        delete[] buf;
        lock.exit();
}
```

5.4.2 AudioProcessorEditor

The editor side (GUI) of the application (`PluginEditor`) contains a `TimelineDisplay`, a `SimulatorDisplay`, two `MeterComponent`, two sliders, a button for opening an impulse response and three buttons for controlling the simulator. The `TimelineDisplay` is responsible for showing the impulse response applied to the sound, the `SimulatorDisplay` is responsible for showing the current state of the simulation as well as drawing geometry, while the two meter components is responsible for showing the input and output decibel level, respectively. The two sliders controls the dry and wet amounts.

5.4.3 CUDA namespace

This part of the code contains all the wrappers over the base CUDA driver API types. All of the classes requires a *current* module to work.

Module A module is composed of a CUDA context, a CUDA device and a CUDA module. The context holds all the state related to the current thread. If we were to use any object belonging to a given module from a given thread, we would have to make sure the thread is current. This means that CUDA knows what context to

target for any of it's functions. In this application there exists several concurrently running threads and as such we have to make sure to make the context current when using functions from different threads. One example off this can be seen in 5.4.5. The constructor of the module takes in the path to a compiled ptx-file.

Kernel Kernel is a class for wrapping kernel functions loaded from a module. This class is written using varadic templates for better type safety; the templates is used for the arguments to the kernel. Kernels can be made from the Module class, and when a kernel is created it is important to make sure that the arguments matches the ones in the loaded module file. Further one has to make sure that the name given to the module for loading is correct. The operator `()` is overloaded, so launching a kernel simply looks like a function call, which is quite neat.

Texture2D Texture2D holds all the data related to the simulation. It is both writable and readable, and internally holds a `CUarray` we can query in order to set and get the actual GPU contents. One interesting feature about this class is that it is possible to asynchronously get the data. This ensures that getting the data does not interfere with the simulation loop. There is an internal option to query either blocking or unblocking. Blocking causes the query to stall until any kernel is done using it. This has implications for performance. Non-blocking on the other hand gets the data regardless of any kernel currently writing to it. By using this approach we get better performance, but the contents have quite noticable vertical synchronization issues. As the contents of the texture is only used for displaying, it does not really matter in our use-case, so here non-blocking is used.

SurfaceReference In CUDA (and OpenGL) there are a construct called *samplers*. A sampler is a object that makes reading from textures possible. It holds the sampling parameters for a texture access inside of a kernel. In order to read from a texture or surface reference inside of a kernel, a sampler has to be used. Texture references are used for reading texture values (from 1D, 2D or 3D coordinates) and has a few other special features inherited from the graphics world related to interpolation. Surface references are similar, but are used for both reading and writing. They do not have the interpolation features of the texture references. In our project only surface references are used. Actual textures are bound to such references, and the references represents that array (or memory object) in a kernel. One interesting feature of the newer CUDA architectures is bindless textures, introduced with the Kepler architecture. In older versions of CUDA one would have to manually bind textures and surfaces, which gives lower performance, while in newer versions references are belonging to the textures themself. This is why we take in `CUsurfObejct` in the kernels, as seen in section 5.4.5. The simulator application does not use bindless textures, but manually binds each texture refrence per iteration.

A SurfaceReference is a wrapped CUDA object that makes CUDA able to write and read from a given surface. That is any surface reference is belonging to a

Texture2D. The surface reference is made by calling a function in the Texture2D class.

5.4.4 GL namespace

One common way of using OpenGL is to write wrappers hiding the complexities of the OpenGL "state machine". For example creating OpenGL objects often involves generating an object. This object then lives until it is instructed to be destroyed. An effective way to make sure that this is done for every OpenGL object is to implement the generate part in the constructor of the wrapper, and the destroy part in the destructor. This is what has been done for this project. One important point to stress is that these generation and destruction functions would crash if we do not have a valid OpenGL context. That is why most of the classes below is only used as pointers. By using pointers we can explicitly ensure that the constructors and destructors are called when we have a valid context.

Many of the different OpenGL objects require that they are currently "used" in order to set its values and use them for their intended purpose. That goes for vertex buffers, vertex arrays, shaders and textures. The way I have handled how they are bound and unbound is by having a function called `with(std::function<void()>)` which takes in another function. This function passed as an argument can then be absolutely sure that the current vertex buffer or shader (...) is bound and ready for use [8]. Further, we do not have to know the detail of how or when objects are bound and unbound.

VertexBuffer and VertexArray In OpenGL there exists several different specialized memory objects. One of these is vertex buffers. They hold data about attributes like position, color, texture coordinates etc. Further, there exists index buffers which holds information about what vertices makes up a triangle or line or points. In this project vertex buffers are primarily used for drawing a quad covering the entire display, so that we can draw a texture to the screen. A VertexArray holds a set of related buffers and vertex pointers which together defines one drawable primitive. Vertex pointers defines how OpenGL should handle the contents of a given buffer. For example a vertex buffer can contain 3D positions. A vertex pointer then tells OpenGL that each vertex consists of three floating point values, and each vertex have a given stride and offset between them. Buffers can also be interleaved, meaning that positions and texture coordinates can exists in the same buffers. VertexArrays makes it possible to define these layouts once, and then not having to deal with it again when rendering.

Texture, Texture1D and Texture2D The different texture classes represent the textures in OpenGL. Texture1D and Texture2D inherits from the abstract Texture class. Both of them is required to implement a function for setting their data.

[8] This is not strictly true here. For example, if we tried to use `with` on two nested shaders, one would bind over the other. This is not a problem as long as we are careful with their usage.

Confusingly both the CUDA namespace and this namespace contains Texture2D, but they are not the same and works quite differently.

One obvious problem with the `with()`-functions described above is that we would have to nest several abstract functions in order to use several textures. This obviously looks ugly and is hard to maintain. In order to solve this there is a static method called `withMultiple()` which takes in several textures and binds them to different texture slots. This has the advantage of clean and correct code. Preferably this would be extended to the shaders and vertex arrays/buffers as well, but is not really a concern for this project.

Shader A shader program is a small program describing how a vertex should be displayed on the screen. A description can be found in section 4.4.1. Here, a wrapper over the raw OpenGL calls is created. In order to have a correctly functioning shader program a few steps has to be taken. In listing 7 the construction of a shader program can be seen.

Listing 7: Creation of a shader

```
shader = std::make_unique<Shader>();
shader->attach(vertexShaderSource, GL_VERTEX_SHADER);
shader->attach(geometryShaderSource, GL_GEOMETRY_SHADER);
shader->attach(fragmentShaderSource, GL_FRAGMENT_SHADER);
shader->link();
shader->validate();
```

As we can see, the source code of the different shader types has to be attached, then linked and finally validated. If the source code of any of the different shaders is incorrect the application will crash. Shader programs can also take in values called *uniforms*. A uniform is a global shader variable declared with the "uniform" storage qualifier, and any uniform has a unique location. These locations is queried in the Shader wrapper and stored for later usage. The programmer is able to upload these values by using the different `glUniform***(...)`-functions.

5.4.5 SimulatorProcessor

This is the class that controls the simulator. It has a `cuda::Module`, a couple of `cuda::Texture2D` and a couple of `cuda::Kernel`. An important point to stress is that the CUDA objects has to be created *after* the module has been created and made current. The class contains textures for pressure, velocity, forces, geometry as well as auxiliary values for the PML. Further, it has four kernels; one for iterating the pressure values, one for iterating the auxiliary values and two for velocity. The first one called `initialVelocity` is split into it's own as we want to use the force texture as an impulse. The force is not required beyond the first iteration, so we have another velocity kernel for further iteration.

The simulation runs on it's own thread. This ensures that the performance is consistent and not dependent on what else is happening in the application. However, we have to take some steps in order to ensure that we don't end up with

race conditions or deadlocks. The core simulation loop runs in a while-loop and checks a condition on each iteration. The core loop can be seen in listing 8.

Listing 8: Core simulation loop

```
while(simulationState == Started)
{
        CUDA_D((*iteratePressureKernel)(grid, block, sharedMemSize * sizeof(
            float2), pressureSurfObject, velocitySurfObject, auxSurfObject,
            timeStep));
        CUDA_D((*iterateAuxKernel)(grid, block, sharedMemSize * sizeof(float2),
            auxSurfObject, velocitySurfObject, pressureSurfObject, timeStep));
        CUDA_D((*iterateVelocityKernel)(grid, block, sharedMemSize * sizeof(
            float), velocitySurfObject, pressureSurfObject, geometrySurfObject,
             timeStep));
}
```

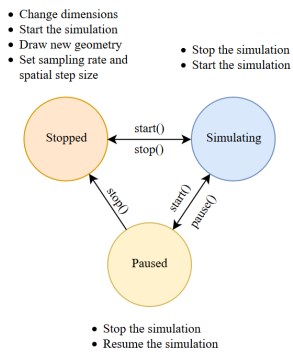

Figure 27: Simulation states and allowed modification to the states

When the simulation is requested to stop or pause, the calling thread will set the `simulationState` variable to `Stopped` or `Paused` and then wait until the simulation thread has completed. This is accomplished with a thread join call. By the time the calling thread continues, it can be sure that the simulation thread is not running. Extensive checking of the simulation state is done anywhere where functions that modifies the object state is done. For example, modifying the dimensions of the domain obviously should result in an error while the simulation is running. The allowed functions during the different simulation states can be seen in figure 27. Any modification not in the list will result in the functions throwing an error. The simulator processor does not create any CUDA textures until the simulation is requested to start. On stopping it also deletes all the texture resources associated with the simulation. This is done in order to be able to change the dimensions and other attributes while the simulation is stopped, but does cause some overhead when starting and stopping the simulation. The kernels and module does however exist during the lifetime of the class.

The SimulatorProcessor queries the GPU for the pressure texture at a fixed rate (currently 10 Hz). It does so asynchronously and broadcasts a change message when the results are ready. In fact, a lot of the dynamic events in the simulator broadcasts change messages that other objects can listen to. This ensures that the simulator can keep a consistent state, and other listeners only have to react to something happening (as opposed to continually checking if some state has changed). The dimensions changing and change of simulation state is events other

objects can listen for. One subtle effect of changing the state of the simulation is that a state change takes some time. Using state change broadcasters and listeners ensures that state change can happen asynchronously.

As described in section 3.2.2, the time stepping scheme requires that the velocity and pressure calculations are interleaved. This implies that we need two kernels (at least) for calculating the values, because the next time step requires the values of the previous half-step. As a consequence we have to launch twice the amount of kernels as opposed to combining the two kernels into one as we could have done with a regular forward Euler scheme. The benefits of a leap-frog scheme outweighs this loss in performance however, as described in section 3.

5.4.6 Custom components

Some custom components was developed for this application. The SimulatorComponent and SimulatorGL is for displaying the current pressure values for the simulator as well as editing the geometry. TimelineComponent and TimelineDisplayGL is for showing the current convolution applied to the audio signal. There is also a switch for toggling between the simulator display and the timeline display.

SimulatorComponent and SimulatorGL These two components are responsible for showing the current simulator state. SimulatorComponent is really a wrapper. It's main purpose is to draw the axes of the SimulatorGL component. SimulatorGL however, is the one that is responsible for both showing the simulator state and making editing the geometry possible. It inherits from both the JUCE OpenGLRenderer and JUCE component. OpenGLRenderer enables components to render OpenGL on a background thread, and every virtual function here (`newOpenGLContextCreated()`, `renderOpenGL()` and `openGLContextClosing()`) are called from said thread. One important consequence of how JUCE works is that a context is created and destroyed only when the component it is belonging to is visible on the screen. This implies that we can't use the CUDA-OpenGL interopability in the same way as we did in the simulator application. This is because CUDA OpenGL interopability is one-way from OpenGL to CUDA, ie. OpenGL creates and "owns" resources and buffers and passes them to CUDA. The CUDA simulator is intended to live for the duration of the application, while the OpenGL context only lives while the display is visible. Further, they run in different threads, which have implications of how a potential single context would work. The simulator application is single threaded and only contains a single OpenGL context. Thus interopability is much easier there. The way interopability is solved here is to have the simulator thread broadcast a change message each time the pressure values are read back, as described in section 5.4.5. The SimulatorGL component listens for this event and writes the pressure values from the simulator to an OpenGL texture. Another point to stress is that the OpenGL context can only live in one thread (the one created by OpenGLRenderer), so we have to explicitly call a function called `executeOnGLThread` belonging to the context object. This all works okay, however screen tearing due to the non-blocking asynchronous call is an issue. The renderer

is also set to not continuously render it's content (the default behaviour) and only redraw when something happens.

The simulator display also contains functionality for "editing" the geometry. Currently the user have to paint using the mouse in order to edit the geometry that is sent to the GPU. The user is supplied with a brush, which is it possible to change the size, falloff and amount; similar to what can be found in Photoshop or GIMP. If a wall is painted with a higher falloff, more of the energy will be absorbed into the wall, similar to how PML works. Ideally, the painting should work in the same way as it does in the final simulator application, however due to time constraint this sort of brush painting was what ended up in the VST.

TimelineComponent and TimelineDisplayGL

Similarly to the simulator display, the timeline display contains an OpenGL context. TimelineDisplayGL's job is to render the current convolution buffer we are applying to our sound. It does this by listening for the convolution buffer changed broadcaster in the AudioProcessor class. Once the buffer changes it grabs the sample data and transforms it into a dB scale. Other than that, the timeline display works in much the same way the simulator display does, apart from not having any interacting elements. The TimelineComponent also draws some axis values.

MeterComponent The meter component is is responsible to show the input and output levels. It does this continually through a Timer. A `juce::Timer` provides a virtual function which is continuously called at a fixed frequency. By implementing this function we get the input and output levels from the AudioProcessor.

5.4.7 Other

A Label implementing the `juce::Timer` called `TimedLabel` is used for displaying the frequency the simulator runs at. The SimulatorProcessor keeps tab of how fast it is currently running.

TwoStageFFTConvolver This is a class provided from the github repository [14], and makes it possible to generate a convolution output given an impule response. The AudioProcessor keeps two of these for left and right channels.

5.4.8 The CUDA module

The CUDA module can be found in `solver.cu` and contains heavily templated functionality. The general idea in the iteration kernels is to write any values that needs to be read by several threads into shared memory. This has the benefit of reducing texture fetches from global memory. Each thread in a block is responsible for one cell in the leapfrog scheme. One divergence or gradient calculation involves

looking up n neighbouring cells in top, left, bottom and right directions (n being the order of the derivative stencils). As each thread is responsible for one cell we can reduce these $4 * n$ extra global memory look-ups by placing the content of each cell in the shared memory. Then each cell only has to check the shared memory for the neighbouring values. As each block contains 32 threads in each direction we also have to write n values from the top, right, bottom and left blocks. This is what the `P` template value is used for in all of the texture look-ups. The function that writes to the shared memory is called `writeSummedToSharedMem(...)` and takes in an arbitrary number of surfaces and simply adds them together. After a write has been done, we have to synchronize the threads, as we are not guaranteed that every thread in the block has written to the shared memory once we proceed. Similarly to the write function, we have a read function that is able to read for memory. Again, templates is heavily used in this code. It doesn't matter what the type the shared memory has or how much padding there is; the compiler will figure it out for us.

The code is written in such a way that the convolutional stencils easily can be exchanged. In `Convolution.h` the convolution stencils are defined. It allows for 2nd, 4th and 6th order derivatives, and the stencils are generated from the Python code described in section 3.4.1. This makes testing the efficiency and results of different differentiation orders very easy, as changing the order only involves changing one define macro. Gradient and divergence are both implemented using the same derivative functions. The PML values is calculated by a simple function. Inside the PML region the values rises from 0 to `pmlMax`, depending on how far the cell is from the actual domain.

This module contains four kernels that is loaded by SimulatorProcessor as kernels. They more or less directly implement the integration steps found in section 3.3.2. The only difference is that the two auxiliary fields are combined into a two-channel texture.

5.5 Notes about debugging CUDA and OpenGL

It's hard. Which is why every CUDA and OpenGL call is surrounded by the macros `CUDA(...)`, `CUDA_D(...)` and `GL(...)` for the CUDA runtime API, CUDA driver API and OpenGL respectively. These macros typically first executes whatever is inside the macro, then checks if any errors occurred, and if so throws an appropriate error. These checks can easily be turned of when compiling for release, as they tend to degrade performance a bit. This has been a crucial in order to work with CUDA and OpenGL, as all these APIs typically happily will continue on as if nothing wrong has happened.

The different macros had to be defined a bit differently. For example, the CUDA runtime API mostly returns an error code for each of it's function. This error code can them be checked against the enum `cudaSuccess`. The same goes for the driver

API. OpenGL works differently. Most functions returns `void` and the way to catch errors is to explicitly check if an error occurred directly after that call. If this is not done for *every* OpenGL call, a previous error may erroneously show up here. When we are confident that our code works correctly (ie. gives no runtime errors) we can turn off all these error checks for the release-build. Some examples can be seen in listing 9. Note that from OpenGL version 4.3 and on-wards, proper error handling has been introduced through callbacks (`glDebugMessageCallback(...)`), negating the need for this convoluted error handling mechanism. However not all implementations of OpenGL supports version 4.3 yet.

Listing 9: Return codes for different API functions

```
// CUDA runtime API
cudaError_t cudaMalloc(void** devPtr, size_t size);
cudaError_t cudaPeekAtLastError(void);

// CUDA driver API
CUresult cuArrayCreate(CUarray* pHandle, const CUDA_ARRAY_DESCRIPTOR*
    pAllocateArray);

// OpenGL
void glGenTextures(GLsizei n, GLuint* textures);
GLenum glGetError(void);

#ifdef _DEBUG
#define GL(x) x; if (int err = glGetError() != 0) {
    /* throw an error with an appropriate error string here */
}
#else
#define GL(x) x;
#endif
```

6 Results

This chapter discusses some of the simulation results obtained, as well as the efficiency of the final applications. Some discussions regarding the early builds are also present here.

6.1 Early results

The early versions of this project produced some less than satisfactory results. The two early versions, the basic CUDA version and the OpenCL version, was scrapped. From these two projects I had learnt a lot of how to efficiently employ CUDA in conjunction with the JUCE framework and OpenGL. The OpenCL VST can be seen in figure 28, and the code can be found in the `ReverbSimulator` repository. It only contained the very simple simulator found in section 5.1.1, and had no way of capturing the output of the simulation or drawing geometry. It did contain a lot of the OpenGL techniques that ended up in the final simulator, such as the LUT techniques, the drawing of a quad as a texture and the first version of the shared class can be found here.

Unfortunately, the early CUDA VST version stopped compiling properly [9], so the results from that iteration is hard to obtain. However the code exists in the `ConvolutionReverb2` repository, and by tweaking the build options or replacing it, it should be possible to get it back up and running again. This version of this project included VST plugin with a built in simulator. It was capable of loading in a `png` image and use that as geometry and generate a waveform out. The plugin was very primitive and used only the CUDA runtime API for interfacing with the GPU. This ment using only using pitched arrays and manual array indexing, which implies that the GPU was not able to exploit spacial locality in the same way a 2D texture would. The waveforms generated by the plugin was in very poor quality as the time steps was too low, resulting in frequencies only up to around 2kHz. Further, there was no absorption on the boundary (ie. no perfectly matched layer), so the waves continually bounced around in the room "forever". There was however introduced a dissipation constant, so that the waves lost energy as time went on. This was done by simply multiplying the pressure by some constant at each time step.

6.2 Python results

Python was used as a prototyping and testing ground, and a lot of interesting results was obtained here before they were further implemented with C++ and CUDA. The analytic solver was attempted to implement here; for example `interface.py` and `boundaryTest.py` was an attempt to implement the boundary interfacing from [1], but implementing boundary exchange across arbitrary directions and PML for avoiding the waves to reflect back inside a boundary turned

[9] The Visual Studio Build System for CUDA stopped working halfway through, which is why I had to create a custom build system for the final simulator application

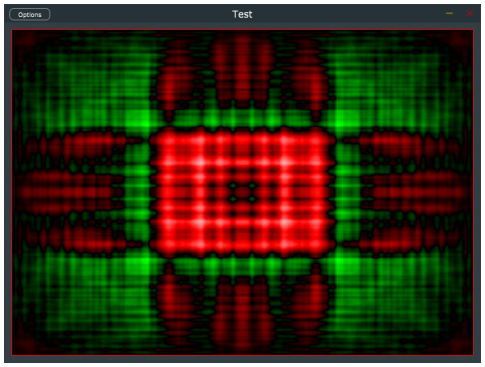

Figure 28: A screenshot of the earliest simulator

out to be too hard. Other files in this repository includes the leap-frog integrator (`twostepleapfrog.py`) which also includes an implementation of the PML absorber. While the PML approach with auxiliary PDEs worked great here, it did not work correctly in the corresponding CUDA implementation (from the final convolver VST). This is because of numerical instability caused the PML regions to grow it's values exponentially. The most likely source for this error is that the PML calculation in the Python versions were updated at a different time point (ie. after the velocity update) compared to the CUDA version (before the velocity update due to implementation difficulties and order of update rules).

6.3 Results from the first iteration of the simulator application

The first iteration of the new solution had absorbing boundaries, editable geometry and included the possibility to save and load wav files. However, the maximum frequency simulated was too low and the numerical dispersion was too high. But the results did seem promising for further iteration. The simulation managed to simulate 44100 samples in 15 seconds, or about 1:15 slower than real time, for a grid of size 589 * 295. Some of the results with corresponding spectrogram can be seen below. Figure 29 and the other similar ones shows the pressure values during simulation colorized from black to white with blue corresponding to no pressure. Figure 30 and the other similar ones shows the frequency content as a spectrogram. The x-axis corresponds to time, while the y-axis corresponds to frequency. The yellow parts corresponds to loud frequencies, red is clearly audible, blue are barely audible while black corresponds to no sound. The spectrogram ranges from 0Hz to 22.5 kHz along the y-axis.

Clearly the maximum simulated frequency is too low. These examples only have a frequency up to 12.5kHz, while the desired frequency is around 22kHz. Further there is clearly visible numerical dispersion errors at 8.3 kHz and 12.5 kHz. While

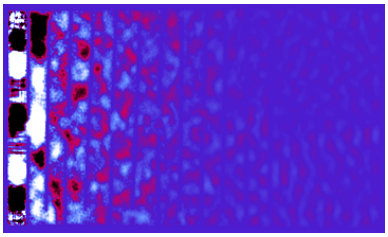

Figure 29: Wavefield

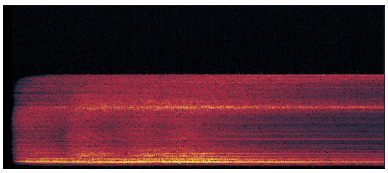

Figure 30: Spectrogram

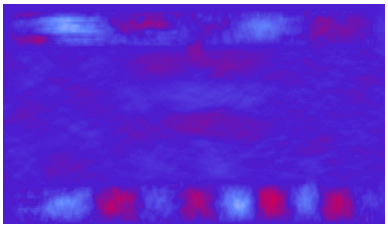

Figure 31: Wavefield

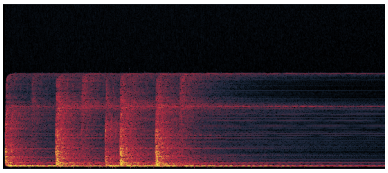

Figure 32: Spectrogram

numerical dispersion in FDTD simulation is impossible to completely get rid of, we can however simulate with smaller time steps. This does however require a lot more processing time.

On the upside, the simulation sounded interesting. For example by creating an enclosed area around the sources and creating small air holes, the resulting wave file sounded like audio source had been played through a tube, as in figure 33. Further, I was able to create a flute-like effect, when using a very thin tube the source went through and placing several air holes on top of the tube.

6.4 Final results from the simulator application

As the preceding results had rather devastating audible artifacts, adjustments had to be made. By doing multiple steps per resulting time step I was able to generate some more promising results, as seen in figure 38. These results were generated with 8 steps per time step. There still exists some artifacts as seen by the smearing of frequencies at around 20 kHz and to a lesser extent 17 kHz. These frequencies was however hardly audible, and a quick fix for this would be to low-pass these frequencies in post processing as most impulse responses hardly contain useful data at those frequencies.

Below, in table 1, we can see how the final simulator application performed for different block sizes, dimensions and number of steps per iteration. The higher the step per iteration is, the higher frequencies the simulator is able to simulate. 8 steps proved to be sufficient for getting frequencies up to 20.5 kHz without too

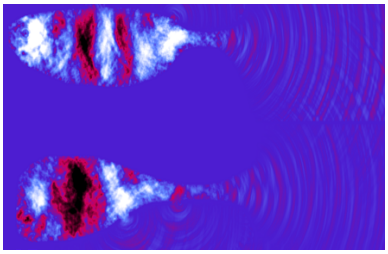

Figure 33: Wavefield

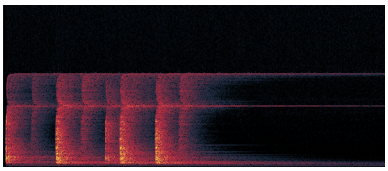

Figure 34: Spectrogram

Figure 35: Input geometry

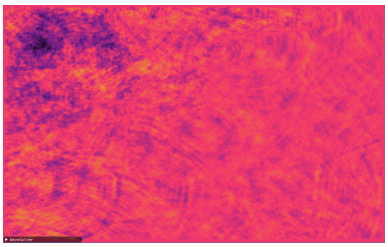

Figure 36: The simulation during execution

much audible artifacts. We are able to extract some interesting information from this table. For example, changing the block size does not have any significant impact on performance. Larger block size gives slightly worse performance. I think this is because a block can only execute 32 threads simultaneously (in a thread warp). Increasing the number of threads in a block will then result in more thread scheduling within a block. It will however also lower the number of blocks in total. So perhaps the block scheduling is more efficient than the thread scheduling, at least in this case.

All the values in the table were generated using a PML layer size of 20 on either sides, but the size of the PML layer should not have any significant impact on performance. This is because we don't store the PML values in a field, but rather calculate them based on the cell position. There is a significant amount of overhead in this application per frame. Table 1 was generated using a screen size of 700 * 700 pixels. Making this value smaller might have reduced the overhead. Further the ImGui library is also drawn each frame. ImGui is not optimized for performance, but for the comparison of values in the table, we can assume that ImGui takes the same amount of time for any configuration of parameters. Thus it probably does not have any significant impact on the results. Most of the overhead comes from the `glfwSwapBuffers()` in the `Program` class. This is to be expected as it is this function that forces the actual drawing to happen. Most GPU drivers using OpenGL usually defers execution of the commands we give it until a swap-buffer occurs. For most of the simulations, I achieved a frame rate of about 30 fps, with

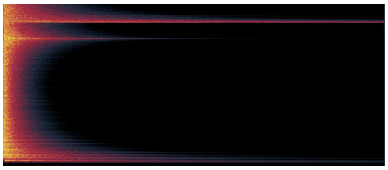

Figure 37: Output spectrogram

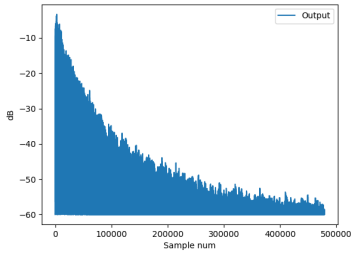

Figure 38: Output pressure over time

a target of 60 fps. The target is used for the simulator for calculating how many iterations can be done each frame. So the simulator squeezed in a certain number of iterations in one frame and we still got 30 fps given a target of 60 fps. This implies that we have about 50% overhead per frame, going to tasks like drawing, input polling and other related tasks.

Table 1: Iterations per second for the final simulator application

6th order simulation				
Block size	**Domain size**	**1 step per iteration**	**4 step per iteration**	**8 step per iteration**
8,8	100,100	54421 Hz	16457 Hz	8399 Hz
	200,200	34669 Hz	9647 Hz	4949 Hz
	500,500	8031 Hz	2031 Hz	1018 Hz
	1000,1000	2355 Hz	566 Hz	247 Hz
	2000,2000	581 Hz	132 Hz	63 Hz
16,16	100,100	51506 Hz	15190 Hz	7873 Hz
	200,200	33017 Hz	9178 Hz	4700 Hz
	500,500	7947 Hz	2046 Hz	979 Hz
	1000,1000	2220 Hz	549 Hz	246 Hz
	2000,2000	585 Hz	136 Hz	62 Hz
32,32	100,100	48460 Hz	14339 Hz	7300 Hz
	200,200	27863 Hz	7613 Hz	3774 Hz
	500,500	6745 Hz	1724 Hz	846 Hz
	1000,1000	1890 Hz	457 Hz	203 Hz
	2000,2000	499 Hz	122 Hz	59 Hz

6.5 The convolver VST application

While the convolver VST application was not completely done by the end of the project, it's main function ended up working quite great, thanks to the *Hifi-Lofi* FFT convolver library [14]. This library enabled me to supply an impulse response

file and processing blocks of audio in order to apply the effect. I had to use two of these instances, one for the left channel and one for the right. There did not appear to be any significant cost in terms of efficiency to using the library.

In figure 43 we can see the resultant audio given the geometry in figure 41. At 6:40 we can see the simulated audio given the drum loop at 0:00. Whats interesting here is that the simulated audio is quite similar to the source convolved using the IR (at 18:50) which is as expected. Further, the convolved signal is almost the exact same as the reference convolver VST (Image-line's *Fruity Convolver*). The simulator is able to simulate audio across the whole 0 Hz to 22.5k Hz, which was the target frequency. There are little artifacts, the only audible ones are at 17 kHz and 19 kHz. This boost in gain at those frequencies might be a result of the geometry, however that is highly unlikely as we see the same boosts at similar frequencies in figure 40 and 37 as well. The artifact is however greatly reduced here, as compared with the results in 6.3, and a simple low-pass after the simulation could hide some of these frequencies. In figure 43 we also can see that the tail is not long enough. This results in unrealistic cutoff of the reverberation. By simulating for longer, this could have been avoided.

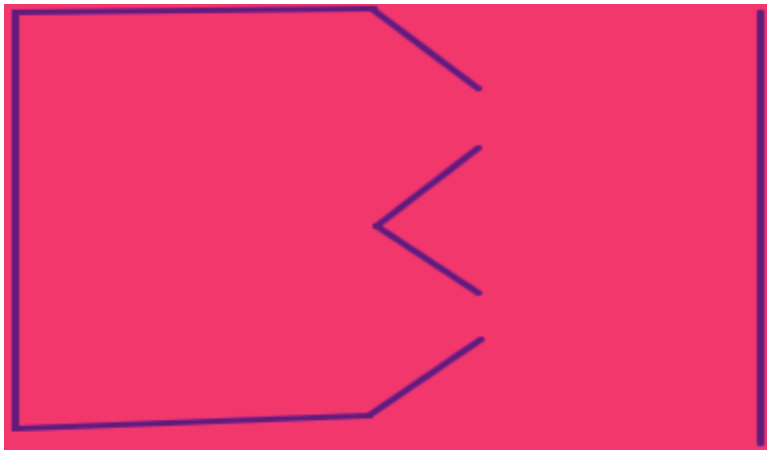

Figure 39: Input geometry

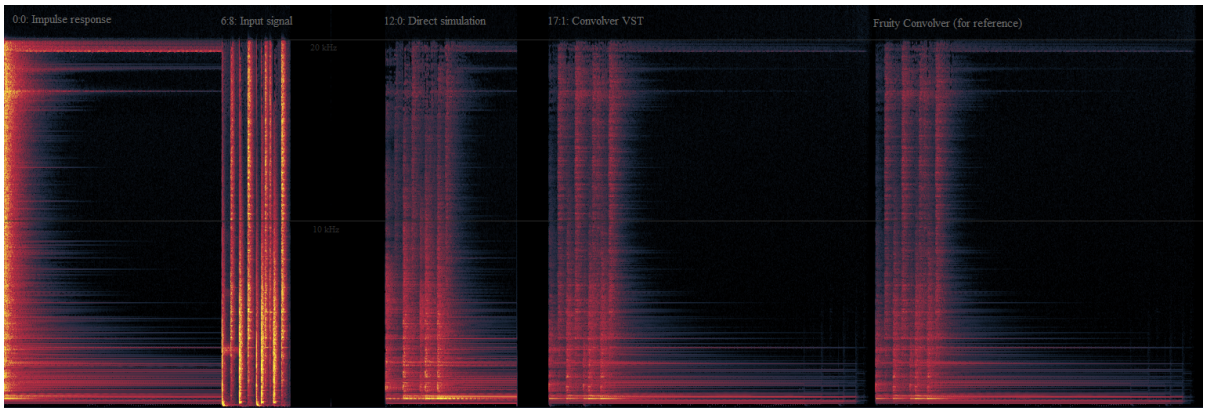

Figure 40: Output spectrogram (`https://www.dropbox.com/s/w17840o5iyaf061/res.1.1.mp3?dl=0`)

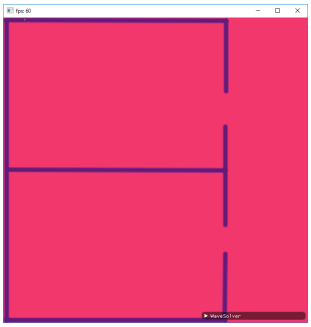

Figure 41: Input geometry

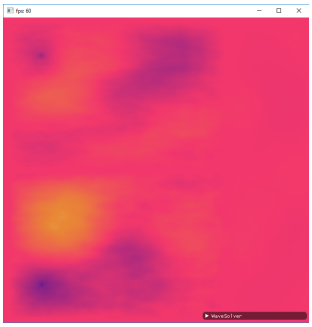

Figure 42: Application during simulation

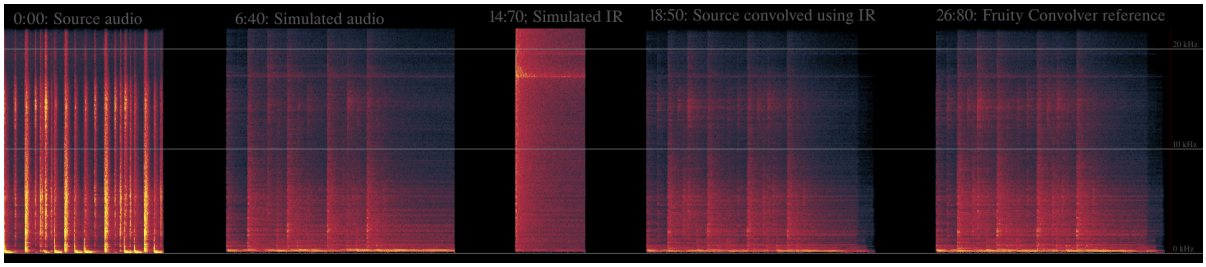

Figure 43: Output spectrogram (`https://www.dropbox.com/s/8tjgekws7vfnoig/res.1.4.mp3?dl=0`)

6.6 Notes on the other attempted methods

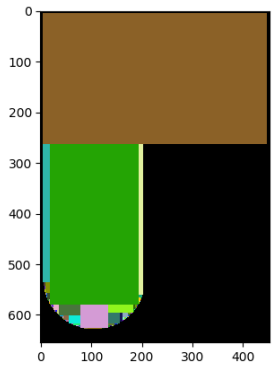

Figure 44: Partition of a 2D scene

The most promising method among the proposed methods was the one described in [1]. This method, as mentioned earlier, uses the analytic solution of the wave equation in order to simulate acoustics with almost no artifacts. The analytic solution itself was not too hard to implement, but harder to understand than the other methods. The approach consisted of several steps. The pre-processing part consisted of loading the geometry from an image and then dividing the geometry into disjoint regions. The algorithm started by picking a random point in the grid and then expanding it until further expansion was not possible. After this the algorithm picked a new uncovered point and expanded this. This was repeated until the entire domain was covered. An example can be seen in figure 44.

After this pre-processing had been done, the actual simulation could begin. Given an initial impulse that was transformed into frequency domain, the method continually updated the values for each partition. The simulation loop consisted of a simple forward Euler scheme. When in frequency domain, one large advantage is that the Laplacian operator is possible to implement by a simple multiplication (as opposed to a convolution). This obviously increases performance by quite a bit, but the border exchange and external forces had to be transformed with a FFT anyway. One of the main reasons this method did not work out was the border exchange. It consist of calculating the continuation of the equation at the, for example, left hand side of an interface. Then putting that continuing force in the partition to the right. As the method uses cosines for it's basis function, the method assumes an even extension at the interface. This means transforming the convolution kernel in such a way that the sound wave propagates into the next partition. This is described in more detail in [1]. This had to be separately done for 4 different direction, where I only successfully managed to do this for one of the direction. In addition to this the outgoing waves inside the given region had to be absorbed by a perfectly matching layer. This was probably the part that resulted in me giving up on the method. I did however successfully implement a perfectly matched layer in the FDTD method, while this was for the analytic solution. The method in the paper further included a peak extractor and generation of high frequency data, which reduced both the memory footprint and computation time.

This might have been the ideal way to solve this problem as the dispersion problems would greatly have been eliminated (except at the boundaries). Further it would have been the most accurate. However, the mathematical formulations and

requirements was very hard to understand. Given more time and a concentrated effort, a better result might have come from this.

7 Conclusion

This section describes some of the final conclusions that can be drawn from this thesis, whether the goals of the project was met or not, some retrospective comments on how the project went and possible future work.

7.1 Results

As we have seen in this thesis, numerical simulation with FDTD methods is quite challenging and involves quite a large amount of mathematical background. One of the main challenges of this project was to obtain such knowledge while also learning both CUDA and the JUCE framework. The FDTD methods employed in this thesis have quite a few unintended artifacts; numerical dispersion being one of the most severe ones. Dealing with such errors while having a usable simulation speed was not really achieved. Further, best practices for CUDA and ways to write efficient code was hard as the CUDA API almost feels like a black box. For example, many of the returns from erroneous calls with the structures used for the different functions (ie. `cuMemcpy3D` using `CUDA_MEMCPY3D`-structure [15]) was somewhat difficult to interpret and often resulted in trying things until it worked. I will say however that a lot of the errors probably was not research in the right way from my part. One issue I tended to forget about was that the structures were not zero-initialized, and thus having arbitrary values for the fields not used.

While easy to use and relatively straight forward, the JUCE framework contained a few peculiarities I wouldn't really want to have in a framework. It seemed that the JUCE developers have reimplemented (or created wrappers for) a lot of stuff found in the standard libraries. For example; the JUCE framework makes extensive use of their own `String`-type, threads are reimplemented (if they use the standard library under the hood is unknown to me), mutex locks are reimplemented and so on. Writing messages to the console initially appears to not work as the messages are directed to the immediate output, and integrating existing project using `printf` or `std::out` requires quite a bit of rewrite. Their OpenGL implementation was, in my opinion, horrendous. The way it worked was by having an OpenGL context object with a public member called `extensions`. The different functions from OpenGL was members of the `extensions`-object, but a very large amount of the functions I needed was not part of this object. Thus I initially had to manually load function pointers, but this quickly turns into quite messy code. The solution I ended up with was completely ditching the `extensions`-object and loading OpenGL through GLAD; a little redundant (`extensions` load some function pointers, and GLAD then loads them again), but turned a nightmare in merging the different projects together into something more maintainable. For example, for the VST I essentially would have had to rewrite the entire shader class and pass the `OpenGLContext` around. In addition to this, the JUCE framework contains a lot of auto-generated code and macro switches, which I was not interested in using. Replacing the macros with constant values (ie. I know the application is going to be an effect unit, I

don't need the ability to change this through the ProJucer) and never opening the ProJucer (the program responsible for the auto-generated code) became the norm.

While I'm happy that I got some useful results, the quality of said results were not as good as originally envisioned. The simulator application were quite able to produce some interesting results using odd geometry (think tubes with blow-holes, flutes or similar), but the maximum simulated frequency was to low. In order to remedy this, resolution could be increased by doing several iterations per sample, but then efficiency became too bad. The original goal was to be able to do acoustic simulation in near realtime, but I was only really able to simulate the pressure field at around 1000 samples per second (1.0 kHz) for a 1000 x 1000 grid. The runtime convolver application was however quite usable if one were to load impulse responses and then convolving the incoming audio. No undesirable effects due to the convolver could be heard, and the convolver produced quite satisfactory results compared to the reference reverberation effect *FL Convolver* from FL Studio 20.

7.2 Goals

The initial goal was to create something similar to [1] for use within a digital audio workstation, specifically a VST. While an unfinished VST was produced, the quality of the simulation was far from the same as [1]. One of the main issues for me in this project was that [1] and a lot of the other papers are produced by people with background in acoustics. The intended expertise I intended to bring was writing the code as efficient as possible using CUDA. However, this was not what happened. A lot of time was spent struggling to get CUDA or some contruct in C++ to work right. For a given task in CUDA to be done, a lot of boilerplate code has to be working properly, debugging an error in such boilerplate code can be quite hard. Examples of this can be found in the final simulator application, where a lot of CUDA driver API calls are used directly. By the time of writing the VST application, I knew enough of the driver API in order to write working wrapper classes.

I still think it should be possible to integrate a simulator like [1] into a VST plug-in. Some of the methods and techniques for dealing with acoustic simulation in an efficient manner was discovered too late in this project. For example, it is common to split a reverberation effect into two parts; early reflection and late reflection. The early reflection accounts for the echos that hits the listener back first and is the part of the reverberation that makes the impulse response the most unique. The late reflection accounts for the longer-lasting tail of reflection, where the individual reflections are indistinguishable from each other. In most practical cases the late reflection can be parameterized by the decay rate, the amount of high frequency content and how fast that high frequency content decay, seen in figure 45 (possibly more parameters depending on the use case and realism). This late reflection is then possible to generate using simpler techniques such as Schroeder Reverberators [11] and thus ditching a huge portion of the simulation. This would

probably be the best way to create such a VST, if I had to do it again (or if I am to take this project further).

7.3 Retrospective

7.3.1 What went right

By the end I had actual result from the simulation. This was for a long time not the case, as dealing with a quite low-level API like the CUDA driver API is not easy. Further, looking up the CUDA driver API on, for example, StackOverflow does not result in many hits. The audible simulation results is somewhat usable, where the user can create a simulation in the simulator application and then load the resulting impulse response in the convolver application.

A lot of the CUDA code ended up looking quite elegant by the end, and no functional errors, crashes or memory leaks occurred. For example, by using templated classes launching a kernel simply looks like a function call. The threading in the convolver application worked quite well and did not result in any deadlocks or race conditions. The efficiency of the main audio processing loop also worked quite well.

7.3.2 What went wrong

The main problem in this project has been the incredibly steep learning curve, and trying to learn quite a bit of a very advanced mathematical field. Thus a lot of expectation management had to be done by the end. The efficiency of the simulator application by the end were far from satisfactory, which has been one of the goals of this project. While usable, the results from the simulator applications contained audible artifacts and a too low frequency (given the simulation time). The two main technologies I had to learn for the duration of this project (CUDA and JUCE) took long time and required quite a bit of patience and motivation in order to master. Further, the alternative technologies and techniques explored had to be learned as well (most notably OpenCL and BLAS), taking up a significant amount of time. Fortunately, I was quite comfortable with OpenGL, Python and C++ so this went better. However, I had not used C++ templates before in the same way done for this project, so that had to be learned as well.

7.3.3 What could have been done better

More planning and research before the final implementation of both the simulator and the VST could have been done. For example it is quite disappointing that the ER/LR split was not thought of earlier, as it would have greatly increased the quality of the results obtained here. Consulting outside expertise was also something I didn't really do. Asking for help with the CUDA code and the acoustical / mathematical background theory probably would have helped a lot.

7.4 Further work

7.4.1 Finishing the convolver VST

As it stands right now, the convolver application was not completely finished. The most notable functionality missing includes writing the simulation results back from the simulator part and using it as an impulse response and being able to display, edit and add/remove source and destination locations. Further there is some steps that could have been made in order to increase performance. A new feature of CUDA 10.0 called *CUDA Graphs* were explored, but not implemented. Graphs makes it possible to define a computation graph and have the kernels automatically called. This would remove the need for having a simulation loop where we manually call the kernels, and thus possibly increase performance by not having the CPU interfere at all. It is unclear at this point if it would bring greater efficiency to the simulation, but the problem in this thesis could be well suited for such an approach (as we can seen in figure 5 from section 3.2.2 we could define a graph somewhat like what is displayed here).

As the VST simulation stands right now, there seems to be some problems with how the auxiliary PDEs work, some values gives rise to numerical instability and the computation simply explodes. Using the same approach to the Perfectly Matched Layer as was done in the simulator application, this would probably have fixed that issue as well as greatly reduced the complexity of the simulation.

Obviously, the simulator VST looks quite ugly as of now. Ideally one would have a designer create the user interface. VSTs can typically looks quite pleasing to the eye if done correctly. The user interface as it stands is however quite functional and easy to use.

7.4.2 Doing an ER/LR split

As described in section 7.2 above, it is possible to split the reverberation into two separate parts. Using the simulator to generate the early reflection and artificial reverb to generate the late reflection would be the optimal way for generating high quality reverb effects.

7.4.3 Using neural nets to auto-encode impulse responses

As the time stepping schemes is quite demanding, requiring at least 44100 iterations per second, an alternative approach is possible. Similar to the ER/LR split we can try to parameterize the IRs by using convolutional neural nets. As the frequencies in an impulse response gets higher, it gets harder to differentiate between the different sounds. Ie. comparing the higher frequencies between two different impulse responses yield little audible differences. The most important parameter is how fast the higher frequencies decay, as seen in figure 45. We can exploit this, and then considerably lower the time step we are using. However, the Courant number still has to hold as to avoid numerical stability, which may require a different

(possibly implicit) integration scheme. [1] exploit this by extracting peaks from the output response and then generating the high frequency data after-the-fact, thus also saving on the storage requirement. Another possibility is using a form of neural net, or auto-encoder, as was done in [16] for generating the high frequency content. The learning data could be generated by actually recording impulse responses from real locations; ie. playing a loud impulse (for example a gun show, snare drum hit etc.) and capturing the result with two microphones.

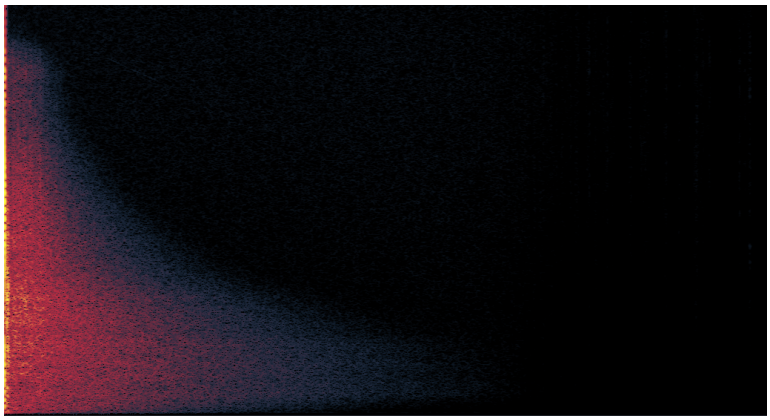

Figure 45: Impulse response with decaying high frequency content.

References

[1] R. N. Nikunj Raghuvanshi and M. C. Lin, "Efficient and accurate sound propagation using adaptive rectangular decomposition." https://www.researchgate.net/publication/220668337_Efficient_and_Accurate_Sound_Propagation_Using_Adaptive_Rectangular_Decomposition. (Accessed on 25/2-19).

[2] A. Allen and N. Raghuvanshi, "Aerophones in flatland," *ACM Transactions on Graphics*, vol. 34, pp. 134:1–134:11, 07 2015.

[3] Steinberg, "Vst." https://www.steinberg.net/en/products/vst.html. (Accessed on 25/2-19).

[4] Juce, "Juce." https://juce.com/. (Accessed on 25/2-19).

[5] S. Bilbao and B. Hamilton, "Higher-order accurate two-step finite difference schemes for the many-dimensional wave equation," *Journal of Computational Physics*, vol. 367, 04 2018.

[6] B. Hamilton and S. Bilbao, "Hexagonal vs. rectilinear grids for explicit finite difference schemes for the two-dimensional wave equation," vol. 133, p. 3532, 05 2013.

[7] L. Feynman and Sands, "The feynman lectures on physics." http://www.feynmanlectures.caltech.edu/I_47.html. (Accessed on 8/5-2019).

[8] E. Kreyszig, H. Kreyszig, and E. J. Norminton, *Advanced Engineering Mathematics*. Hoboken, NJ: Wiley, tenth ed., 2011.

[9] S. G. Johnson, "Notes on perfectly matched layers." http://www-math.mit.edu/~stevenj/18.369/pml.pdf. (Accessed on 13/5-2019).

[10] B. Hamilton and S. Bilbao, "Hexagonal vs. rectilinear grids for explicit finite difference schemes for the two-dimensional wave equation," *21st International Congress on Acoustics, Montral, Canada, 2013*, 2013.

[11] M. R. Schroeder and B. F. Logan, "Colorless artificial reverberation," *IRE Transactions on Audio*, vol. AU-9, 1961.

[12] NVIDIA, "Cuda accelerated libraries." https://developer.nvidia.com/gpu-accelerated-libraries. (Accessed on 21/5-19).

[13] nvpro, "Single-threaded cuda opengl interop." https://github.com/nvpro-samples/gl_cuda_interop_pingpong_st. (Accessed on 20/5-19).

[14] HiFi-LoFi, "Realtime audio convolution." https://github.com/HiFi-LoFi/FFTConvolver. (Accessed on 20/5-19).

[15] NVIDIA, *CUDA driver API*, 10.1 ed.

[16] V. Kuleshov, S. Z. Enam, and S. Ermon, "Audio super resolution using neural networks," *CoRR*, vol. abs/1708.00853, 2017.

NTNU

Kunnskap for en bedre verden