CHALMERS | UNIVERSITY OF GOTHENBURG

Physically-Based Animation of Fire for Android

Bachelor's thesis in Computer Science and Engineering

FREDRIK ALBERS, KARL ANDERSSON, ANTON FORSBERG
LOVISA GRAHN, DANIEL OLSSON, AXEL SVENSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Göteborg, Sweden 2020

BACHELOR'S THESIS IN COMPUTER SCIENCE AND ENGINEERING

Physically-Based Animation of Fire for Android

FREDRIK ALBERS, KARL ANDERSSON, ANTON FORSBERG
LOVISA GRAHN, DANIEL OLSSON, AXEL SVENSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG

Göteborg, Sweden 2020

Physically-Based Animation of Fire for Android
FREDRIK ALBERS, KARL ANDERSSON, ANTON FORSBERG
LOVISA GRAHN, DANIEL OLSSON, AXEL SVENSSON

ISSN 1654-4676
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone: +46 (0)31-772 1000

Cover:
Screenshots of the resulting fire of this project. The fire uses an pyramid as emitter, and has the grid resolution of 102x402x102

Chalmers Reproservice
Göteborg, Sweden 2020

Physically-Based Animation of Fire for Android
FREDRIK ALBERS, KARL ANDERSSON, ANTON FORSBERG
LOVISA GRAHN, DANIEL OLSSON, AXEL SVENSSON
Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg

Abstract

Animation of fire and fluids has been around for at least 20 years in the field of computer graphics. These animations have over the years progressed into real-time simulations running on the GPU and further refined, thanks to modifications such as vorticity and wavelet turbulence, adding higher visual detail. With the improving performance and capability of mobile phones, our issue was how well an animation of fire can be achieved and under which limitations, when developed for a modern mobile device.

For our application, we used Java, C++ and GLSL. The simulation was built upon three layers: simulation overview, simulation operations and slab operation. In order to simulate the behavior of a fluid, we chose to use and compare the Euler equation and the Navier-Stokes equations. To optimize the performance of the fire while still maintaining a high level of detail, we used wavelet turbulence.

The application was able to achieve our initial goal of reaching 30 frames per second with relatively good resulting flame on a modern mobile device. However, older devices might not be able to achieve this performance at all.

Keywords: Fire simulation, Fire simulation Android, Fluid dynamics Android, Wavelet Turbulence, Simulation GPU, Fire simulation 3D

Sammandrag

Animering av eld och fluider har funnits i minst 20 år inom datorgrafik. Dessa har genom åren utvecklats till realtids-simulationer som körs på GPU, samt ytterligare modifierats med vorticitet och Wavelet turbulence som lett till en visuellt högre detaljrikedom. Med den ökande förmågan och prestandan hos moderna mobiltelefoner är vår frågeställning följande: Hur väl kan en animation av eld skapas och vilka begränsningar finns?

För att skriva applikationen användes Java, C++ samt GLSL. Simuleringen består av tre lager: översikt av simuleringen, simulerings-operationer och slab operation. För att simulera en fluids beteende, använde vi oss av Eulers ekvationer, då dessa resulterade i snabbare rendering än det andra valet: Navier-Stokes ekvationer. För att optimera prestandan användes Wavelet turbulence.

Applikationen kunde nå det ursrungliga målet av 30 bildrutor per sekund med en relativt bra resulterande flamma på en moden mobil enhet. Dessvärre, äldre enheter kan eventuellt inte nå denna prestanda.

Preface

This thesis is a depiction of our process of developing an application for Android handheld devices, that simulates a three-dimensional fire that utilizes the GPU of the device. We hope that our thesis will be useful for other students interested in computer graphics and physically-based rendering.

Acknowledgements

We want to thank our supervisor Marco Fratarcangeli for his continuous support and feedback throughout the project. We also want to thank Qualcomm Technologies for providing us with the Snapdragon Profiler software.

Contents

List of Figures

List of Tables

Glossary

advection The act of moving something along a velocity field. 7–10, 12–14, 21, 31

Android An operating system for mobile devices. 2, 4, 6, 21, 30, 40

API Application programming interface, an interface that defines how other applications communicate with a specific software. 2, 4, 21, 22

AVD An emulator that is used to run a simulated Android device. 4

black-body radiation Thermal radiation, the conversion of heat to light, emitted from a black body. 3, 7, 14, 16, 27

buoyancy An upward force exerted by a fluid that opposes the weight of a partially or fully immersed object. 8, 14

C++ A popular object-oriented programming language with low-level capabilities. 2, 4, 5, 21, 30, 31, 38, 40

chromatic adaptation The ability of the human visual system to adjust to changes in illumination. 17, 18

CMake A software tool for managing the build process of software. 4

CPU Central Processing Unit, a computer component that executes instructions. 2, 6, 22, 31, 33, 36, 37

diffusion Resistance to movement in a fluid. 8, 9, 14

dissipation Reduction of an attribute, for example velocity. 3, 10, 14, 15

eddy The swirling of a fluid and the reverse current created when the fluid is in a turbulent flow regime. 12

FPS Frames Per Second, the number of frames that a machine is outputting to a monitor each second. 2, 6, 37, 41

Git A version control tool to track changes and manage differing versions of text-based projects. 5

GitHub A web service that publicly or privately hosts git projects. 5

GLM A mathematics library written for C++ that is intended for graphics software and based on specifications in GLSL. 5

GLSL OpenGL Shading Language, the programming language that is used for shaders in OpenGL. 5, 30, 31

GPU Graphics Processing Unit, a hardware component dedicated for processing graphics. 1, 2, 5, 6, 19, 21, 22, 39–41

Gradle A build tool that incrementally builds the project. 4, 5

graphics pipeline A conceptual model that describes what steps a graphics system needs to perform to render a 3D scene to a 2D screen. 21, 22

homogeneous A property that something is uniform in composition or character. 7, 11

IDE Integrated Development Environment, a software tool dedicated assist and encapsulate development of a program. 4, 38

incompressible The property of constant material density. 7, 21, 32

IntelliJ IDEA An IDE designed for Java development. 4

inviscid A property for a fluid with no viscosity. 7

Java An object-oriented programming language, that for example is used in Android development. 3, 4, 21, 30

kinematic viscosity Ratio of the viscosity to the density of a fluid. 7, 9

MIT-license A permissive free software license. 2, 39

NDK Native Development Kit, which provides tools for developing with C and C++ in android applications. 4, 21, 40

noise function A signal that is seemingly random. 10, 11, 13

OpenGL A graphics programming api. 2, 5, 21, 22, 30, 31, 38

OpenGL ES A subset of OpenGL versions designed for mobile and other embedded devices. 2, 21–23

parallelization The act of designing a computer program or system to process data in parallel. 22

ray casting Method for visualising volumetric data. 15, 20

Scrum Agile framework for software development. 29

SDK Software Development Kit, a set of tools for developing software for a specific platform. 21

shader A program that runs on the GPU. 5, 21, 22, 30, 40

SIMD Single Instruction Multiple Data, when the same operation is performed on multiple sets of data. 22

spectral radiance Radiance of an object for a given wavelength. 16, 17

sprite A two-dimensional image that is drawn directly to a screen. 1

subceed The opposite of exceed. 12

trichromatic Colors are derived from three separate channels. 16

tristimulus values The specific values for the three channels in the given color-space. 18, 19

turbulence noise Summed up perlin noise function with different frequencies by taking the absolute value. The noise take the shape similar to that of turbulence . 10–12

viscosity Resistance to deformation. 7–10

vorticity Measure of rotation in a fluid. 10, 25

vorticity confinement Technique to generate the swirling effects in smoke. 7, 25, 37

wavelet turbulence Optimization technique for fluid simulations that involves scaling up a velocity field using noise. 2, 7, 10, 26, 31, 35–37, 41

1 Introduction

The first chapter introduces the purpose of the project, the scope, and the challenges that has to be dealt with. It also provides the reader with some light background knowledge about the subject prior to the later chapters.

1.1 Background

To simulate a fire in the field of computer graphics is not trivial and comes with its own set of challenges. Especially if one wants the behavior of the fire to be as accurate and realistic as possible [1]. One common way to solve this problem has been to use 2D sprites. These sprites were independent components that were not easily affected by other components on the screen, and could be based on filmed footage or artistic reconstruction [2]. The technique is efficient and has been used a lot in the gaming industry [3], but is lacking in realism due to the repetition of the same footage. If the simulation of fire were to be based on a model with larger physical accuracy, parts of the lost realism could be added back into the simulation, and there would be no need for repeated footage. A physically-based model provides the developers with an opportunity to simulate a fire in a higher resolution, and also add responsiveness to any external forces [3]. One model suitable and commonly used for simulating a gaseous-like phenomena like fire is the Navier-Stokes equations. These equations describe the mathematical model of flow in fluids [4].

1.1.1 Area of Usage

The uses of physically simulated fire are mostly used in the movie industry as *Computer-generated imagery* (CGI) visual effects where practical effects are not viable or wanted at all, as in animated films. If the simulation is done in real-time then the method can be applied in the game industry as well. In that context the physical properties can be used to give an interesting interactive environment.

There is also a potential use in the act of firefighting and fire prevention. Fire-fighters could perhaps use the simulation in training environment to know what to expect when out in the field, possibly using *VR*. There could also be a use (if the simulation is accurate enough) to apply the method when designing buildings. It could be used to predict the fire spread and design the buildings to minimize the spread as well as facilitate when developing efficient evacuation plans. These uses may not be of great financial interest but if they are possible to implement, they are a large improvement in safety in everyday life.

1.1.2 The Expensiveness of 3D Fluid Simulation

When simulating a flow of fluids, a discretization of the fluid domain is usually done, transferring continuous values to discrete ones. There are two ways to discretize the area that contains the fluid, Eulerian discretization and Lagrangian schemes. Eulerian discretization divides the area into fixed cubical cells, also known as voxels in computer graphics. Each cell consists of one velocity vector and scalar quantities (such as pressure or density). In the case of fire simulation, the temperature and the implicit surface of the reaction zone, are also stored in the cell centers [1]. Lagrangian schemes on the other hand, do not have a fixed position in space for the computational elements. This makes them difficult to implement efficiently on the GPU [2]. Since we want to be able to utilize the GPU, we will look further into performance of fluid simulation with Eulerian discretization.

In 3D, velocity vectors $A \times B \times C$ are stored in memory. The scalar pressure field has a larger impact on the perceived realism by the viewer and therefore a higher resolution grid should be used for it [5]. This will lead to the memory consumption of fluid simulation being $O(N)$, where N is the number of cells in the scalar pressure field. The complexity of the fluid simulation algorithm is of order $O(N)$ as well [4].

Due to these reasons, 3D simulations are extremely demanding compared to 2D simulations. The extra dimension that 3D requires, in comparison with 2D, will make the complexity and memory consumption $O(n^3)$ as compared to $O(n^2)$, where n is the number of cells per dimension. Another reason 3D simulations are computationally heavier, is because sampling has to be done from six neighbors for every cell, as compared to four samples in a 2D simulation [6].

The expensiveness of 3D simulation is one reason why no widespread three-dimensional fluid simulation exists on handheld devices, which will be described in the following section.

1.1.3 Mobile Device Versus Desktop

The main difference between simulations on a mobile device and a desktop machine, is the amount of computation power available on the CPU and GPU. Performance on mobile devices are limited due to constraints to both size and power usage, and also because of different architecture design [7]. Hence, a GPU running on a handheld device will not have the same performance conditions as a desktop GPU. Further limitations are then consequently integrated in application programming interfaces (APIs) for the mobile GPUs. An example is OpenGL ES, a specification for communicating with embedded system GPUs, that is a subset of its original version OpenGL but with less functionality. Recent OpenGL ES versions however have come closer to the functionality of OpenGL[8].

1.2 Purpose and Goals

Due to the advantages with physically-based animation of fire, it would be preferable to use in most scenarios where the fire needs to be responsive to external forces. One would wonder how well suited this technique would be for less powerful devices, such as mobile devices. Hence, the purpose of this thesis is to explore the possibilities and limitations of simulating and rendering a physically based fire on handheld devices.

The goal of the software that we are going to develop is to achieve a physically-based simulation of fire done in real-time, that is 30 FPS (*frames per second*) and above (games usually run best at 30 or 60 FPS) [9]. The simulation will make use of a sophisticated rendering system to approach a realistic image from a 3D environment. To increase the performance, the simulation should be done on the GPU to exploit the vast amount of threads it can spawn to efficiently parallelize the calculations. Part of the project goal is to have a demonstration of the final product such that others could take part of our solution and further improve if needed. This will be achieved by having our code to be open source under the *MIT-license*, which makes it possible for other practitioners to use our program freely when referencing the creators of the application. As a special optimization technique for the simulation we will take advantage of the method *wavelet turbulence* for better graphical effects and faster calculations. This enables the possibility to port the software to slower systems, such as handheld devices.

1.3 Problem Statements

- Is it possible to simulate and render a physically-based fire in three dimensions on a handheld device running Android?
- Which frame rates are achievable when rendering the fire?
- Which resolution on the volumetric grid is feasible?
- Is there any significant increase of performance by using wavelet turbulence?
- Is the simulation interactive?

1.4 Scope of Project

Due to the area of fluid simulation being quite vast, we put up some restrictions for the thesis. This thesis will only cover the simulation of a flame being ignited by some shape, and not how a fire spreads or how materials are affected by this. Even though our approach of simulating the phenomena could be used to simulate different kinds of viscous fluids, this thesis will only focus on simulating a flame within a limited volume. The expected results of the simulation should also be a fire with physically plausible appearance; hence this thesis will only cover the performance of three-dimensional simulations of a flame. To do this computational heavy task on a mobile phone, we take advantage of the GPU multi-threading capabilities when rendering to alleviate real time rendering with high computational speed. To utilize the GPU, the computer graphics rendering API OpenGL ES will be used. In addition to the OpenGL ES, we will utilize the programming language C++ to

achieve better computation time as it is closer to hardware than other popular programming languages such as Java [10].

The approach we have chosen to simulate a fire is primarily based on three papers. The first paper is "Stable Fluids" by Jos Stam [4], which is used to simulate and render a viscous fluid in real-time. The second paper we will use in our approach to simulate a fire is the paper "Physically Based Modeling and Animation of Fire" written by Duc Quang Nguyen, Ronald Fedkiw, Henrik Wann Jensen [1]. Finally the third paper we will use an optimization method called wavelet turbulence to achieve a high resolution as desribed in "Wavelet Turbulence for Fluid Simulation" by Theodore Kim, Nils Thürey, Doug James, Markus Gross [11]. This paper demonstrates how to render the colors of the fire with a technique based on the black-body radiation model. By choosing these papers as our primary approach, we will enforce the usage of a limited volume when simulating the fire.

1.5 Related Work

The Navier-Stokes equations were written down in the 19th century. They are universal laws of physics that describe the behavior of fluids. Thus far no general analytical solution exists for the Navier-Stokes equations. However, computers allowed researchers to develop algorithms for numerical solutions to the Navier-Stokes equations. This revolutionized the field of fluid dynamics and branched off in a new discipline called computational fluid dynamics (CFD). For instance, CFD lead to the development of fluid solvers used in aerodynamics and weather simulations. Numerical solutions for transient flow rely on time-discretization of finite time steps. Applications with emphasis on physical accuracy required to be stable and hence, in general, very small time steps had to be used. This resolved in complex and computational heavy algorithms, which in general require a significant amount of time to terminate.

Jos Stam's fluid solver from the paper "Stable Fluids" [4] focused instead on a simple and fast solution suited for the use in real-time computer graphics. Physical accuracy was secondary to the shape and behavior of the fluid. At the time this approach to fluid solvers could not be found in the CFD literature. This paper presented for the first time an unconditionally stable physical based fluid model. The stability allowed the use of arbitrary large time steps without the possibility to "blow-up". Thus, the algorithm could run on slower hardware in real-time.

One shortcoming of the stable fluids algorithm Jos Stam mentions is the tendency of the flow to dampen too rapidly compared to actual experiments. This "numerical dissipation" comes from the linear interpolation only achieving first order accuracy and thus results in a large error. Higher order interpolation could improve the accuracy but would significantly increase the complexity and computational cost of the algorithm.

The paper "FlowFixer: using BFECC for fluid simulation" [12] presented a simple solution to this problem that could be implemented directly on top of the first order semi-Lagrangian integration. Similarly, the paper "An Unconditionally Stable MacCormack Method" [13] presented a method that also achieves second order accuracy but at a reduced computational cost compared to the BFECC method.

The second order accuracy allows fluid features near the Nyquist limit to be robustly resolved. However, the paper "Wavelet Turbulence for Fluid Simulation" [11] focused instead on efficiently resolving frequencies greater than the Nyquist limit. This resulted in a method that allows for the possibility of fluid simulations at spatial resolutions to still achieve high levels of detail at a minimum cost.

In the paper "Physically Based Modeling and Animation of Fire" [1] a proper model for simulating fire is presented. There they present two steps of simulating the fire as the blue core and the hot gaseous products, that differs slightly. To render the fire, they calculate the black-body radiation, to get a more accurate and realistic looking fire. This is the paper that we are basing our thesis on as it is in many ways similar to our scope.

2 Method

This chapter will present the general methods and software tools used for development, as well as specifying the intended outcome for the project.

2.1 Tools

The following section will describe the tools that were used during the project. In this context, tools could be IDE, API, etc.

2.1.1 Android Studio

Android Studio is the official IDE for developing Android applications. The IDE itself is based on another popular IDE for Java development called IntelliJ IDEA, thus having similar looks and functionality. The added functionality in Android Studio includes a Gradle build system for Android environments, an *Android Virtual Device* (AVD) manager and support for *native development kit* (NDK) when developing in native code, to name a few. Android Studio offers common features such as inline debugging to run code line by line, profilers to keep track of CPU and memory usage, and syntax highlighting to assist in writing compilable code and correct syntax errors. Furthermore, Android Studio is available for the most common desktop operating systems [14].

2.1.2 Android Virtual Device

The emulator makes it possible to run a simulated version of an Android device on a desktop machine by using an instance of an AVD. AVDs are handled by the AVD manager, where the specs and other properties can be tailored to aim for a specific set of phones [15]. However simulating android with an application performing heavy calculations, significantly slows the Android simulation down in comparison to a mobile device running Android natively. Therefore it is of great concern to accelerate the android simulation using the CPU with virtualization technology, such as *VT-x* for Intel processors or *AMD-V* for AMD processors [16].

2.1.3 Android NDK

The Android Native Development Kit (NDK) provides set of tools that enables to use C and C++ code for Android development. Which is especially useful when developing an application where performance is critical. The improvement to performance makes it suitable for the project of animating a fire as physically correct as possible. The NDK also enables the inclusion of libraries written in C or C++ [17].

The way that the NDK works is that it has a shared library that is compiled using C or C++ where the native part of the application lies. It is also possible to define static libraries. These libraries are built depending on the *Application Binary Interface* (ABI) which corresponds to different processor architectures. Since android uses Java as standard programming language with the android API, there needs to be a way to communicate between the different languages. Fortunately the *Java Native Interface* (JNI) is an interface that is used to have components from Java and C++ communicate with each other [18].

2.1.4 CMake

CMake is a build system to manage executable files, source code and libraries in a native build environment [19]. The system uses configuration files to specify commands such as linking libraries or to create executables. The CMake build system is used in the project within the NDK. It manages the native C++ files and links the libraries used in the code. CMake and Gradle are used together as the application's build systems, to manage the C++ and Java files respectively in order to build the entire application.

2.1.5 Gradle

Gradle is an open source build tool that incrementally builds the project, based on Groovy or Kotlin DSL [20]. The project will make use of Gradle to handle the part of the application that is written in Java. Gradle uses

a *directed acyclic graph* (DAG) to determine which files need to be rebuilt [21]. Because of this evaluation, unnecessary compilations are avoided. Between builds, Gradle checks if any data has been changed (input, output or implementation). If nothing has changed, the task will not be rebuilt. If another computer has already compiled a task, there is no need for the task to be recompiled locally. Instead, Gradle will load the output from the build cache. If any changes have been made to the file, it will be executed once more. Since Gradle only build files that have been changed, the whole application will not necessarily have to be rebuilt [22].

2.1.6 Git and GitHub

Git, in combination with GitHub, is used as the *version control system* (VCS) for the project. A VCS is used to manage multiple versions of a software, add new features through branches and restore previous changes committed to a file. Git in particular is a distributed VCS, meaning that it is located at multiple places and servers [23]. Git can be used locally but also over the internet, by for example pushing commits to a repository at GitHub. The benefit of using a distributed system such as Git with GitHub is that every group member can clone changes from the repository and add individual changes to a new branch. These changes can then be reintegrated into the repository while other members can work on another feature branch simultaneously. To summarize, Git enhances the project workflow and increases reliability of the software by keeping older commits logged in its history.

2.1.7 OpenGL Mathematics

OpenGL Mathematics, or GLM for short, is a mathematics library for the C++ programming language. It is intended for use with graphics software and OpenGL applications, since it is based on specifications for the OpenGL Shading Language (GLSL). GLM is directly usable in code by importing a single header file, no other compilation is needed. The library comes with built-in functions and classes for random number generation, matrix transformations, vector operations, noise functions etc. [24]. Its similarity with GLSL makes it compatible to use the functionality available by the library and pass it on to different shaders executed by the GPU with ease.

2.1.8 Mobile Devices

Various mobile devices will be used to run the application during the development. This is due to the different devices the authors of this paper have available at hand, mainly their personal devices. This will lead to possibly different results depending on the device, therefor it would be wise to have a predefined device as the standard device when comparing the performance.

The results provided in this thesis are achieved with a OnePlus 7. The chosen device is used since it reaches the current expectations of performance when used for gaming, and it is the best at hand for the authors of this thesis. The important specifications of the device are shown in Table 2.1.[25]

OS:	OxygenOS based on Android™ 10
CPU:	Qualcomm® Snapdragon™ 855 (Octa-core, 7nm, up to 2.84 GHz)
GPU:	Adreno 640
RAM:	8GB LPDDR4X

Table 2.1: OnePlus 7 specs

To measure the performance, a program called Snapdragon Profiler is used. This tool is used to measure and analyse the Qualcomm® Snapdragon™ processors, which in recent versions also includes the Adreno graphics card [26]. It graphs several aspects, such as usage of CPU, GPU, memory and power to help developers get an oversight of the program, and potentially find performance bottlenecks [27]. When obtaining the profiler, one needs to submit a request to achieve a developer account at Qualcomms platforms, which they kindly did for the cause of this thesis.

2.2 Simulation

The approach of simulating a fire on a handheld device consists of taking advantage and be inspired of already existing solutions for fluid simulations in real-time. Firstly, the extensive work on approximating fluid dynamics will be used. Secondly, there are multiple papers which covers the translation of these approximation to code, most notably the paper "Stable Fluids" by Jos Stam [4]. This paper has also been getting some adaptations, one of such are the an GPU version of the stable fluids solver, presented by Nvidia [6]. Another adaptations is the paper "Physically Based Modeling and Animation of Fire" presented in the introductory chapter [1], which uses the stable fluids solver to simulate a fire.

The unique task of this project is the translation of these works to a mobile device, while simulating in three dimensions. The task is considered done when the resulting flame starts to resemble the previous works of physically-based fluid simulation and fire simulation.

2.3 Performance

Certain types of simulations are naturally more demanding and resource-intensive in three dimensions than counterparts in two dimensions, as the amount of data that is operated on goes to a higher magnitude under the same resolution. While performing three-dimensional fluid simulation in real time is not anything new [2], attempting to achieve this for mobile devices can prove to be difficult due to overall reduced available resources and performance. For this reason, it is particularly important to put a great deal of focus on optimization and prioritize optimization above other aspects of the animation. The optimization will however only be considered during animation, and therefore not make any particular effort to optimize the initial loading time of the application. The optimization will focus on performance by trying to maximize the FPS output to achieve a smooth animation of the fire. The resulting FPS will be compared between before and after optimization to confirm that performance has increased. Other aspects such as CPU utilization and memory usage will also be investigated to evaluate if the simulation is too demanding on the device.

2.4 Visuals and Correctness

As the aim of this thesis is to achieve real-time fire animation, the focus will be on cheaper approximations where it is considered reasonable and generally aim for convincing visuals rather than a physically accurate simulation. This means that any method of achieving a more visually realistic flame is preferred over using a method that follows the actual physics of a real fire but has a high impact on performance. As such, there is little reason to make a comparison of the result to a real or otherwise more accurate flame beyond determining if the result is visually similar to an accurate flame. However, the visuals should preferably resemble the results from similar works.

2.5 Intended Outcome

The expected outcome is to produce an application that on a modern Android phone, works correctly in displaying a fire simulation. The code for this application should also be well structured and commented in order to make it easier to use as an example on how fire animation can be implemented under the given circumstances. The visual appearance of the fire will be judged as a success if the authors of this thesis believe that the animation visually behaves like a fire and is visually pleasing. The goal regarding performance is to achieve 30 frames of animation per second (FPS) when running on a modern Android device. The specific number of FPS is chosen as a possible target as this number, as well as 60 FPS, is a common target within computer graphics. The performance should also have a stable level of performance, without any momentary performance drops caused by the animation itself.

To measure the performance of the animation, Android Profiler and Snapdragon Profiler will be used to measure the percentage of CPU utilization, number of megabytes being used and how long it takes to animate one frame on average. This will be used to evaluate the performance of the application. The primary device to run the simulation on is a OnePlus 7.

3 Theory

In this section, the mathematical models used to animate the gaseous-like phenomena will be presented, as well as some manipulations, optimizations and the approach to rendering the fire in three dimensions. This approach is heavily based on selected papers and will be described with sufficient amount of detail. For more details, the reader is recommended to read the original papers.

The fire simulation is divided into three primary steps: velocity, temperature and density. These steps are the fundamentals of the simulation. A substep, wavelet turbulence, is used at the end of the velocity step as an additional optimization measure, but this does otherwise not affect the general simulation steps. In addition, the utilization of other physical models such as black-body radiation and vorticity confinement are used to achieve a more visually interesting result.

3.1 Physical Modelling

When animating the behavior of a fire, four three-dimensional fields are used to store the current state and calculate the upcoming states at any time of the fluid [1], [4]. The first field stores the velocities over the grid as a vector field $\mathbf{u}$, the second field stores the pressure as a scalar field $\mathbf{p}$ [6], the third is a smoke density field $\mathbf{d}$ and the fourth is a temperature field $\mathbf{T}$ [1], [4]. To be able to predict the next state, assumptions about the nature of the fluid must be made. In this thesis, the fluid is considered as an incompressible and homogeneous fluid [6].

An incompressible fluid has a constant volume over time, while a fluid being homogeneous implies that the fluid has a constant density in space [6]. These properties together suggest that the density of the fluid constant over time and space, which provides the possibility to use Navier-Stokes equations or the Euler equation to calculate the states of the fluid.

3.2 Velocity Step - Momentum of the Flame

The velocity step calculates the state of the velocity grid, which will affect how the flame moves. This step consists of three or four components, depending on which approach is used when simulating the flame. The two approaches that are considered is to use either the Navier-Stokes equations or the Euler equation.

3.2.1 Navier-Stokes Equations vs Euler Equation

The Navier-Stokes equations and the Euler equation are used in different scenarios when simulating incompressible fluids. The Euler equation is used for inviscid fluids, whilst the Navier-Stokes equations are used when incorporating viscosity. The two equations (Equation 3.1 and 3.2) are rather similar, where t resembles the time in space, ρ is the overall density, υ is the kinematic viscosity, $\mathbf{f}$ is the external forces and ∇ denotes the vector of spatial derivatives, $(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z})$. [4], [6]

$$\text{Euler equation: } \frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \frac{1}{\rho}\nabla \mathbf{p} + \mathbf{f} \tag{3.1}$$

$$\text{Navier-Stokes equations: } \frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \frac{1}{\rho}\nabla \mathbf{p} + \upsilon \nabla^2 \mathbf{u} + \mathbf{f} \tag{3.2}$$

Equation 3.1 and Equation 3.2 are both subject to the *incompressibility-constraint* $\nabla \cdot \mathbf{u} = 0$, which ensures that the velocity field is divergence free [6]. Both equations also share the *Advection* component, *Pressure* component and the *External Forces* component. These three terms are also considered as three components of the velocity step. To briefly explain what each term implies, the *Advection* term is generally applicable for transporting objects, temperatures, densities, or in this case the velocity itself, along the flow of the velocity field. The *pressure* term simulates the force that comes from a difference in pressure. Lastly, the *force* term

adds external forces to the velocity field. This includes any other types of forces that are applied on the fluid, such as the buoyancy force or the forces from wind.

The difference between Equation 3.1 and Equation 3.2 is the $v\nabla^2\mathbf{u}$ term, which describes the *diffusion* of the fluid. Diffusion can be thought of as the thickness of the fluid, and the term adds an inertia to the movement, or the velocities over the grid, which will slow down the movement of the fire [6]. The important factor in the diffusion, as mentioned earlier, is the viscosity constant, which controls the "thickness" of the fire. In the Euler equation, the viscosity is assumed as zero, which nullifies the term. This assumption will decrease the physically correctness of the simulation, which implies that the Euler equation is a more approximate approach than the Navier-Stokes equations [28].

3.2.2 Solving the Behaviour Equations

A solution using numerical integration provides a straightforward approach that solves the equations incrementally [6]. However, the equations need to be transformed to be more suitable for a numerical solution.

Firstly, the $\nabla \cdot \mathbf{u} = 0$ condition must be ensured for the velocity field after applying each component in the velocity step [4]. This can be achieved by projecting the resulting velocity field onto its divergence free part. This introduces the operator $\mathbf{P}$ which performs this projection.

The projection utilizes a mathematical result known as *The Helmholtz-Hodge Decomposition*. The theorem shows that a vector field $\mathbf{v}$ with non-zero divergence can be decomposed in the form of Equation 3.3, where $\mathbf{u}$ has a zero divergence and p is a scalar field. In this context, p represents the pressure field [4].

$$\mathbf{v} = \mathbf{u} + \nabla p \tag{3.3}$$

The projection simply subtracts the gradient of the scalar field of $\mathbf{v}$, resulting in the divergence free vector field $\mathbf{u}$ as shown in Equation 3.4 [4], [6].

$$\mathbf{P}(\mathbf{v}) = \mathbf{u} = \mathbf{v} - \nabla p \tag{3.4}$$

The missing piece of Equation 3.4 is to calculate ∇p, which is done by utilizing the Helmholtz-Hodge decomposition again. When multiplying both sides of Equation 3.3 with ∇, Equation 3.5 is achieved since $\nabla \cdot \mathbf{u} = 0$. This equation is also known as a *Poisson equation* for the pressure field p [4], [6].

$$\nabla \cdot \mathbf{v} = \nabla^2 p \tag{3.5}$$

When the projection is applied to Equation 3.3, the observation that $\mathbf{P}(\nabla p) = 0$ can be made, since $\mathbf{P}(\mathbf{u}) = \mathbf{u}$ as $\mathbf{u}$ is divergence free, and $\mathbf{P}(\mathbf{v}) = \mathbf{u}$ according to Equation 3.4 [4], [6]. Using this observation, the projection can be applied to the Navier-Stokes equation, resulting in Equation 3.6. The $\mathbf{P}(\nabla p)$ term equals zero, while $\mathbf{P}(\frac{\partial \mathbf{u}}{\partial t}) = \frac{\partial \mathbf{u}}{\partial t}$ as a consequence of $\mathbf{u}$ being divergence free.

$$\frac{\partial \mathbf{u}}{\partial t} = \mathbf{P}(-(\mathbf{u} \cdot \nabla)\mathbf{u} + v\nabla^2\mathbf{u} + \mathbf{f}) \tag{3.6}$$

Equation 3.6 is the algorithm that Jos Stam uses in his paper *Stable Fluids* to simulate a fluid [4], [6].

3.2.3 Advection

Advection is the flow of a quantity along the velocity field of the fluid. The velocity field itself is also advected, and this self-advection is represented in the Navier-Stokes equations with the advection term. Computing the advection is done by moving along the velocity field for each element in the grid. One of the easiest methods for doing this is using Euler's method, which works by moving the position, $\mathbf{r}$, of a particle the distance it would travel in the velocity field during a short time interval δt, see Equation 3.7 [6].

$$\mathbf{r}(t + \delta t) = \mathbf{r}(t) + \mathbf{u}(t)\delta t \tag{3.7}$$

However, Euler's method suffers from a drawback that the method cannot be used for large time steps. This is due to the fact that the method can become unstable and produce results inconvenient for the simulation, if the time step is larger than the size of a grid cell. In 1999, Jos Stam came up with another solution using an implicit method [4]. To solve the equation, the problem can instead be inverted by tracing the particle back in time to its previous position and use the values at that position for the grid cell at the start. The formula for updating a quantity, q, using this method is done using Equation 3.8. The $\mathbf{x}$ in the equation is the position of the particle [6].

$$q(\mathbf{x}, t + \delta t) = q(\mathbf{x} - \mathbf{u}(\mathbf{x}, t)\delta t, t) \tag{3.8}$$

3.2.4 Diffusion

The diffusion of a fluid represents the resistance of movement, how the velocity of the fluid is affected and dissipated by its viscosity. Viscous diffusion can be represented by a partial differential equation [6]:

$$\frac{\partial \mathbf{u}}{\partial t} = v\nabla^2\mathbf{u} \tag{3.9}$$

The partial differential equation is solved using an approach similar to the one used in advection. Likewise, using the explicit Euler method will result in an unstable simulation, hence why Stam's implicit solution is used for this part as well [6]:

$$(\mathbf{I} - v\delta t\nabla^2)\mathbf{u}(x, t + \delta t) = \mathbf{u}(x, t) \tag{3.10}$$

The $\mathbf{I}$ in Equation 3.10 is the identity matrix. This equation is a Poisson equation for the velocity of the fluid that is solvable using an iterative method, applying Equation 3.11 for each cell in the grid [6]. The kinematic viscosity used in the simulation is $18 * 10^{-6} m^2/s$, which is the kinematic viscosity of air at 60°C [29].

3.2.5 Solving the Poisson Equations

A Poisson equation is commonly recognized as a matrix equation, $A\mathbf{x} = \mathbf{b}$, where A is a matrix, $\mathbf{x}$ is the resulting vector while $\mathbf{b}$ is a vector of constants [6]. In the context of where the Poisson equations are used in this thesis, the matrices are represented as the Laplacian operators ∇^2.

An trivial technique to use when solving a Poisson equation is called *Jacobi iteration*, which is an iterative solver technique [6]. The solver starts with some initial values for the resulting vector $\mathbf{x}$ and produces an improved result for every iteration. The Jacobi equation is given in Equation 3.11, where k indicates the iteration number while α and β are constants.

$$X^{(k+1)}_{x,y,z} = \frac{X^{(k)}_{x-1,y,z} + X^{(k)}_{x+1,y,z} + X^{(k)}_{x,y-1,z} + X^{(k)}_{x,y+1,z} + X^{(k)}_{x,y,z-1} + X^{(k)}_{x,y,z+1} + \alpha b_{x,y,z}}{\beta} \tag{3.11}$$

The values of X, b, α and β differ depending on which context they are used in. When solving the pressure equation (Equation 3.5), X represents the pressure p, b represents $\nabla \cdot v$ while $\alpha = -(\partial x)^2$ and $\beta = 6$. In the diffusion context (Equation 3.10), both X and b represent $\mathbf{u}$, while $\alpha = \dfrac{-(\partial x)^2}{v\partial t}$ and $\beta = 6\alpha$. [6]

The specific value 6 is used since it represents the number of neighboring grids considered. In this case, a three-dimensional grid is used [2].

The consequences of using this non-exact solution when solving for the pressure is that the incompressibility-constraint is not satisfied and thus the volume will not be preserved [2]. The visual artifacts of this are not as noticeable in fire simulation as they would be in a water simulation [2]. But it is a compromise that must be made due to the performance constraints of real time simulations.

3.2.6 Vorticity Confinement

Low viscosity fluids, such as the simulated fire consisting of heated air, often contains rotational flows called *vorticity*, also known as curl [6]. The curl of a point in a vector field is a mathematical way of expressing the rotation at that point as a vector [30]. When simulating dissipation numerically, there is a tendency for the curl to get minimized and have less impact on the simulation. Vorticity confinement is a way to add the rotational flows back into the fluid to preserve this otherwise lost behavior.

First, the vorticity ω is computed for the velocity field using the curl operator.

$$\omega = \nabla \times \mathbf{u} \tag{3.12}$$

The vorticity is then computed into a normalized vorticity field $\mathbf{\Psi}$.

$$\mathbf{\Psi} = \frac{\eta}{|\eta|} \qquad \eta = \nabla|\omega| \tag{3.13}$$

The last step is to calculate the vorticity force to be added to the velocity field in order to approximate the vorticity lost from dissipation.

$$f_{vorticity} = \varepsilon(\Psi \times \omega)\delta x \tag{3.14}$$

Here ε is a customizable scale parameter to adjust the amount of force added and δx refers to the distance between two grid elements in the vector field [6].

3.3 Wavelet Turbulence

To achieve a higher level of detail for temperature and smoke without also increasing the resolution of the simulated velocity, a method called wavelet turbulence is used. This method takes a lower resolution velocity grid and scales it up while keeping a high level of detail. To achieve this high level of detail, a noise function is used to fill in the gaps in detail as the velocity field is scaled up.

Subsequent simulation steps will use the scaled-up velocity field during the advection step. With this, the velocity step will be calculated on the lower resolution while the temperature and smoke fields will be calculated on the higher resolution. If successful, this will overall decrease calculations as it reduces the size of calculations during the velocity step [11].

3.3.1 Turbulence Noise

The main functionality behind wavelet turbulence is the use of a turbulence noise function. Turbulence noise is similar to other noise functions, with the difference of it taking the absolute value for each octave. The noise is used to fill in the missing high frequencies of the velocity grid once it has been scaled [11].

The noise functions in question is based on Perlin's turbulence function. The turbulence function makes use of the noise function already supplied by Perlin and applies the property of turbulence as previously described. This is shown in Equation 3.15. Here i denotes the band, $\mathbf{x}$ denotes the position and p denotes the noise function [31].

$$\sum_{i=i_{min}}^{i_{max}} p(2^i\mathbf{x})\frac{1}{2}^{i-i_{min}} \tag{3.15}$$

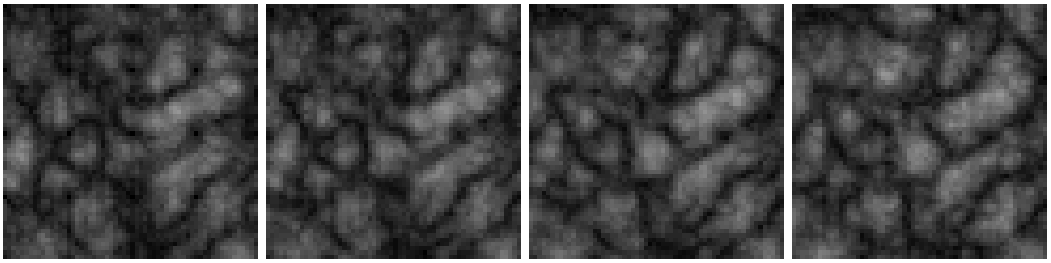

Figure 3.1: *The first four slices of 3D turbulence noise with resolution* 60^3

Visual result from the turbulence function in Equation 3.15 is shown in Figure 3.1. The turbulence function returns a scalar whereas a vector value is required. To solve this, the turbulence function is applied three times to be able to generate a three-dimensional vector. This derivation is shown in Equation 3.16, where p_1, p_2, p_3 denotes the different turbulence fields [11].

$$\mathbf{p}(\mathbf{x}) = \left(\frac{\partial p_1}{\partial y} - \frac{\partial p_2}{\partial z}, \frac{\partial p_3}{\partial z} - \frac{\partial p_1}{\partial x}, \frac{\partial p_2}{\partial x} - \frac{\partial p_3}{\partial y} \right) \tag{3.16}$$

By substituting $p(\mathbf{x})$ with $\mathbf{p}(\mathbf{x})$ in Equation 3.15 the final noise function $\mathbf{y}(\mathbf{x})$ can be derived, as can be seen in Equation 3.17.

$$\mathbf{y}(\mathbf{x}) = \sum_{i=i_{min}}^{i_{max}} \mathbf{p}(2^i \mathbf{x}) \frac{1}{2}^{i - i_{min}} \tag{3.17}$$

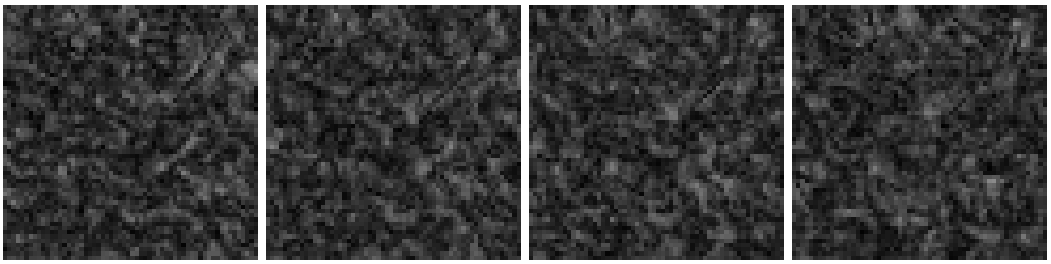

Figure 3.2: *The first four slices of the final vector noise 3D with resolution* 60^3*. The grayscale represents the length of the vector.*

The noise can be visualized by taking the length of the vectors as shown in Figure 3.2. This method generates noise everywhere in the velocity field. For an accurate simulation, the noise has to be weighted depending on the lower resolution velocity. Otherwise there will be flows in the simulation where there should be none [11].

3.3.2 Kolmogorov Theory

The turbulence noise can be weighted by the velocities' energy density. The energy density can be calculated by taking the kinetic energy of the velocity, which is described in Equation 3.18.

$$e(\mathbf{x}) = \frac{|\mathbf{u}(\mathbf{x})|^2}{2} \tag{3.18}$$

This follows the general kinetic energy equation, $E_k = \frac{mv^2}{2}$ with the exception of the mass being substituted by the constant 1. This substitution is possible due to the gas being incompressible, hence the mass can be assumed to be homogeneous throughout the simulation.

To inject the turbulence noise within the higher frequencies, the noise needs to be weighed with the energy density in said frequency. This can be calculated by using the Kolmogorov theory, which states that the energy

spectrum of a turbulent fluid approaches a five-thirds power distribution [11]. This is shown by the recurrence relation in Equation 3.19.

$$e_t(2k) = e_t(k)^{-\frac{5}{3}}, \quad e_t(1) = C\varepsilon^{\frac{2}{3}} \tag{3.19}$$

However, while this equation calculates the total energy density, a better approach would be to calculate the energy density on a specific frequency cell. By having a function $\hat{e}(\mathbf{x}, k)$ evaluate the energy density at a certain frequency band and on a specific position, Equation 3.20 can be derived [11].

$$\hat{e}(\mathbf{x}, 2k) = \hat{e}(\mathbf{x}, k)^{-\frac{5}{3}}, \quad \hat{e}(\mathbf{x}, 1) = C\varepsilon^{\frac{2}{3}} \tag{3.20}$$

By using the energy to scale the turbulence, the noise can be expected to only be applied in the areas where there is a higher velocity in a certain frequency band. This narrows down the noise from being globally applied to only applied within energy dense grid cells [11].

3.3.3 Scattering

Since the turbulence noise is not recomputed, the noise will appear static when applied to the simulation. Therefore, it would be of interest to use advection and advect the noise with the velocity flow. This could easily be achieved by advecting the texture coordinates $\mathbf{c} = (c_u, c_v, c_w)$ and then using these new coordinates when accessing the noise [11].

The deviation of the new texture coordinate ultimately mirrors the physical phenomena called *scattering*. Scattering occurs when a local eddy either breaks down to two smaller eddies known as *front scattering*, or two local eddies combines to form a greater eddy called *back scattering* [11].

A problem with this arises when the new texture coordinates deviate too far from the original set of coordinates. When deviation is too great, it has either *front scattered* or *back scattered* at which point the texture coordinate should be regenerated. To detect when this is the case, a method to obtain the deviation has to be used [11].

To calculate the deviation, the deformation can be obtained by using the Jacobian of the texture coordinates. The Jacobian represents the transformation of the texture coordinates as the partial derivation of the spatial position. Once the Jacobian is obtained, the deviation can be quantified by its eigenvalues. If the eigenvalue exceeds 2, the local eddy has front scattered and the texture coordinate should be regenerated. In the other case if the eigenvalue *subceed* $\frac{1}{2}$ then the local eddy back scattered and the texture coordinate should also regenerate. The texture coordinate should otherwise stay the same [11].

3.3.4 Calculating the Eigenvalues

To calculate the eigenvalues of the Jacobian, an iterative method called the QR algorithm is used [32, p. 396]. The algorithm takes the advantage that a real matrix A with linearly independent columns can be decomposed as an orthogonalized matrix Q and an upper triangular matrix R, which is known as the QR-decomposition [32, p. 395]. By using the QR-decomposition, matrix A can be written as $A_0 = Q_0 R_0$. The premise of the QR algorithm is to iteratively multiply Q and R in the opposite order, resulting in a more orthogonalized version of A after every iteration. This can be seen in Equation 3.21. After some iterations A converges, and the eigenvalues of A are obtained by its diagonal.

$$A_{k+1} = R_k Q_k \tag{3.21}$$

The first step of using the QR algorithm on the Jacobian, is to assure that each column of the Jacobian is linearly independent. This can be done by examining if the determinant of Jacobian is nonzero, which also assures that the Jacobian is invertible [32, p. 94]. If the Jacobian is not invertible, it will be unusable in the upcoming steps of Wavelet Turbulence. Hence, the Jacobian is provided with default eigenvalues outside the span of $\frac{1}{2}$ and 2, which regenerates the texture coordinates. Otherwise, if the Jacobian is invertible, the QR algorithm may be used.

The decomposition of Jacobian is made by using the Gram–Schmidt orthogonalization, where Q and R can be obtained as in Equation 3.22 and 3.23 respectively [32, p. 107]. Note, the Jacobian is rewritten as $[j_1, j_2, j_3]$ to fit the approach.

$$\begin{aligned} &Q = [e_1, e_2, \ldots, e_n] \\ &e_1 = a_1/|j_1| \\ &e_2 = f_2/|f_2|, \text{where } f_2 = j_2 - (j_2 \cdot e_1)e_1 \\ &e_n = f_3/|f_3|, \text{where } f_3 = a_3 - (j_3 \cdot e_1)e_1 - (j_3 \cdot e_3)e_3 \end{aligned} \tag{3.22}$$

$$R = Q^T QR = Q^T A \tag{3.23}$$

The eigenvalues of Jacobian are obtained from the diagonal after iterating the QR-algorithm 20 times, which is an arbitrary amount of iterations.

3.3.5 Texture Distortion

The advection of the texture coordinated distorts the noise by stretching and rotating. This has the possibility of compressing the noise, which violates the incompressibility of the gas. To avoid this, the derivatives for the Cartesian axes must be calculated. This is done by projecting the Cartesian axes to texture space and taking the derivatives of the texture coordinates of the noise, as can be seen in Equation 3.24. Where $\mathbf{J}(\mathbf{c}(\mathbf{x}))^{-1}$ denotes the inverse of the Jacobian of the coordinate textures [11].

$$\left[\frac{\partial p}{\partial c_u}\frac{\partial p}{\partial c_v}\frac{\partial p}{\partial c_w}\right]\mathbf{J}(\mathbf{c}(\mathbf{x}))^{-1}\left[\,\mathbf{v_x}\,\mathbf{v_y}\,\mathbf{v_z}\,\right] = \left[\frac{\partial p}{\partial x}\frac{\partial p}{\partial y}\frac{\partial p}{\partial z}\right] \tag{3.24}$$

Applying this method with the noise function the function $\mathbf{z}(\mathbf{c})$ is derived that should be used to avoid texture distortion [11].

3.3.6 Fluid Synthesis

Applying all the previous sections will forge a method for synthesizing the fluid from a lower resolution to a higher resolution. This is the final step, which creates the high-resolution velocity as shown in Equation 3.25. The method $I(\mathbf{x}, \mathbf{X})$ interpolates a value from the lower resolution field $\mathbf{x}$ to fit within the higher resolution. $\mathbf{U}(\mathbf{X})$ is the resulting velocity in the higher resolution [11].

$$\mathbf{U}(\mathbf{X}) = I(\mathbf{u}, \mathbf{X}) + 2^{-\frac{5}{6}} I\left(\hat{e}\left(\mathbf{x}, \frac{n}{2}\right), \mathbf{X}\right)\mathbf{z}\left(I\left(\mathbf{c}, \mathbf{X}\right)\right) \tag{3.25}$$

The noise will be applied to frequencies between $i_{min} = log_2 n$ and $i_{max} = log_2 \frac{N}{2}$, where n is the length of the smaller resolution and N is the length of the higher resolution. In the case of uneven dimensions, n can be substituted with its minimum length and N substituted with its maximum length. The reason for this interval is to fill in the detail between the lower and higher resolution frequencies [11].

3.4 Adding External Forces

The "force term" $\mathbf{f}$ is solved numerically same as the other terms (Equation 3.26).

$$q(x, t + \delta t) = q(x, t) + \mathbf{f}\Delta t \tag{3.26}$$

In Equation 3.26 q is the quantity being affected by the force.

3.4.1 Buoyancy Force

The temperature of a fluid is one of the depending factors affecting the density of it [33]. The basic principle of density variations is that lower density substances "float" above higher density ones. This includes hot air rising above cold air when heated by a fire. To model this type of behavior a force can be applied to the velocity field. This is described as a *buoyant force* [2]. To implement buoyancy and keep track of the temperature state of a fluid, a new scalar field T needs to be added.

$$\mathbf{f}_{buoyancy} = \sigma(T - T_0)\hat{\mathrm{j}} \tag{3.27}$$

The parameter σ is a scaling factor for the buoyancy force, T is the temperature at a position and T_0 is the ambient temperature. The force is therefore applied to particles of the fluid where the temperature is higher than the ambient temperature [6]. The vector $\hat{\mathrm{j}}$ is determining the direction of the force applied. Since the buoyancy force is meant to resist gravity, the vector has to point in the opposite direction, upwards, perpendicular to the ground.

3.5 Simulation Step for Substances

Moving a general non-reactive substance, represented by a scalar value a, through a fluid adhere to a similar process as the fluid who follows the Navier-Stokes equation [4]. The equation for the evolution of these scalar values is depicted in Equation 3.28[4].

$$\frac{\partial a}{\partial t} = -(\mathbf{u} \cdot \nabla)a + \kappa_a \nabla^2 a - \alpha_a a + S_a \tag{3.28}$$

Equation 3.28 shares the components of a advection term ($-u \cdot \nabla a$), diffusion term ($\kappa_a \nabla^2$) and a force term (S_a) with the Navier-Stokes equation [4]. But in addition, this equation contain a dissipation term $-\alpha_a a$ where α_a is the dissipation rate and is solved numerically, as depicted in Equation 3.29[4].

$$a(x, t + \Delta t) = \frac{a(x, t)}{(1 + \Delta t \alpha_a)} \tag{3.29}$$

If the fluid simulation is based on the Euler equations and not the Navier-Stokes equations, the diffusion term ($k_a \nabla^2$) would be discarded.

3.5.1 Density Step

The density of the smoke has no dissipation [1], this leads to Equation 3.30, describing a density moving through a velocity field [34].

$$\frac{\partial \rho}{\partial t} = -(u \cdot \nabla)\rho + \kappa \nabla^2 \rho + S \tag{3.30}$$

To solve Equation 3.30, look at the following sections for more details: Section 3.2.3 for the advection term, Section 3.2.4 for the diffusion and 3.4 for the "force term".

3.5.2 Temperature Step

Temperature dissipates heat through the process of thermal radiation (black-body radiation in this case) giving rise to the fires distinct color (see Section 3.7 for more detail). The dissipation term differs from the one depicted in Equation 3.28, this one is derived from the conservation of energy and is as follows $-c_T(\frac{T - T_{air}}{T_{max} - T_{air}})^4$. c_T is the cooling constant, T is the temperature of the fire, T_{air} is the temperature of the environment and T_{max}

is the maximum temperature the fire can reach. The complete equation for the temperature step is portrayed in Equation 3.31.

$$\frac{\partial T}{\partial t} = -(u \cdot \nabla)T + \kappa \nabla^2 T - c_T(\frac{T - T_{air}}{T_{max} - T_{air}})^4 + S \tag{3.31}$$

Solving for the dissipation term is done with analytical integration (Equation 3.32) [1].

$$T(x, t + \delta t) = -c_T(\frac{T(x,t) - T_{air}}{T_{max} - T_{air}})\delta t \tag{3.32}$$

See the relevant Sections 3.2.4, 3.2.3 and 3.4 for solving the remaining terms.

3.6 Boundary Condition

Differential equations defined on a finite domain requires boundary conditions to be well posed [6]. A well-posed differential equation is one that has a solution, the solution is uniquely defined and the equation's solution is not discontinuous [35]. For the simulation's initial conditions, it is assumed that the velocity and pressure are zero. As the fluid evolves with each time step the pure Neumann boundary condition is applied to the pressure and the no-slip condition to the velocity.

The Poisson-pressure equation described in Section 3.2.5 requires pure Neumann boundary conditions for a correct solution [6]. The definition of the condition is shown in Equation 3.33 and is interpreted as the rate of change of pressure in the direction normal to the boundary is zero [6].

$$\frac{\partial p}{\partial n} = 0 \tag{3.33}$$

The no-slip condition for the velocity assumes that nothing can flow outside of the simulation grid, essentially treating it as being inside a box. The no-slip condition is thus defined as that the velocity goes to zero at the boundaries [6].

Because of the performance constraints in real time simulation, both conditions are approximated to be the average of two cells adjacent to any edge satisfying the boundary condition [6].

Equation 3.34 shows these conditions applied to one side of a two-dimensional case.

$$\begin{aligned} \text{no-slip: } \frac{u_{(0,j)} + u_{(1,j)}}{2} &= 0 \text{ for } j \in [0, N], \\ \text{Pure Neumann: } \frac{p_{(0,j)} - p_{(1,j)}}{\Delta x} &= 0 \text{ for } j \in [0, N]. \end{aligned} \tag{3.34}$$

This approximation of the pure Neumann boundary condition is easily solved by setting the boundary value to the one just inside the boundary. The same is done for the no-slip condition except the negative value of the cell just inside the boundary is used.

3.7 Calculating the Color of the Fire

In this section, a physically based method for calculating the color of the fire simulation is being described. This process is the one outlined by Nguyen et al. in the paper "Physically Based Modeling and Animation of Fire" [1]. The methods described in Section 3.7.1 and 3.7.2 are performed during volume ray casting (see Section 3.9), while the methods in Section 3.7.3 and 3.7.4 are performed as post-processing effects.

3.7.1 Black-body Radiation

A participating medium can be described by its properties to scatter, absorb and emit light. Fire is one example of a participating medium that is also a black-body radiator, meaning that no scattering of light occurs [1]. While scattering do occur in the smoke produced by the fire, it is not included in the model described below. For black-body radiation, light transport in a participating media is described by Equation 3.35 [1].

$$\begin{aligned}(\vec{\omega} \cdot \nabla)L_\lambda(x, \vec{\omega}) &= -\sigma_t(x)L_\lambda(x, \vec{\omega}) + \sigma_a(x)L_{e,\lambda}(x), \\ \sigma_t &= \sigma_a.\end{aligned} \tag{3.35}$$

L_λ is the spectral radiance, σ_t is the extinction coefficient and σ_a the absorption coefficient. $L_{e,\lambda}$ the emitted spectral radiance in Equation 3.35 is Planck's formula and λ is the wavelength of the light.

$$\begin{aligned}L_{e,\lambda}(x) &= \frac{2C_1}{\lambda^5(e^{C_2/\lambda T)} - 1} \\ C_1 &\approx 3.7418 \times 10^{-16} Wm^2 \\ C_2 &\approx 1.4388 \times 10^{-2} mK\end{aligned} \tag{3.36}$$

Planck's formula where T is the temperature at the position x [1].

The light transport function (Equation 3.35) is continuous and thus for the use case of real time simulation estimated through discretization. A ray of light is seen as multiple short segments, where the terminating ray segment is calculated in accordance to Equation 3.37 [1].

$$L_\lambda(x) = e^{-\sigma_t \Delta x} L_{(n-1),\lambda}(x + \Delta x) + \sigma_a L_{e,\lambda}(x)\Delta x \tag{3.37}$$

$L_{(n-1),\lambda}$ is the spectral radiance of the previous line segment and Δx is the length of the line segment.

This calculation needs to be done for multiple samples across the visible wavelength spectrum to approximate the full spectral distribution of the emitted radiance coming from the fire.

3.7.2 Converting Radiance to CIE XYZ Color-space

The human vision is trichromatic because the eyes have three different types of cone cells, responsible for color vision. The cones respond to radiant energy by sending out a certain nerve signal depending on the received wavelength of the light. The three cones have different response curves allowing humans to detect different colors. The Commission Internationale de l'Éclairage (CIE). CIE have also standardized the conversion of spectral radiance to the CIE XYZ color space seen in Equation 3.38 [36].

$$\begin{aligned}X &= \int L(\lambda)\bar{x}(\lambda)d\lambda, \\ Y &= \int L(\lambda)\bar{y}(\lambda)d\lambda, \\ Z &= \int L(\lambda)\bar{z}(\lambda)d\lambda.\end{aligned} \tag{3.38}$$

In Equation 3.38 λ is the wavelength, $L(\lambda)$ is the spectral radiance and $\bar{x}(\lambda)$, $\bar{y}(\lambda)$,$\bar{z}(\lambda)$ are the spectra curves known as standard observer or color matching functions.

The color matching functions are standardized as tabular values, but analytic approximations may also be used. The analytic approximations in Equation 3.39 exhibit less error than the experimental measurements

used to generate the standardized curves [36], thus the physically accuracy is not compromised by relying on these approximations.

$$\begin{aligned}
\bar{x}_{31}(\lambda) &= 1.065\exp(-\frac{1}{2}(\frac{\lambda-595.8}{33.33})^2) + 0.366\exp(-\frac{1}{2}(\frac{\lambda-446.8}{19.44})^2),\\
\bar{y}_{31}(\lambda) &= 1.014\exp(-\frac{1}{2}(\frac{ln(\lambda)-ln(556.3)}{0.075})^2),\\
\bar{z}_{31}(\lambda) &= 1.839\exp(-\frac{1}{2}(\frac{ln(\lambda)-ln(449.8)}{0.051})^2).
\end{aligned} \tag{3.39}$$

The conversion of spectral radiance to the CIE XYZ color space through Equation 3.38 involves integrals and solving this analytically is too demanding in the use case of real time simulation. For this reason, the integral approximations in Equation 3.40 is used instead.

$$\begin{aligned}
X &= \sum L(\lambda)\bar{x}(\lambda)\Delta\lambda,\\
Y &= \sum L(\lambda)\bar{y}(\lambda)\Delta\lambda,\\
Z &= \sum L(\lambda)\bar{z}(\lambda)\Delta\lambda.
\end{aligned} \tag{3.40}$$

For $L(\lambda)$ in Equation 3.40, the spectral radiance approximation from Equation 3.37 is used.

3.7.3 Chromatic Adaptation

Human color vision comes from three types of cone cells. The cones respond to different wavelength to a varying degree. One of the cones is responsive to most of the long-wavelengths, another for the middle-wavelengths and the third for the short-wavelengths. The cones are therefore understandable referred to as LMS [37]. The LMS color space is a mapping of the responses of the three cones to different wavelengths, where the maximum response of a cone often is normalized to the value 1 while the value 0 corresponds to no response (See Figure 3.3).

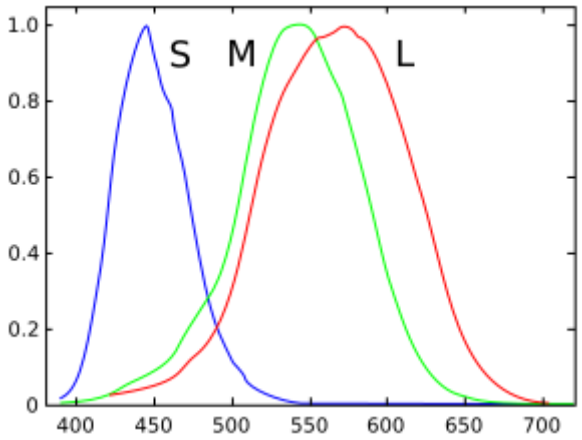

Figure 3.3: *Normalized responsivity spectra for L, M and S cone cell. From [38]. CC BY-SA 3.0*

Chromatic adaptation is the cones ability to adapt to different colored illumination in order to approximately preserve the appearance of object colors [37]. Fairchild also describes von Kries' idea that lay the foundation of modern chromatic adaptation models, that each of the cones adapt independently from one and another.

This idea is also described in Equation 3.41.

$$\begin{aligned} L_a &= k_L L, \\ M_a &= k_M M, \\ S_a &= k_S S. \end{aligned} \tag{3.41}$$

In Equation 3.41 L, M and S are the initial cone responses. k_L, k_M and k_S are the separate scalar values for each cone. L_a,M_a and S_a are the post-adaptation cone signals [37]. For a fire simulation the eyes must adapt to the color of the spectrum with the maximum temperature [1]. This corresponds to the following cone scalar values depicted in Equation 3.42.

$$\begin{aligned} k_L &= 1/L_{max}, \\ k_M &= 1/M_{max}, \\ k_S &= 1/S_{max}. \end{aligned} \tag{3.42}$$

The chromatic adaptation transform (CAT) uses the LMS color space but the previous step described in Section 3.7.2 uses the CIE XYZ color space. Therefore, a transform from XYZ to the LMS color space is done using the MCAT02 matrix transformation (Equation 3.43).

$$M_{CAT02} = \begin{bmatrix} 0.4002 & 0.7076 & -0.0808 \\ -0.2263 & 1.1653 & 0.0457 \\ 0.0 & 0.0 & 0.9182 \end{bmatrix} \tag{3.43}$$

This Matrix (Equation 3.43) is multiplied with XYZ tristimulus values to get the LMS cone responses. The inverse of this matrix thus describes the conversation of LMS cone responses to XYZ tristimulus values. The complete chromatic adaptation transform is described in Equation 3.44.

$$\begin{bmatrix} X_a \\ Y_a \\ Z_a \end{bmatrix} = M^{-1} \begin{bmatrix} 1/L_{max} & 0 & 0 \\ 0 & 1/M_{max} & 0 \\ 0 & 0 & 1/S_{max} \end{bmatrix} M \begin{bmatrix} X_r \\ Y_r \\ Z_r \end{bmatrix} \tag{3.44}$$

X_r, Y_r and Z_r are the raw XYZ tristimulus values, X_a, Y_a and Z_a are the post-adaptation tristimulus values, M is the matrix depicted in Equation 3.43. Note that the LMS values are transformed back to the XYZ color space due to its versatility to work with and the 3x3 matrix is equivalent to Equation 3.41 with the cone scalar values depicted in Equation 3.42.

3.7.4 sRGB Conversion and Gamma Correction

The XYZ tristimulus values need to be converted to a format the screen can understand and display with the expected colors. The sRGB color space is one example and the conversion is done through a matrix multiplication (Equation 3.45) [39].

$$\begin{bmatrix} R_{sRGB} \\ G_{sRGB} \\ B_{sRGB} \end{bmatrix} = \begin{bmatrix} 3.2410 & -1.5374 & -0.4986 \\ -0.9692 & 1.8760 & 0.0416 \\ 0.0556 & -0.2040 & 1.0570 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{3.45}$$

But the sRGB tristimulus values need to be transformed to nonlinear values. This is called gamma correction and the reason behind this is that monitors display colors nonlinear and thus need to be compensated to get the desired linear relationship.

$$\begin{aligned}
\text{If } R_{sRGB}, G_{sRGB}, B_{sRGB} \leq 0.00304 : \\
R'_{sRGB} &= 12.95 \times R_{sRGB}, \\
G'_{sRGB} &= 12.95 \times G_{sRGB}, \\
B'_{sRGB} &= 12.95 \times B_{sRGB}. \\
\text{else if } R_{sRGB}, G_{sRGB}, B_{sRGB} > 0.00304 : \\
R'_{sRGB} &= 1.055 \times R_{sRGB}^{1/2.4} - 0.055, \\
G'_{sRGB} &= 1.055 \times G_{sRGB}^{1/2.4} - 0.055, \\
B'_{sRGB} &= 1.055 \times B_{sRGB}^{1/2.4} - 0.055.
\end{aligned} \tag{3.46}$$

Equation 3.46 describe the algorithm for the gamma correction [39].

3.8 Using the Fire as a Light Source

Getting the correct color of the fire is important for physical realism. But one other important aspect for physical realism is how the fire interact with its surrounding environment. Fire emits light in all directions and thus affect the perceived color of nearby objects, because they reflect the light from the fire. Calculating the color of the reflected light is done by firstly calculating color of the incoming light. This can be done by treating every voxel in the fire simulation as a separate point light source [2]. The emitted light color coming from a voxel is calculated as described in Section 3.7 and the color of the incoming light is the summation of all the voxels emitted light color.

3.8.1 Phong Reflectance Model

The Phong reflectance model calculates the reflected color of an object, the model has historically been widely used in computer graphics [40]. The phong model is designed based on empirical observations [40]. A physically based model would have been preferred instead. Phong reflectance model is depicted in Equation 3.47, it is an linear combination of ambient (I_a), diffuse (I_d), and specular (I_s) light. Specular light never enters the surface of an object and have instead been directly reflected by the object. On the contrast diffuse light have first entered the surface of the object before being reflected [40].

$$I = k_a + k_d I_d max(N \cdot L, 0) + k_s I_s (max(V \cdot R, 0))^n \tag{3.47}$$

k_a, k_d and k_s in Equation 3.47 are scalar values for controlling the relative contribution of the ambient, diffuse, and specular light. N is the surface normal, L is the direction of the incoming light and R the direction of the reflected light. R can be calculated as $R = 2N(N \cdot L) - L$ [40]. n is the shininess of the object, a perfect reflector would have a value of ∞ [40].

3.9 Volumetric Rendering

Conventionally GPU accelerated rendering of objects are done with meshes constructed out of triangle primitives. However, the data generated for visualization by the fluid simulation consist out of three-dimensional grids (the temperature and density grid). Techniques for producing two-dimensional images from these volumetric data sets whiteout explicitly extracting geometric surfaces from the data is called direct volume rendering [41].

3.9.1 Volume Ray Casting

Ray casting is a direct volume rendering method where a light ray is sent for every pixel and then traversed through the volume, gathering a number of samples along its path. Each light ray is either absorbed, emitted or scattered. The emission is generated from the black body generation that depends on the temperature, which in turn gives the color. This process is described in Section 3.7.1 and 3.7.2.

The alpha value itself is obtained from the density value of each sample along the ray. The alpha value is calculated for each sample throughout the volume by the over operation in Equation 3.48.

$$A = 1 - \prod_{j=1}^{n} (1 - A_j) \tag{3.48}$$

The number of samples depend on the occasion, but it should generally be close to the resolution of the volumetric texture.

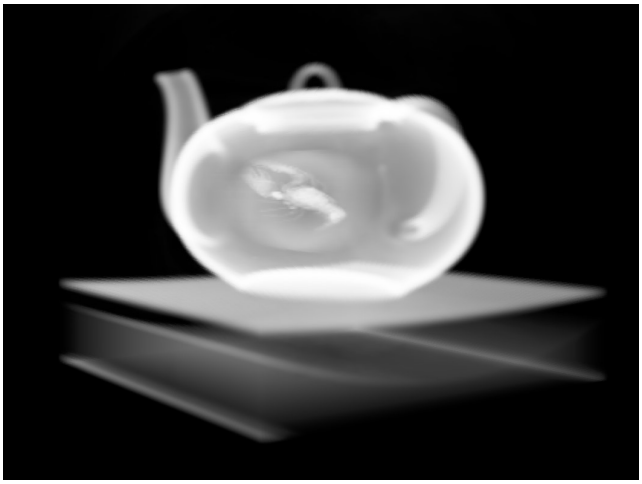

Figure 3.4: *Example result of volume ray-casting*

4 Implementation

This chapter is dedicated to describing how the application was built and how certain tools were used to contribute to the development process. The implementation is primarily based on the physical models presented in the theory chapter. These models were used to make a physically based approximation in accordance to the goals of the project.

4.1 Android

The mobile operating system Android was chosen as the main development platform due to its widespread availability on mobile devices, as well as possessing a wide variety of development tools and guides [42]. The Android application was written in the Java programming language through the Android Software Development Kit (SDK), which is the most common way of developing Android applications. The Android Studio IDE was selected for the project because of its availability on multiple desktop operating systems and being the standard development environment for Android development.

The application itself was built upon the standard fundamentals provided by the Android SDK tools [43]. One of the main components used was a so-called activity, which represents a screen with a user interface [43]. The activity component was required for creating an application entry point and initialize the renderer used by the OpenGL ES graphics API. It was also responsible for loading the native libraries used in the project.

Android offered a Native Development Kit (NDK) toolset for developing applications through native code such as C and C++, which was highly preferable when developing an application with a high-performance demand [44]. The fact that Android is built upon Java also include the virtual machine that Java is built on [45]. The choice of running a virtual machine impacts performance on Android [46], which was the primary reason for why the NDK was used over the default Java-based SDK when programming the simulation. The NDK was used in combination with the OpenGL ES graphics API in order to communicate with and perform computations using the GPU.

To run the application, either an Android emulator or a physical Android device could be used. Because of the emulator's wide range of capabilities [15], it was a suitable tool for running and debugging the application.

4.2 Modelling the Simulation

The simulation was primarily based on the Navier-Stokes equations for incompressible flow; what parts they consist of and how the differential equations were solved numerically. The problem-solving process had to be modelled in a graphics programming context in order to use the GPU in the simulation process.

The fields that represent the state of fluid, such as the velocity vector field, had to be modelled into appropriate data structures on the GPU. For this, 3D textures were used as they can both represent scalar and velocity fields. Each texel, or texture element, consists of color values, a value for each red, green and blue channel, and a value for the alpha channel. These channels were used to input values for a scalar or a vector. A texture is then made up of multiple texels, where each texel is representing the state of a field at a specific position. Multiple 3D textures, one for each field, were then used to represent the state of a three-dimensional fluid.

The simulation was performed through the graphics pipeline of OpenGL ES, which involved creating custom vertex shaders and fragment shaders. The OpenGL ES 3.1 specification contains a number of programmable stages that are customizable [47]. The ones most commonly used are vertex shaders, used for handling vertex points and data associated with them, and fragment shaders that are run for each pixel of a primitive that has been rasterized to the screen. Shaders can be summarized as programs running on the GPU and the different shaders handle different areas of the graphics rendering. The various parts of the equations, such as the advection, projection and external forces, were modeled into separate shaders that handled the corresponding part of the simulation. Shader programs, i.e. a collection of shaders, can be enabled or disabled by calling certain OpenGL functions and selecting between any compiled shader program. This explains how the different pairs of shaders for the simulation steps were executed and combined into a full simulation.

4.3 OpenGL ES

OpenGL is a low-level graphics API. OpenGL ES (sometimes shortened as GLES) is a differing version of OpenGL designed for embedded and mobile systems. While OpenGL and OpenGL ES have been equal for the projects purposes for the most part, versioning and available features may vary between the two versions. One such difference that became relevant for this project is which texture formats that can be counted as color-renderable. If a texture format is classified as color-renderable, then that texture format may be used for a texture that is rendered to through a framebuffer. As a framebuffer is used here in order to update fields during simulation, the texture format used to define the fields need to be color-renderable in order to work within the simulation. In the simulation, field values were stored as floats, which is not guaranteed to be color-renderable in GLES 3.1. Instead, GLES could check for if the specific implementation includes the extension "EXT_color_buffer_float", which if present, would guarantee that some floating-point texture formats would be color-renderable. While this additional condition reduces the applicability of this implementation, it is a fair trade to the alternative of creating a more complex implementation where units are transformed between floating-point values and a different, color-renderable format.

4.4 Utilizing the GPU

Simulating a fire, and computer graphics in general, can benefit a lot from parallelism, and therefore it is suitable to use the GPU in parts of the project. The reason behind this is that the GPU is better suited for problems that include a lot of parallelism [6].

A GPU is designed and optimized for calculations regarding graphics. The main difference between a CPU and a GPU is that a GPU usually runs at slower clock speeds but has a higher amount of threads and is intended for SIMD calculations, i.e. the GPU executes the same instruction set for multiple sets of data [48], [49]. This resembles the working principle of the simulation, where the same calculations are applied to multiple points in a three-dimensional grid. Executing the calculations on the GPU, rather than the CPU, is therefore an important factor when performance is critical.

4.4.1 GPUs Impact on Complexity

As described in Section 1.1.2 the complexity of a fluid simulation is $O(n^3)$ [4]. This is the case when the simulation is done on a three-dimensional grid with the size n of all the three the dimension. However, the GPU reduces this complexity through parallelization, because of our implementation of using the graphics pipeline, the three-dimensional textures storing the data is divided into slices of two-dimensional texture. Each one of slices can be run fully in parallel (assuming the GPU have enough threads) and thus reduces the complexity to linear time, $O(n)$, where n is the number of slices.

This could be improved upon by moving to an implementation based on a General-purpose GPU (GPGPU) pipeline over the GPU graphics pipeline. The General-purpose GPU pipeline is not limited to work one “slice” at the time, but instead can theoretically work on all the grids voxels in parallel. For this implementation in OpenGL ES, a compute shader would be used in replacement of the fragment and vertex shader. This approach could theoretically bring down the complexity to constant time, $O(1)$, assuming again there are enough threads available.

5 Process

This chapter will cover the overall process of developing the simulation. Different iterations and added features of the software are shown here, in addition to the group dynamic and experiences from the workflow.

5.1 Start of Project

The majority of the authors of this thesis had no prior experience with computer graphics, as a remedy, the project started with each group member performing a couple of basic exercises in OpenGL ES to get a better overall grasp of how to use some of the functionality it provides. The first such exercise was simply to render a triangle or a quad primitive to the screen. This exercise covered the basics of entering vertex data and using shaders. Another exercise performed further on was to calculate a new grid value for each grid element by summing all neighbors' values. The underlying reason for this exercise was to learn to access neighboring grid elements, since this would be a common operation used in most shaders of the simulation.

In the beginning of the project, an UML diagram was created, see Figure 5.1, to get a visual overview of the classes involved in the simulation. The diagram was created before any code had been written and the features to be added had not been determined either. The purpose of the diagram was to act as an entry point for designing the structure of the software.

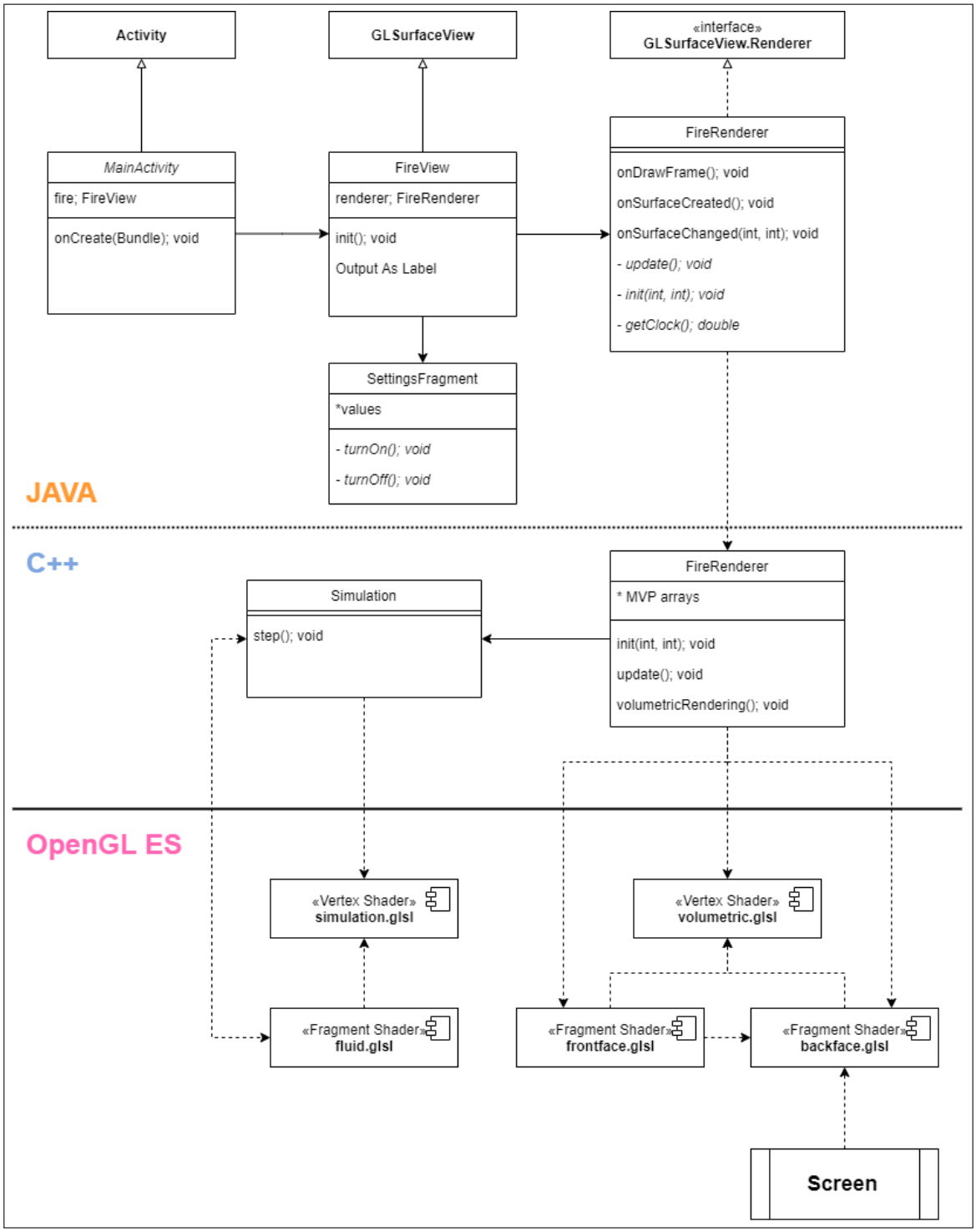

Figure 5.1: *UML diagram showing an estimated set of classes that would be used in development of the simulation.*

The development advanced by converting theoretical knowledge from reading papers into usable code run in the simulation. Since the implementation was based on approximated mathematical solutions to the differential equations posed in the Navier-Stokes equations, it was difficult to perform unit tests and make sure that every part of the simulation was working correctly. At this stage, the goal was to create a simple representation of the fire in order to receive visual feedback from the simulation and have a foundation to further build upon. After some time, visual results started to represent the looks and behavior of fire, which can be seen in Figure

5.2. These representations were used as the foundations for the rest of the project.

5.2 Previous Results

During development, screenshots were taken at multiple stages and for different versions of the fire simulation. The major feature additions are shown here and their impact on the simulation.

5.2.1 First Visual Results

The first visual results, visible in Figure 5.2, were created by combining the mathematical translation of the equations with some experimentation. The results are quite similar in appearance but with smaller differences in color and shape of flames, mainly due to each version being built upon mostly the same mathematical implementation.

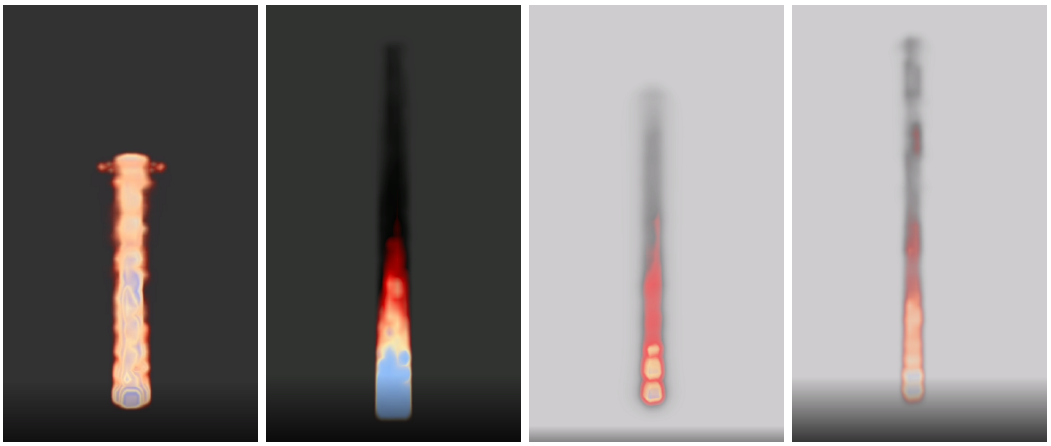

Figure 5.2: *Four different versions that were used as a foundation for the development of the simulation.*

5.2.2 Vorticity

Before vorticity was added, the flames of the fire rose upwards in a constant stream. As mentioned in Chapter 3, the approach used when simulating the fire dissipates rotational flows that are lost when simulating. When vorticity confinement was added the fire seemed more random and introduced a noticeable amount of turbulence in the fluid, that is slightly visible in Figure 5.3.

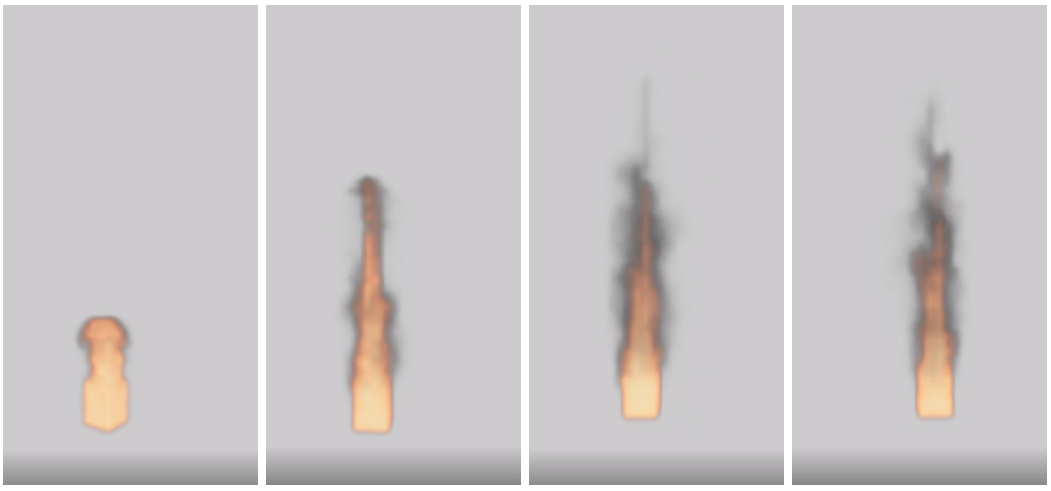

Figure 5.3: *Appearance of the fire after vorticity was added.*

5.2.3 Wavelet Turbulence

Adding wavelet turbulence into the simulation made it possible to use a higher level of detail for the temperature and density, resulting in a more detailed fire animation. Comparing the images for wavelet turbulence, from Figure 5.4, 5.5 and 5.6, with the first visual results, makes it clearer to see the added detail to the simulation.

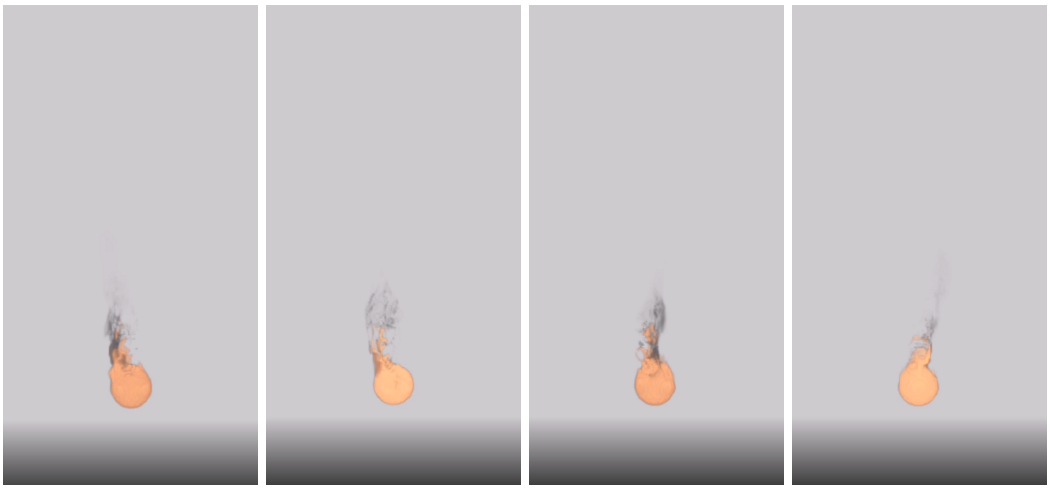

Figure 5.4: *The first basic implementation of wavelet turbulence.*

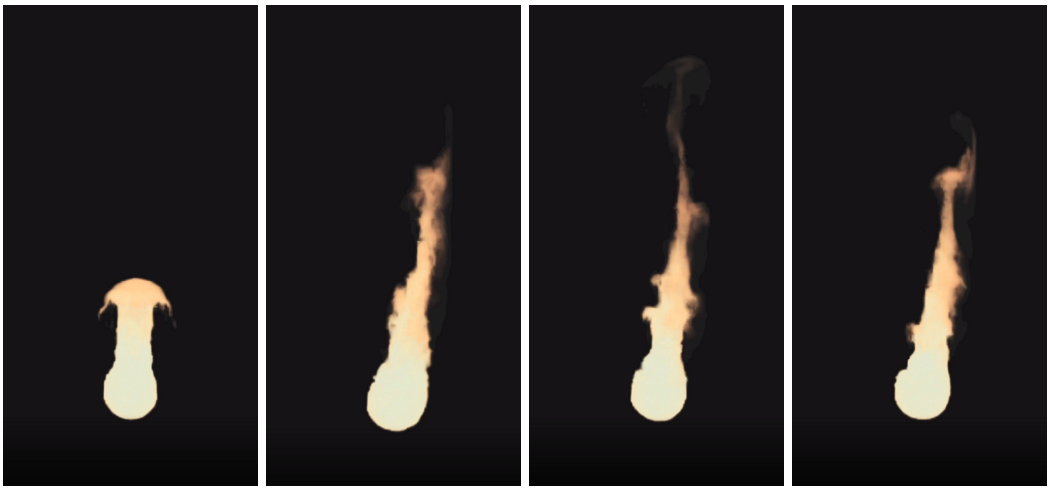

Figure 5.5: *Wavelet turbulence combined with better coloring and vorticity.*

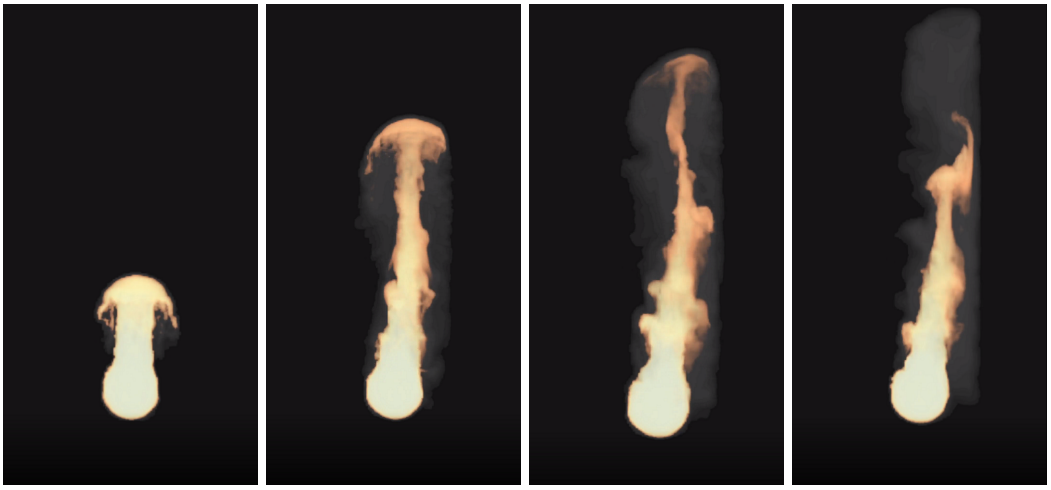

Figure 5.6: *Wavelet turbulence, same as Figure 5.5 but with added smoke.*

5.2.4 Black-body Radiation

Black-body radiation was introduced to mimic the visual property of a hot fluid illuminating the smoke of the fire.

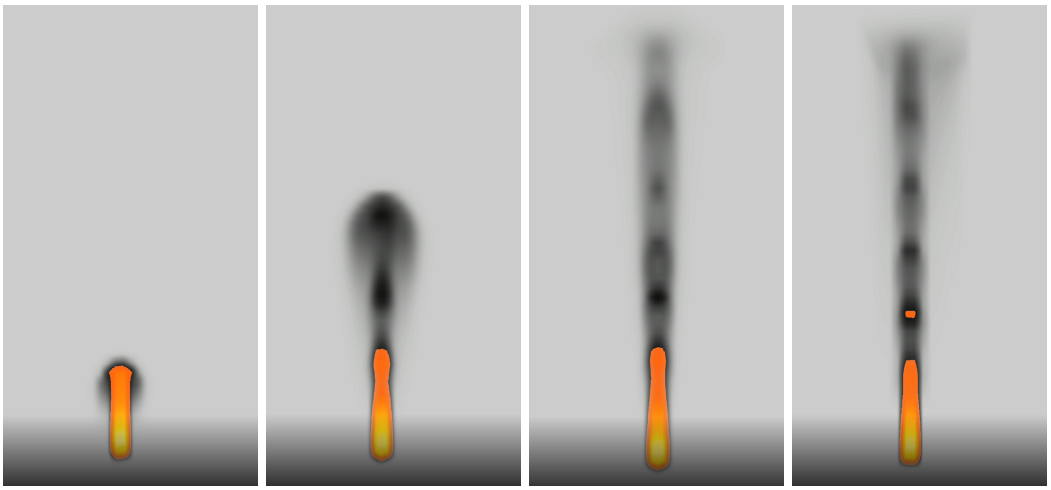

Figure 5.7: *Illuminated effect of the smoke after implementing black-body radiation for an early basic version of the simulation.*

Figure 5.8: *Posterior version of blackbody radiation with later features added.*

5.3 Added Features

The following sections display features that were implemented as an addition to the project to increase the interactivity and notion of realism of the fire.

5.3.1 Touch Interaction

The addition of touch interaction allowed for interacting with the fire by adding a directed force to it. The force direction corresponds to the direction of the user's swipe, dragging a finger across the touch screen. These effects are visible in Figure 5.9, where the visual response can be seen to a user swiping in various directions. This was simply done by adding the swipe as an external force to the velocity step mentioned in Section 3.2.1.

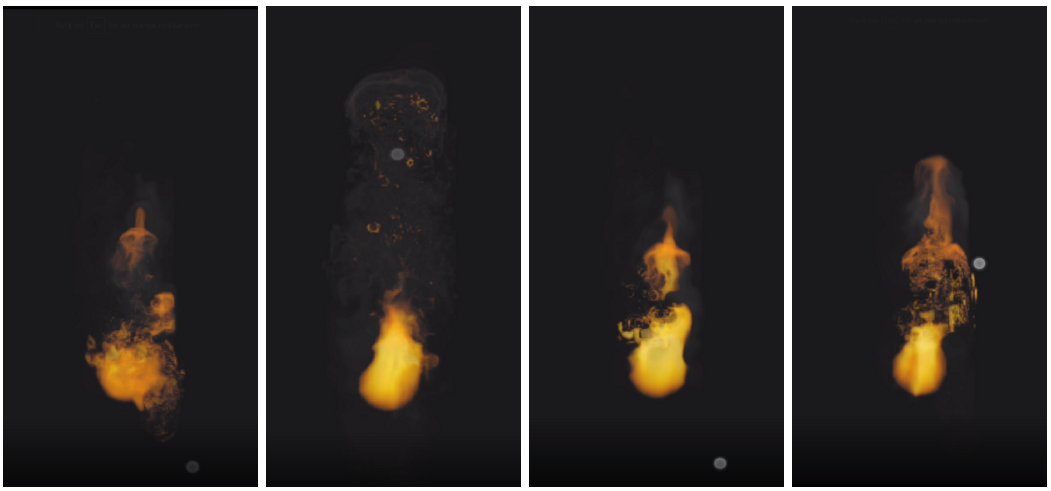

Figure 5.9: *Visual response after adding touch interaction to the fire simulation. The small dot in the picture resembles the touch interaction.*

5.3.2 Light Reflective Walls

Light reflective walls were added to make the simulation more visually appealing and to make the fire look more alive by having it interact with its environment. The result can be seen in Figure 5.10. Unfortunately, due to time constraints the implementation is not complete, for example the incoming light is only calculated for one wall, but all three of the walls in Figure 5.10 use this result for their reflection.

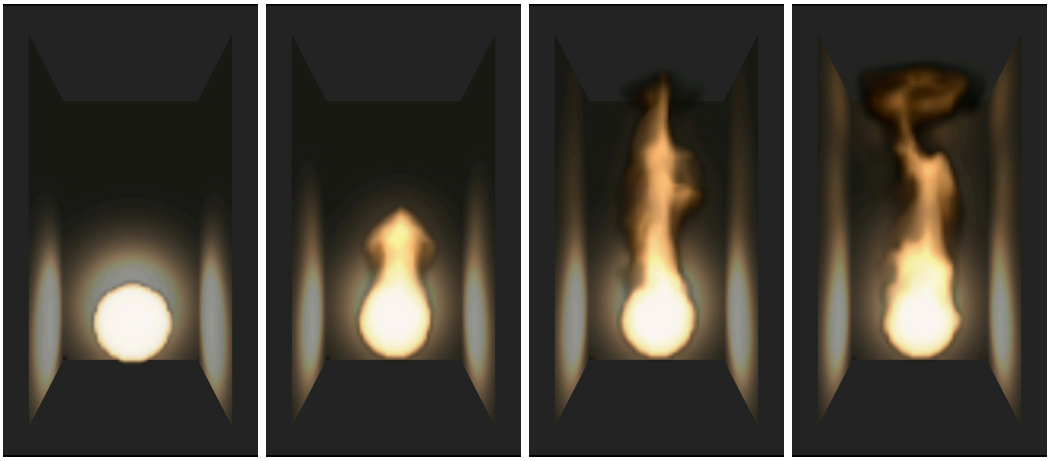

Figure 5.10: *Fire simulation with light reflective walls.*

5.4 Workflow

Approaching the task of simulating a fire on a handheld device, two internal groups were formed; one focusing on the simulation aspect, and one focusing on rendering. Between these, the group was split under the ratio 4:2.

Meetings played a big role in communicating with other group members, distributing tasks and planning future work. Supervisor meetings were held once at the beginning of every week, so that feedback could be received based on the previous week's work. Regular group meetings were held twice a week, where each meeting followed a premade meeting template that contained sections for last meeting topics, what has been done, what went well/bad, among other things. Each meeting protocol was documented and saved to simplify the process of recalling earlier meeting decisions and discussions. A work session usually followed each meeting, where the group members could continue working on parts decided during the meeting. This was also a chance to collaborate with other group members more efficiently, as most of the time was spent working from home.

The workflow was inspired by agile methods such as Scrum. Before development had begun, weeks for each sprint were planned in advance where each sprint duration was roughly two weeks. The agile iterative principles were followed in meetings, going over what had been accomplished retrospectively and planning what should be completed in the upcoming sprint. The tool Trello was used when creating tasks, putting them into either a backlog or sprint task column. In Trello, tasks were assigned to various group members and moved to different columns like ”testing” or ”done” depending on the progress of the task. Moreover, each task's completion time was estimated in order to approximate the workload.

Code for the software was managed using Git and GitHub. The master branch was intended for stable versions with complete features most of the time. New features were developed on separate branches that were branched out from the master branch. Completed features were then integrated by submitting a pull request and assigning a reviewer to review and accept the change. In most cases, the reviewer was a group member responsible for code submissions in either the internal simulation or rendering group.

6 Results

6.1 Overview of our Application

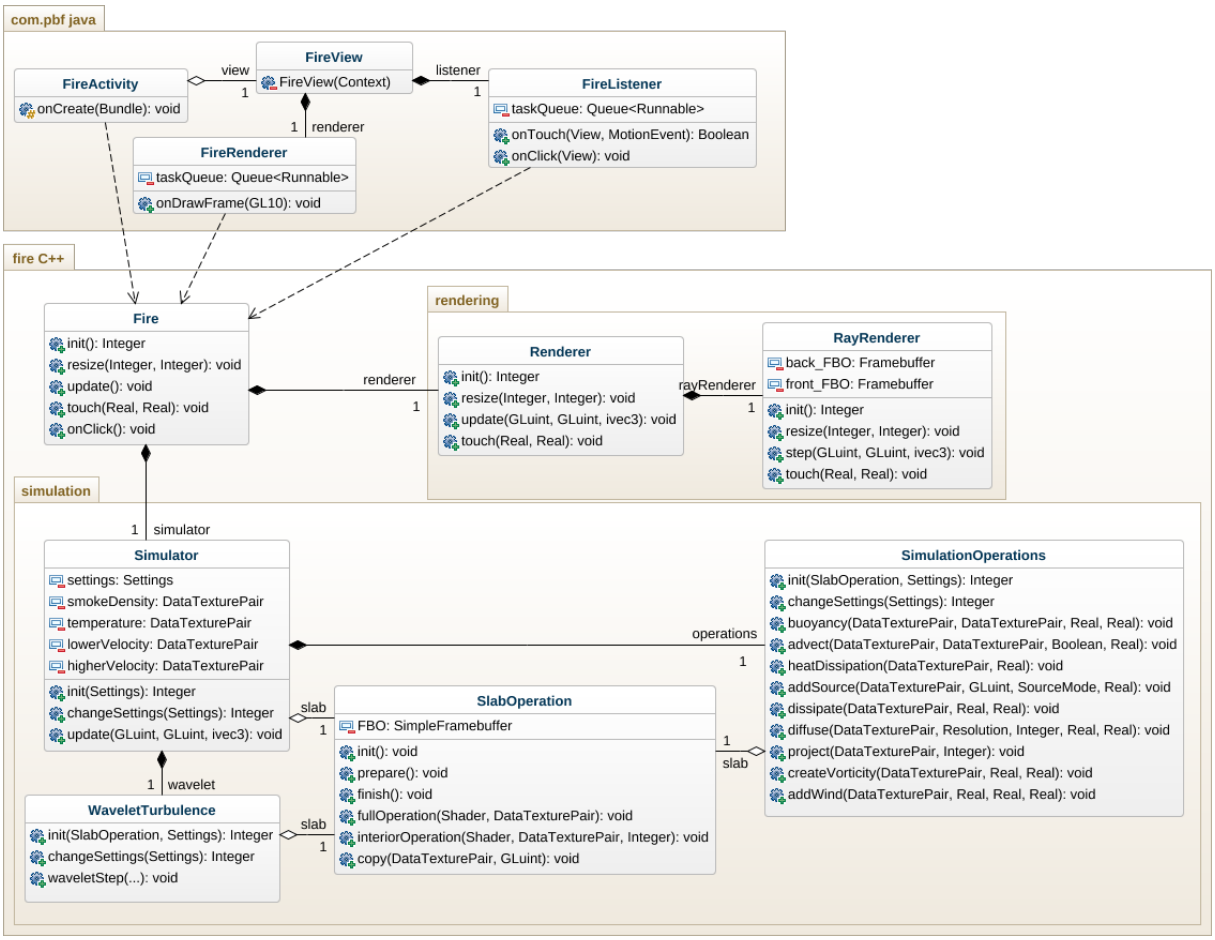

Figure 6.1: *An UML-diagram showcasing the most important parts of the resulting application*

Our finished application is written over three different languages; Java, C++ and GLSL. The Java portion of the application acts as an intermediary step between the Android system and the animation application, which primarily exists in the C++ portion. It is in the Java class FireActivity which the application is being created and initialized, while FireListener and FireRenderer is forwarding processed user input and render updates respectively to the C++ portion.

This C++ portion of the application is primarily divided among three packages; rendering, simulation and util. Among the important classes in the util package are the data texture pair, framebuffer and shader. The data texture pair is a construct consisting of a data texture, a result texture, together with some helper functions. As nearly all operations consist of a texture being updated based on the previous data of the texture, the data texture pair is used to provide an intermediary texture for the updated data from an operation that shares the same texture settings as the original data. This, while also offering a simple and isolated way of switching out the data texture with the result, is why this object is used for data fields such as velocity, temperature and so on. As operations are written as GLSL shaders, we're using a shader class to both compile and load a shader program based on the location of the shader files, but also as a container for the shader program reference. The framebuffer class is a helper implementation of an OpenGL framebuffer, complete with a texture and a renderbuffer, which used as part of the ray tracing of the rendering portion of the application.

The simulation is built upon three layers. The first layer, which is in the simulator class, holds the overview of the simulation. It keeps track of the velocity, temperature, and density textures, and it defines which operators that are used during the simulation of each of these fields. The second layer consists primarily of the simulation operations class, but also the wavelet turbulence class. These classes define all individual operations, and passes the data needed by these operations to OpenGL. The code for the operations themselves is coded in GLSL and are stored separately from the C++ code. The third layer is the slab operation class, which handles those parts that are common between operations. This consists of performing the actual render operation, which has to be done for each texture layer individually. It also maintains the border of the fields that need it. As for the rendering portion, it became small enough to be contained solely in the ray renderer class.

In the simulation, three different types of sizes are defined. The first size type is called simulation size, which is the physical size of the simulation space. While the simulation size is noted to be measured in meters, specific units are used rather loosely throughout the simulation. The important part is that it is defined independently from the resolution of the textures, such that both the simulation size and the resolution can be changed without affecting the other. As for the two resolution sizes, the lower resolution is used for simulating the velocity, with a higher resolution for most other fields. With this difference in resolutions, wavelet turbulence could be used as a method to scale up the velocity to then be used during advection of the other substances.

6.2 Performance and Visuals

We performed different tests according to Section 2.5 to find out how the simulation performs, and how the resulting fire looks when using different resolution sizes. The following sections describe the major performances with the different approaches explained in Chapter 3. For each test, arbitrary samples are chosen to get an overview how the grid resolution affects the different aspects of simulation.

6.2.1 Resolution

The resolution plays a big part of the performance, since the simulation perform its calculation per grid cell. So, we performed two tests to see if the height of the grid did affect the overall performance of the simulation, and how the result looked. The different resolutions and their visuals that we tested can be seen in Appendix A.

The graphs in Figure 6.2 and Figure 6.3 shows how the resolution size correlates to aspects such as time per frame and memory usage, and the data we gathered for the graphs can be found in Appendix A. Our results from this test shows that the correlation between the time it takes to render a frame in the simulation, as well as the memory usage is directly dependent on the resolution size. The result, which we can be seen in Appendix A, also showed that the CPU utilization remained at 10% throughout all tested resolutions for both grid size ratios, implying that the CPU is unaffected by the grid sizes. An interesting observation is that the height scaling ratio of 1x2x1 and 1x4x1 perform fairly similar for low resolutions, and the ratio of 1x4x1 rendered frames faster than the ratio 1x2x1 initially.

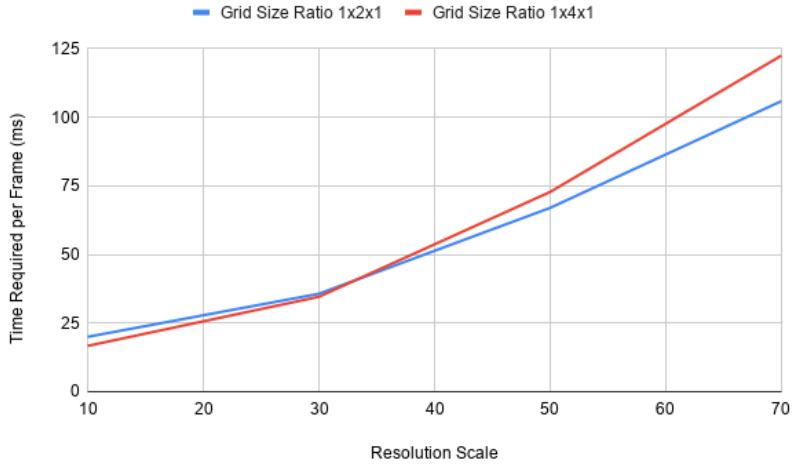

Figure 6.2: *Graphing how grid resolution affects the time required to render a frame of the simulation. The total grid size is achieved by multiplying resolution scale with grid size ratio*

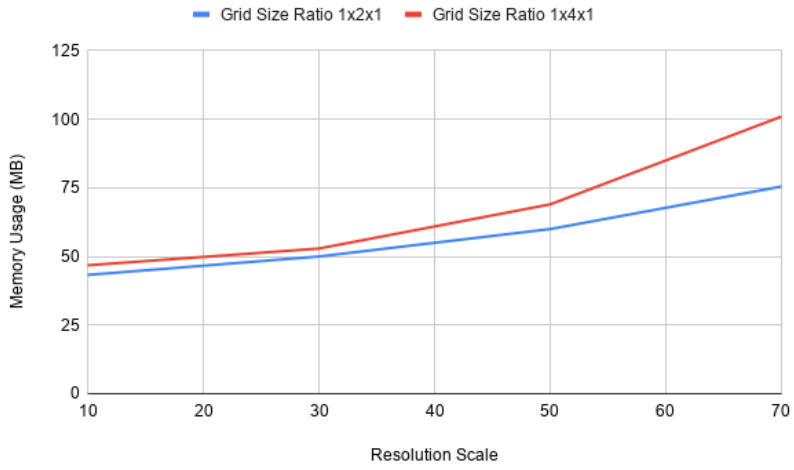

Figure 6.3: *Graphing how grid resolution affects the memory usage of the simulation. The total grid size is achieved by multiplying resolution scale with grid size ratio*

From a visual aspect, the lower resolution had a lot of motion but became more fixed as the resolution increased. This implies that the chaotic behavior decreased as the resolution became higher.

6.2.2 Navier-Stokes vs Euler

In Section 3.2.1, we presented two approaches for simulating the behavior of a fluid. The first approach consisted of solving the Euler equation for an incompressible fluid, whilst the other approach consisted of solving the Navier-Stokes equations. The performance of the two approaches can be seen in Figure 6.4 and Figure 6.5, where the performance is plotted in comparison to each other. The data for each approach can be found in Appendix A and Appendix B, where A presents the Euler approach and B represents the Navier-Stokes approach.

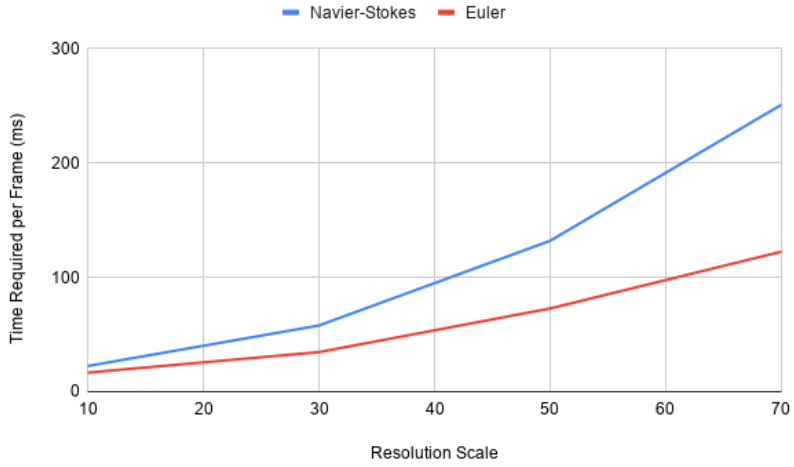

Figure 6.4: *Graphing the performance of the Euler simulation approach and Navier-Stokes simulation approach in regards of time per frame when using a grid scale ratio of 1x4x1*

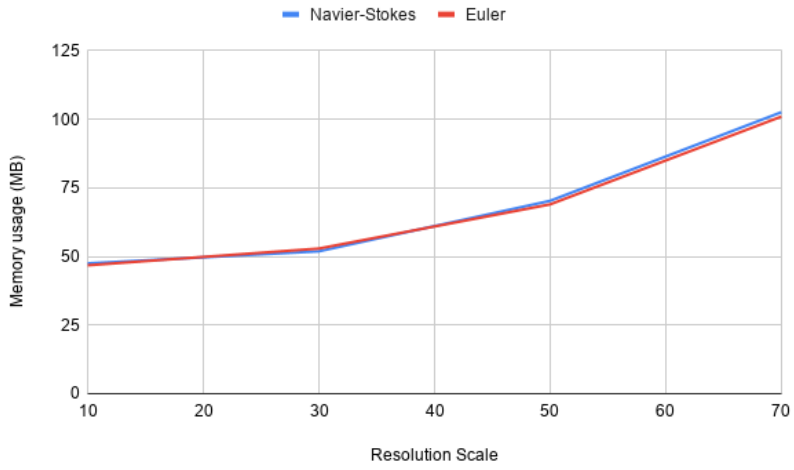

Figure 6.5: *Graphing the performance of the Euler simulation approach and Navier-Stokes simulation approach in regards of memory usage when using a grid scale ratio of 1x4x1*

Our results from this test shows that the Euler approach is rendering the frames quicker than the Navier-Stokes approach, while both memory usage and CPU utilization are equal. The difference in the visual results from each approach is minor in respect to the visuals displayed in Appendix A and Appendix B.

6.2.3 Emitters

We tested four different types of shapes to see if the shape of the source affected the overall performance and/or the visualization of the simulation. When we tested on a grid resolution of 32x122x32, all shapes had the same calculation time per frame and CPU utilization. The memory usage did differ by 1.5 MB, where the sphere and the cylinder demanded more memory. Despite the memory difference, the shape of the emitter did not affect the overall performance of the simulation. Figures 6.6, 6.7, 6.8, and 6.9 show the visuals of the shapes we tested.

Figure 6.6: *Cube rotating 22,5 degrees with every picture*

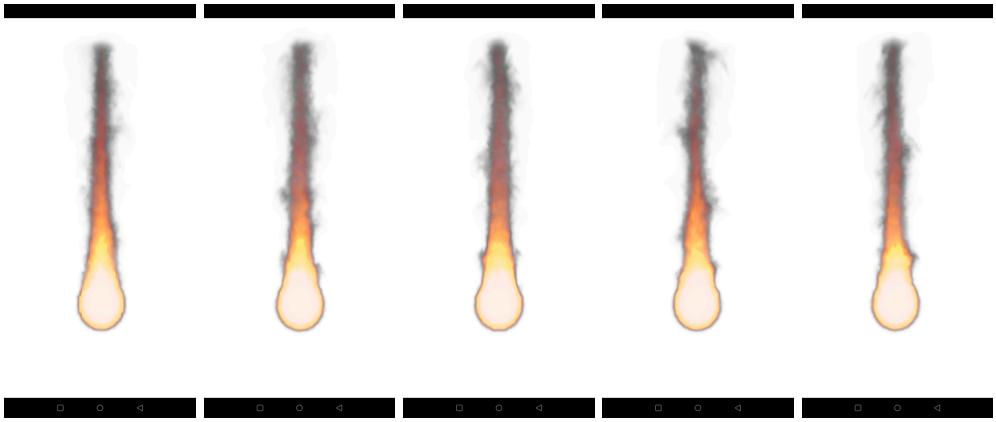

Figure 6.7: *Sphere rotating 22,5 degrees with every picture*

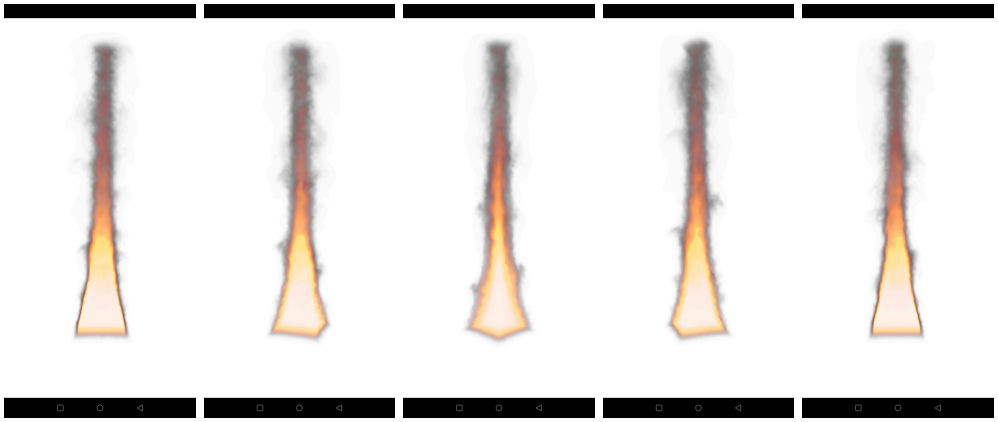

Figure 6.8: *Pyramid rotating 22,5 degrees with every picture*

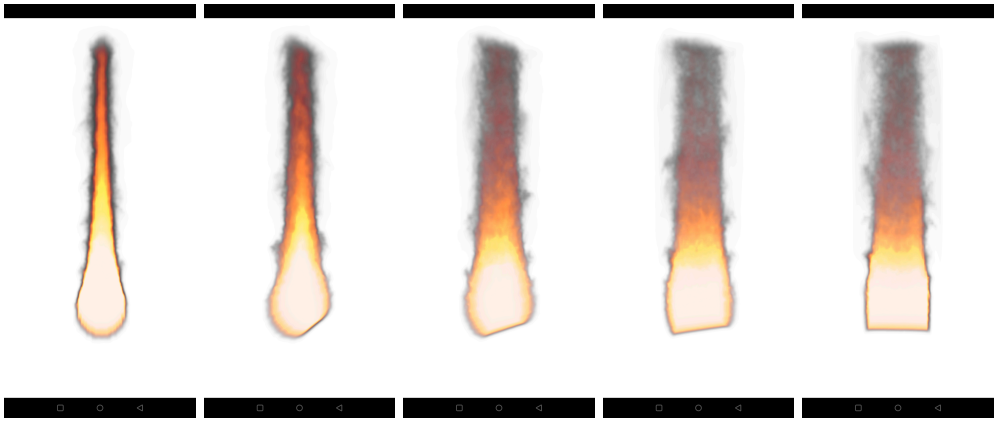

Figure 6.9: *Cylinder rotating 22,5 degrees with every picture*

6.2.4 Wavelet Turbulence

We used wavelet turbulence to optimize the performance for the fire while still maintaining high detail. In Figure 6.10, 6.11 and 6.12, we see the difference in performance between the original simulation with a 1x4x1 ratio and wavelet with the same simulation with a 1x4x1. Whilst the simulation with wavelet turbulence had the same texture size for density and temperature, it had a lower texture size of 12x42x12 throughout the test. The data we used in the graphs may be seen in Appendix C, where visuals of wavelet turbulence also can be seen.

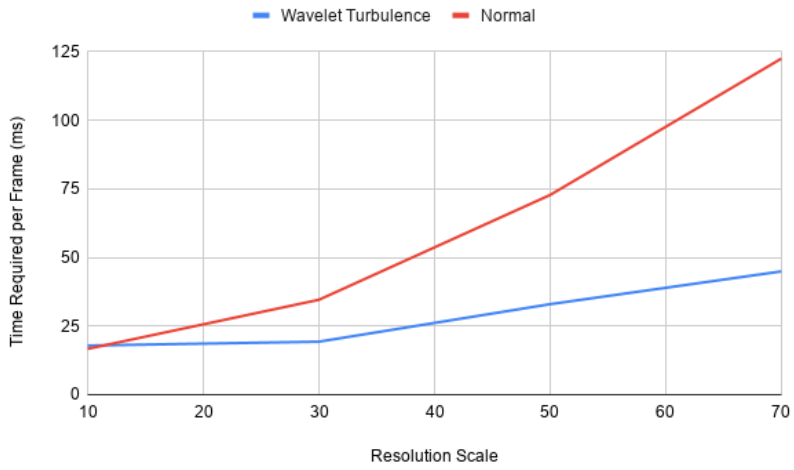

Figure 6.10: *Graphing the performance of the Wavelet Turbulence optimization in comparison to the original simulation approach in regards of time per frame*

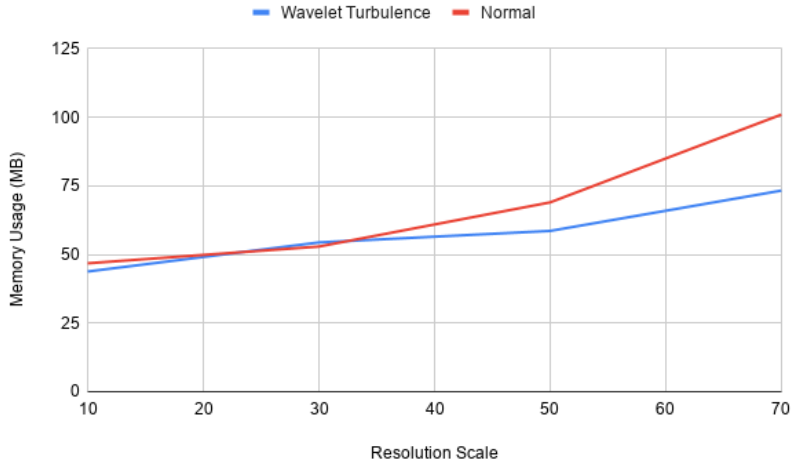

Figure 6.11: *Graphing the performance of the Wavelet Turbulence optimization in comparison to the original simulation approach in regards of memory usage*

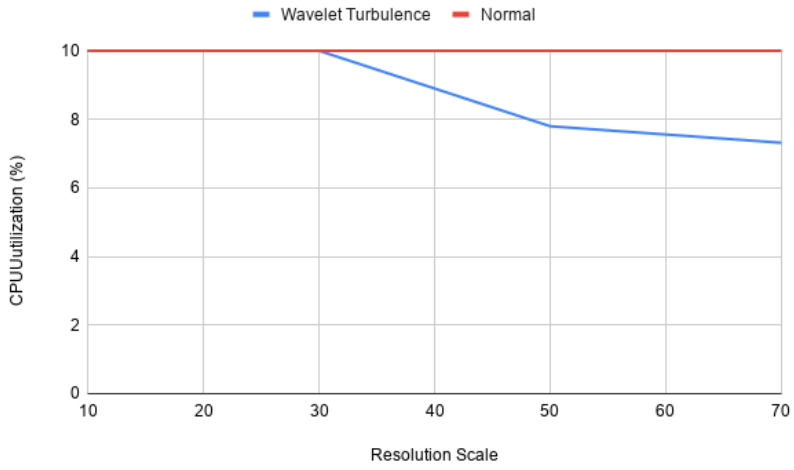

Figure 6.12: *Graphing the performance of the Wavelet Turbulence optimization in comparison to the original simulation approach in regards of CPU utilization*

Our results from this test shows that wavelet turbulence improved the performance in respect to the time required per frame, memory usage, and CPU utilization. The visuals of the fire improved as well, as the notation of natural behavior was kept throughout all resolutions.

7 Discussion

The following sections will deeper discuss issues and aspects such as comparison between the Euler and Navier-Stokes equations and their impact on the performance, wavelet turbulence and possibilities to further develop the project. This chapter also deals with ethical aspects and the need of greater mathematical knowledge.

7.1 Interpreting the Result

The results gained from the test in Section 6.2.1 showed the limitation of using a mobile device to render a simulation based on fluid dynamics. This test case showed that the memory consumption and CPU utilization did not really matter. Nevertheless, the sought performance should consist of 30 to 60 FPS, implying that a frame should be rendered in less than 33 ms. Only two of the samples would be able to accomplish the sought goal, the grid resolutions of 12x22x12 and 12x42x12. Nonetheless, neither of the two grid resolutions visuals showed the sought amount of detail, consequently showing that these samples was not accurate enough. The resolution of 32x62x32 and 32x122x32 received 29 FPS and got more detail than the first two samples. However, as stated in Section 6.2.1, the visual behavior of the flames in these approaches became more static which did not meet the expectations of the authors. The reason of the static behavior is based on the grid being too detailed for the velocity grid, resulting in the vorticity confinement getting less apparent.

As could be seen in Section 6.2.1 as well, the difference in performance of the two grid ratios was not apparent. The grid ratio of 1x2x1 did however perform better in the long run when comparing the grid ratio of 1x4x1, but the visual result of ratio 1x2x1 was unpleasing as the flame did not have as much room to move in the y-axis of the grid. This resulted in choosing the grid ratio of 1x4x1 for the rest of the tests in Section 6.2, as it would provide the flame to move more freely.

In Section 6.2.2, the result showed that the Euler approach performed better than the Navier-Stokes approach in terms of time per screen. While this performance-wise difference was expected as the Euler approach skipped one costly step in the simulation, the more interesting fact is how the approaches differ in appearance and how accurate the simulation came across compared to previous works. The major difference in appearance when running the two approaches side by side, was a notion of inertia in the movement could be seen in the Navier-Stokes approach. Nonetheless, this difference did not improve the realism of the simulation. Thus, the majority of the tests in Section 6.2 was done using the Euler approach. When comparing to previous works, the resulting fire did not look as physically correct as one could hope for when the grid resolution was increased.

When wavelet turbulence was utilized, the performance of the simulation improved, as well as the CPU and memory usage. However, the most important part was that the flame got high amount details while still maintaining good performance. The largest resolution that still meet the expectations of 30 FPS was the grid resolution of 52x202x52, which had sufficient amount of detail while still behaving like a fire. Throughout every resolution that was tested, the flame preserved a unique and chaotic behavior. This was expected as the velocity grid had a lower resolution than the grids representing the temperature density, resulting in one velocity vector affecting more cells in the respective scalar grid. The vorticity confinement added the extra fluttering motion in the flames behavior, resulting in the chaotic behavior. As the resolution for the scalar grids became higher, the amount of detail in the smoke increased massively, and started looking like previous work from researchers in the field.

Interestingly, the shape of the emitter did not affect the performance, CPU or memory utilization enough for an thorough analyze throughout all grid sizes. However, as could be seen in Section 6.2.3, the visuals did vary.

When all of the results was examined, one could conclude that the CPU usage did not vary at all, except a little bit when using wavelet turbulence, and the memory consumption did not come near the full capacity of the device, which can be seen in Table 2.1. Hence, the limit of the mobile device might be how fast it is able to compute all the operations. While the simulation was developed, a few factors did impact the performance. The small amount of bandwidth in a handheld device turned out to be a consistent problem throughout the development. This means that formats for storing values had to be chosen carefully. For instance, when storing float values for the textures, the bigger format was able to store 32 bytes of information. This however proved to lower the performance vastly, which forced the usage of a format that could store 16 bytes of information. Fortunately, this information loss did not drastically affect the visual result, and instead

increase the performance.

7.2 Limitations of the Workflow

The following subsection describe some hindrance encountered during the course of this project.

7.2.1 The Field of Fluid Dynamics

The field of fluid dynamic was relatively new to all of the authors in this thesis. This proved to be a challenge, considering that the majority of the group only taken an introductory course in linear algebra. The papers and other sources used in this thesis were often well written with a logical structure describing the idea of the paper, but most of the sources expected the reader to have good prior knowledge. It became a challenge for most group members to approach these papers and understand the important concepts, consequently rewarding the group with knowledge in advanced linear algebra together with experience with these kinds of papers.

7.2.2 Accessibility

During development, making the application cross-compatible was considered. Not all group members had the Android operating system running on their mobile devices, thus making the application cross-compatible would have made it possible to run the application on mobile devices running other operating systems. Hence, some investigation began to find features or potential cross-platform frameworks, resulting in finding Qt. The framework Qt [50] is a cross-platform IDE with built-in tools for developing applications compatible with multiple operating systems. A basic OpenGL project was setup with Qt to test its functionality and development process. After deliberating within the group, it was decided that Qt was not to be used in further development. The main reason for this decision was that the process of integrating the Qt tools into the project was considered too time consuming. A large part of the time spent implementing other features would have been impacted, hence the reason for sticking with developing for Android exclusively. Retrospectively, developing with Qt from the start of the project would have been another alternative as the code would not have to be migrated from one framework to another.

7.2.3 Faulty Approaches

When deciding how to split the work within the group of this project, it was decided that two members were to primarily work on the rendering, while the remaining four members were to primarily work on the simulation. In hindsight, a ratio of 5:1 with five members working on the simulation might have been a more realistic and better suited choice, as the simulation ended up taking larger focus than initially expected. It could also have been wise to define areas of responsibility as opposed to defining sub-groups. This could likely have been mitigated if the group had a better understanding of the scope of the program, to be able to more thoroughly plan the project early on. More planning could also have been done in regards to the UML-diagram. As the planning of the simulation step process was limited, the entirety of the C++ code specifically made for the simulation was practically in one single class for the first half of the project. A code refactor was done thereafter, where the simulation was refactored into three different abstraction layers, while adapted to adhere to design principles such as *Separation of concerns (SoC)* and to some degree also *Don't repeat yourself (DRY)*. In a project with multiple people working more or less in parallel, difficulties may arise when performing a significant refactor in that manner, as differences and conflicting changes between various versions of the project can grow to be rather difficult to handle.

7.2.4 Unfinished Features

Among the features that were implemented, some of them were not sufficiently finished at the time of the first launch of the application. These include touch interaction, where the fire would react when swiping a finger across the screen. Another feature is light reflective walls, where walls surrounding the fire would make the simulation look more alive.

7.2.5 Ongoing Pandemic

Like the majority of other bachelor's thesis groups at Chalmers (and people across the world) this project has been affected by the Covid-19 pandemic as well. This resulted in no physical meetings or group work sessions, which has negatively affected the workflow because of the difficulties that follow with working remotely, especially when considering what was explained in Section 7.2.2. However, with the nature of the project, the effects of the pandemic on the project had not gone beyond the limitation of these physical meetings.

7.3 Future Work

In this thesis, a handful of algorithms have been used to solve or approximate complicated equations. Most of them were chosen in respect of their ease of implementation, as well as their simplicity. One of those are the Jacobi iteration, which solves the Poisson equation for the pressure and the diffusion. Although providing great result, it demands at least 20 iteration per cell of the grid. This is rather costly, especially considering that each iteration has to update all grid cells before next iteration may start. Hence, a future problem statement could be to improve this step, eventually trying another approach than the Jacobi iteration. Another iterative method used in this project is the QR algorithm, which calculates the eigenvalues for the scattering in Wavelet Turbulence. Despite not being as costly as the Jacobi iteration, as it is able to iterate inside the GPU and not wait for all other cells to be done, it could still potentially slow down the performance. Furthermore, if the QR algorithm does fully converge after its iterations, it may provide a minor numerical error for the solution of the eigenvalues. Therefore, another problem statement for a future project could be to find a better algorithm for calculating the eigenvalues.

The potential performance improvements of an implementation based on a GPGPU pipeline over a more traditional GPU graphic pipeline (from linear to constant time complexity), discussed in Section 4.4.1, make it an area worthy of future investigation. This potential performance improvement could allow for a focus on greater physical realism, by using a higher resolution simulation or implement algorithms with higher order accuracy on their approximations. The result of this thesis would suggest that substantial improvement in these aspects is not feasible at the moment, due to the performance available on today's mobile phones in conjunction with the performance constrains set by targeting a minimum frame rate of 30.

One aspect that would be interesting to further investigate is how the fire simulation affects the battery on a phone. Since the measuring tools used in this project, the profilers, did not show energy consumption there was no straightforward way to measure this. Additionally, to run the application during the development phase, the phones had to be connected to a computer, which led to them being charged throughout the testing and therefore further complicated the measuring of the application's power consumption.

The unfinished functionality to have walls that reflect the light from the fire, is another subject that, can make use of future research in order to gain an understanding of the feasibility of running a completed implementation on a mobile device in accordance with the performance constraints described in this thesis.

An interesting work in the future would be to make the same tests as done in Section 6.2 in some years after the publication of this thesis, to see how the performance would differ. The GPU might be able to utilize more threads, and have a larger bandwidth to work with, optimizing all of the calculations.

7.4 Ethical Aspects

Since the application neither stores personal or sensitive data, it is hard to predict distinct negative consequences from an ethical point of view. The code of the simulation is under MIT-license and is open-source. The reason is to make it possible for others to use the implemented code as a base for further development. The fact that the code is open-source and available for anyone to read, would unfortunately also make it possible for developers which do not have their users best interest in mind, to use the implemented code as a base for applications that stores personal or sensitive data in an irresponsible manner. The authors expectation and intention are that the code will be used in a responsible context with the users best interest in mind.

7.5 Learning Outcomes

During the course of this project, a vast amount of experience was gained by the group members. Aspects like working with people with different academic background and agile work process proved to be rewarding and a fun experience for all group members. These experiences will undoubtedly be used in the future by all group members.

As Section 7.2.1 stated, the group encountered a lot of advanced linear algebra that nobody was familiar with. Despite this being one of the biggest obstacles during the process of this project, the group members still feel like this have been a great experience to reiterate the old knowledge of linear algebra, as well as gaining new understanding in the field. A related experience has been to evaluate and understand field-specific papers that contain information that is hard to comprehend. During the course of this project, all group members improved upon reviewing academic texts and finding the needed information to progress in the project.

Another experience gained from this project comes from developing for Android, especially put together with the NDK. The majority of the group were before this project not particularly familiar with development for Android, or in the programming language C++. Although the programming language was not deemed as a major hindrance when working on this project, it still required new ways of approaching certain problems when addresses and pointers was introduced. However, this experience was appreciated by all group members.

The last learning outcome, and undoubtedly the biggest experience for all the group members, has been to work with computer graphics. While a half of the group have taken the introductory course in computer graphics, everyone learned new things through this project. The ability to use a shader to solve complicated equations in parallel showed the advantages with an GPU.

8 Conclusion

The purpose of this thesis was to explore the possibilities and limitations of simulating and rendering a physically based fire on handheld devices. This has been achieved by using known formulas of fluid behavior and by adding features that make the simulation visually pleasing. The simulation was discretized with use of voxels in different vector or scalar fields, in order to approximate solutions to the fluid equations and use them in a visual representation.

The simulation has taken advantage of the computing power present in modern mobile GPUs to improve performance. Rendering the simulation in 3D greatly impacted performance and frame rate. When the simulation was run on the OnePlus 7, great visual result similar to previous works could be achieved while still having the sought performance of 30 FPS. For a visual comparison one can look at the paper "Physically Based Modeling and Animation of Fire" [1]. The optimal grid size resulted to be around 550 000 voxels while the grid ratio was 1x4x1 and using wavelet turbulence as an optimizing method. However, this resolution would only be feasible for this device only. Older mobile devices, with less capable hardware might not be able to reach the initial goal of 30 FPS. In addition, the simulation would not be very useful in a mobile game as the frame rate would be heavily reduced rendering a single fire in its current state. Adding this in combination with other game elements, the performance would be affected to the point where it is most likely unplayable.

wavelet turbulence proved to increase the performance greatly, resulting in a more physically plausible appearance of the fire, as well as providing relatively good performance compared to the original approach.

Regarding the last problem statement stated in the introduction, if the simulation could be interactive; in Section 5.3, a touch feature was introduced that made it possible to interact with the fire by adding force to the velocity field in the direction of the user touch input.

References

[1] D. Q. Nguyen, R. Fedkiw, and H. W. Jensen, *Physically based modeling and animation of fire*, Stanford University, 2002. [Online]. Available: http://physbam.stanford.edu/~fedkiw/papers/stanford2002-02.pdf (visited on 2020-02-13).

[2] K. Crane, I. Llamas, and S. Tariq, *GPU Gems 3*. Addison-Wesley Professional, 2007, ch. 30, pp. 633–675, ISBN: 978-0-321-51526-1. [Online]. Available: https://developer.nvidia.com/gpugems/gpugems3/part-v-physics-simulation/chapter-30-real-time-simulation-and-rendering-3d-fluids (visited on 2020-02-14).

[3] S. Green and C. Horvath, *Flame on: Real-time fire simulation for video games*, 2012. [Online]. Available: http://on-demand.gputechconf.com/gtc/2012/presentations/S0102-Flame-on-RT-Fire-Simulation-for-Video-Games.pdf (visited on 2020-02-13).

[4] J. Stam, *Stable fluids*, 1999. [Online]. Available: http://movement.stanford.edu/courses/cs448-01-spring/papers/stam.pdf (visited on 2020-02-13).

[5] M. Wrenninge, H. Fält, C. Allen, and S. Marshall, *Capturing thin features in smoke simulations*, sony pictures imagework, 2011. [Online]. Available: http://library.imageworks.com/pdfs/imageworks-library-capturing-thin-features-in-smoke-simulation.pdf (visited on 2020-02-13).

[6] M. J. Harris, *GPU Gems*, 5th ed. Addison-Wesley, 2007. [Online]. Available: https://developer.download.nvidia.com/books/HTML/gpugems/gpugems_ch38.html (visited on 2020-02-13).

[7] J. McCaffrey, "Exploring Mobile vs. Desktop OpenGL Performance", *OpenGL Insights*, P. Cozzi and C. Riccio, Eds., New York, USA: CRC Press, 2012, ch. 24, pp. 337–352, ISBN: 978-143989376-0. [Online]. Available: http://xeolabs.com/pdfs/OpenGLInsights.pdf (visited on 2020-02-13).

[8] Khronos, *OpenGL ES Overview*. [Online]. Available: https://www.khronos.org/opengles/ (visited on 2020-04-29).

[9] Qualcomm Technologies, *Identify Application Bottlenecks*. [Online]. Available: https://www.khronos.org/opengles/ (visited on 2020-04-29).

[10] L. Prechelt, *Technical opinion: Comparing java vs. c/c++ efficiency differences to interpersonal differences*, University of Karlsruhe, 1999. [Online]. Available: https://dl.acm.org/doi/fullHtml/10.1145/317665.317683?casa_token=BV5Q2b5QAUsAAAAA:5ZS5fJrFDOMg4Eg7G9koKtAZ_wyvs9Cx_n12iRMEoFVRiZda2TV3LOz94XPxduD88zhLS5vQwNGGCmo (visited on 2020-02-14).

[11] T. Kim, N. Thürey, D. James, and M. Gross, *Wavelet turbulence for fluid simulation*, Cornell University, 2008. [Online]. Available: https://www.cs.cornell.edu/~tedkim/WTURB/wavelet_turbulence.pdf (visited on 2020-02-13).

[12] B. Kim, Y. Liu, I. Llamas, and J. Rossignac, *Flowfixer: Using bfecc for fluid simulation*, 2005. DOI: 10.2312/NPH/NPH05/051-056.

[13] A. Selle, R. Fedkiw, B. Kim, Y. Liu, and J. Rossignac, *An unconditionally stable maccormack method*, Stanford University, 2007. [Online]. Available: http://physbam.stanford.edu/~fedkiw/papers/stanford2006-09.pdf (visited on 2020-02-13).

[14] Android Developers, *Meet Android Studio*. [Online]. Available: https://developer.android.com/studio/intro (visited on 2020-04-20).

[15] Android Developers, *Run apps on the Android Emulator*. [Online]. Available: https://developer.android.com/studio/run/emulator (visited on 2020-04-29).

[16] Android Developers, *Configure hardware acceleration for the Android Emulator*. [Online]. Available: https://developer.android.com/studio/run/emulator-acceleration (visited on 2020-04-29).

[17] Android Developers, *Getting Started with the NDK*. [Online]. Available: https://developer.android.com/ndk/guides (visited on 2020-04-28).

[18] Android Developers, *Concepts*. [Online]. Available: https://developer.android.com/ndk/guides/concepts (visited on 2020-04-29).

[19] Kitware, *About CMake*. [Online]. Available: https://cmake.org/overview/pdf (visited on 2020-04-11).

[20] Gradle Inc, *Gradle User Manual*. [Online]. Available: https://docs.gradle.org/current/userguide/userguide.html (visited on 2020-04-15).

[21] Gradle Inc, *Build Lifecycle*. [Online]. Available: https://docs.gradle.org/current/userguide/build_lifecycle.html (visited on 2020-04-15).

[22] Gradle Inc, *Gradle Features*. [Online]. Available: https://gradle.org/features/ (visited on 2020-04-15).

[23] Nationalencyklopedin, *Versionshantering*. [Online]. Available: http://www.ne.se.proxy.lib.chalmers.se/uppslagsverk/encyklopedi/l%C3%A5ng/versionshantering (visited on 2020-04-12).
[24] G-Truc, *OpenGL Mathematics*. [Online]. Available: https://glm.g-truc.net/0.9.9/index.html (visited on 2020-04-18).
[25] OnePlus, *Tech Specs*. [Online]. Available: https://www.oneplus.com/uk/7/specs (visited on 2020-04-29).
[26] Qualcomm Technologies, *Snapdragon Developer Tools*. [Online]. Available: https://developer.qualcomm.com/solutions/snapdragon-developer-tools (visited on 2020-04-29).
[27] Qualcomm Technologies, *Use Snapdragon Profiler to Improve Your App's Performance*. [Online]. Available: https://developer.qualcomm.com/software/snapdragon-profiler/app-notes (visited on 2020-04-29).
[28] W.-h. Shi, Y.-p. Wang, and C. Shen, *Comparison of stability between navier-stokes and euler equations*, Shanghai University, Nanjing University of Information Science & Technology, and Yantai Normal University, 2006. [Online]. Available: https://link.springer.com/article/10.1007/s10483-006-0913-y (visited on 2020-03-19).
[29] Engineers Edge, *Viscosity of Air, Dynamic and Kinematic*. [Online]. Available: https://www.engineersedge.com/physics/viscosity_of_air_dynamic_and_kinematic_14483.htm (visited on 2020-04-29).
[30] Nationalencyklopedin, *Rotation*. [Online]. Available: http://www.ne.se.proxy.lib.chalmers.se/uppslagsverk/encyklopedi/l%C3%A5ng/rotation-(av-vektorf%C3%A4lt) (visited on 2020-04-17).
[31] K. Perlin, *An image synthesizer*, New York University, 1985. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/325165.325247 (visited on 2020-04-21).
[32] L. Råde and B. Westergren, *Mathematics Handbook for Science and Engineering*. Studentliteratur, 2018.
[33] Nationalencyklopedin, *Densitet*. [Online]. Available: http://www.ne.se.proxy.lib.chalmers.se/uppslagsverk/encyklopedi/l%C3%A5ng/densitet (visited on 2020-04-18).
[34] J. Stam, "Real-time fluid dynamics for games", *Proceedings of the game developer conference*, vol. 18, 2003, p. 25.
[35] E. of Mathematics, *Well-posed problem*, 2018. [Online]. Available: URL:%20http://www.encyclopediaofmath.org/index.php?title=Well-posed_problem&oldid=42887 (visited on 2020-04-29).
[36] C. Wyman, P.-P. Sloan, and P. Shirley, Simple analytic approximations to the cie xyz color matching functions, *Journal of Computer Graphics Techniques* **2**, no. 2 2013, 1–11, 2013.
[37] M. D. Fairchild, *Color appearance models*. John Wiley & Sons, 2013.
[38] Vanessaezekowitz at en.wikipedia, *Normalized responsivity spectra of human cone cells, s, m, and l types*, 2007. [Online]. Available: https://commons.wikimedia.org/w/index.php?curid=10514373 (visited on 2020-05-11).
[39] M. Anderson, R. Motta, S. Chandrasekar, and M. Stokes, "Proposal for a standard default color space for the internet—srgb", *Color and imaging conference*, Society for Imaging Science and Technology, vol. 1996, 1996, pp. 238–245.
[40] P. Tan, "Phong reflectance model", *Computer Vision: A Reference Guide*, K. Ikeuchi, Ed. Boston, MA: Springer US, 2014, pp. 592–594, ISBN: 978-0-387-31439-6. DOI: 10.1007/978-0-387-31439-6_536. [Online]. Available: https://doi.org/10.1007/978-0-387-31439-6_536.
[41] M. Ikits, J. Kniss, A. Lefohn, and C. Hansen, *GPU Gems*, 5th ed. Addison-Wesley, 2007. [Online]. Available: https://developer.download.nvidia.com/books/HTML/gpugems/gpugems_ch39.html (visited on 2020-04-27).
[42] Android Developers, *Developer Guides*. [Online]. Available: https://developer.android.com/guide (visited on 2020-02-25).
[43] Android Developers, *Application Fundamentals*. [Online]. Available: https://developer.android.com/guide/components/fundamentals (visited on 2020-02-26).
[44] Android Developers, *Getting started with the NDK*. [Online]. Available: https://developer.android.com/ndk/guides (visited on 2020-02-25).
[45] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *Java Virtual Machine Specification, Java SE 7 Edition*. Addison-Wesley, 2013, ch. 1.
[46] S. Lee and J. W. Jeon, *Evaluating performance of Android platform using native C for embedded systems*, 2010. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/5669738 (visited on 2020-02-26).
[47] Khronos Group, *OpenGL ES Version 3.1*, 2016. [Online]. Available: https://www.khronos.org/registry/OpenGL/specs/es/3.1/es_spec_3.1.pdf (visited on 2020-03-22).

[48] Nationalencyklopedin, *GPU*. [Online]. Available: http://www.ne.se.proxy.lib.chalmers.se/uppslagsverk/encyklopedi/l%C3%A5ng/gpu (visited on 2020-04-12).
[49] Nationalencyklopedin, *Datorarkitektur*. [Online]. Available: http://www.ne.se.proxy.lib.chalmers.se/uppslagsverk/encyklopedi/l%C3%A5ng/datorarkitektur (visited on 2020-04-14).
[50] The Qt Company, *Qt Features*. [Online]. Available: https://www.qt.io/product/features (visited on 2020-05-13).

A Performance and Visuals of Difference Grid Sizes

In this Appendix, the tested resolutions in Section 6.2 and the visual results of every resolution are displayed. The approach used when simulating is the Euler approach presented in Chapter 3.

12x22x12

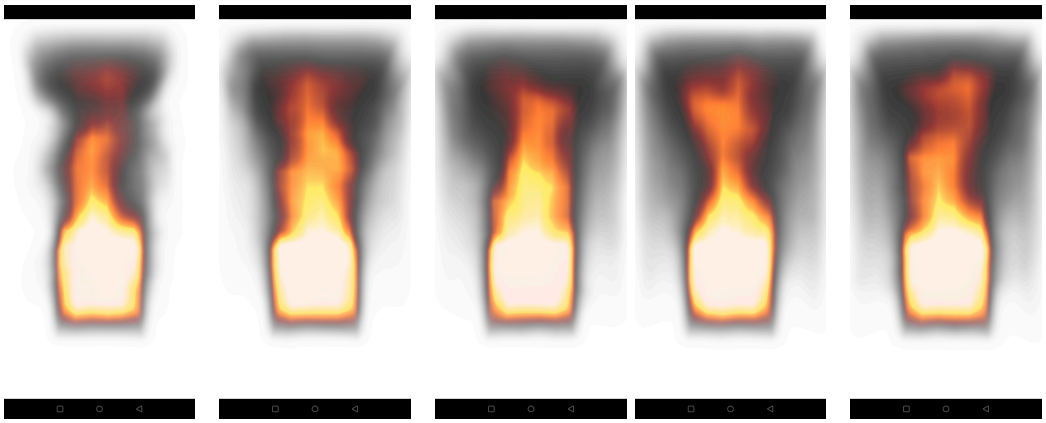

Figure A.1: *Visuals of the fire with a grid size of 12x22x12*

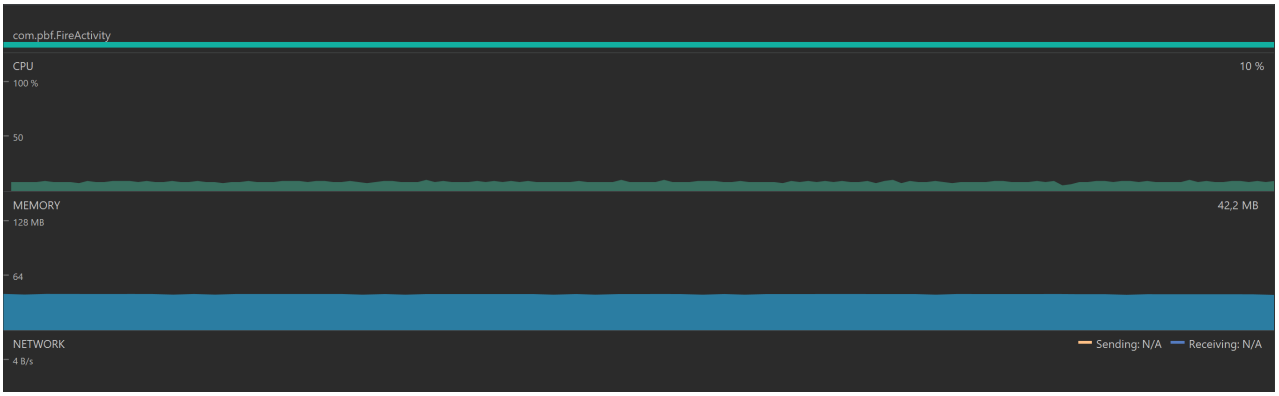

Figure A.2: *Android Profiler graph with a grid size of 12x22x12, measuring CPU utilization and memory usage in MB*

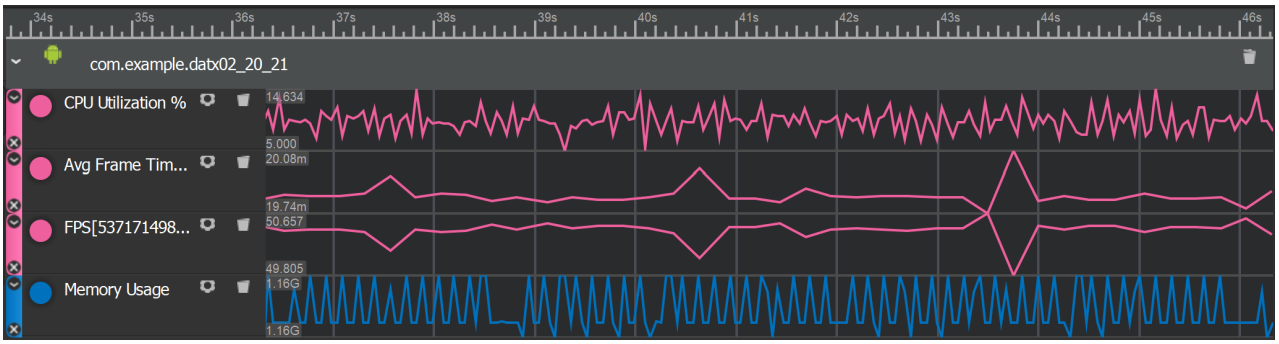

Figure A.3: *Snapdragon Profiler graph with a grid size of 12x22x12, measuring CPU utilization, time per frame, frames per second and memory usage*

32x62x32

Figure A.4: *Visuals of the fire with a grid size of 32x62x32*

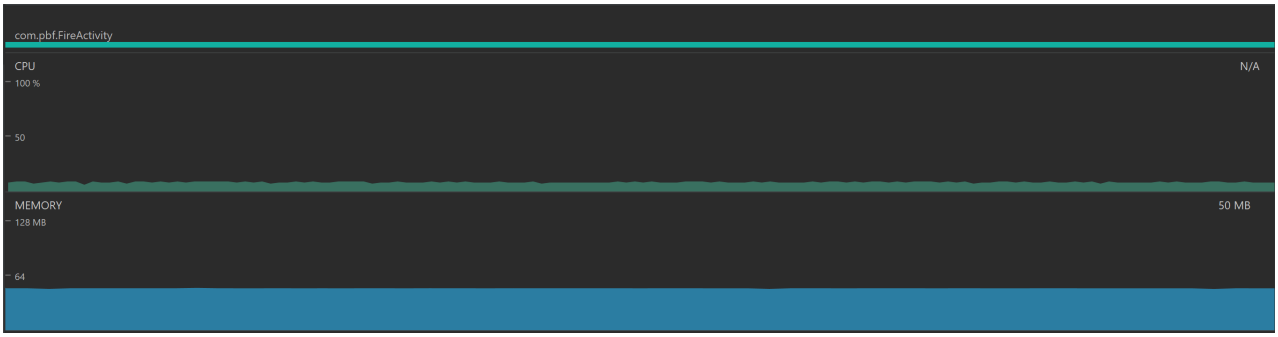

Figure A.5: *Android Profiler graph with a grid size of 32x62x32, measuring CPU utilization and memory usage in MB*

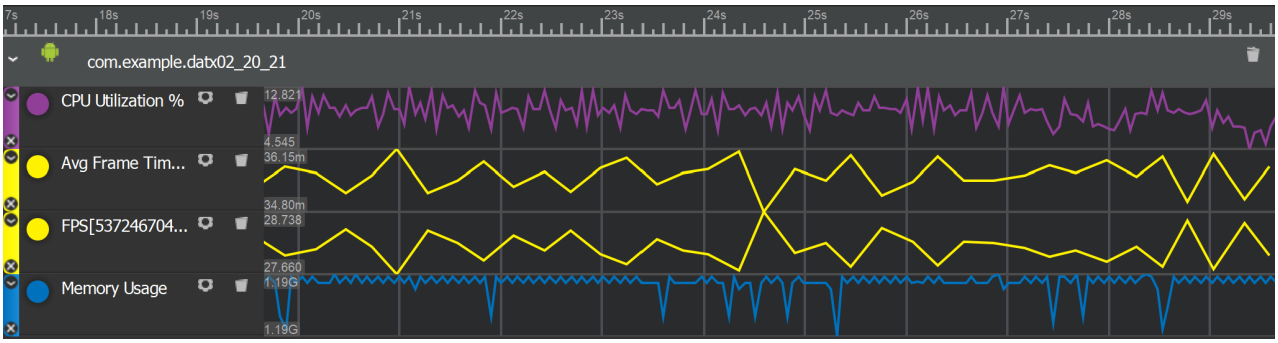

Figure A.6: *Snapdragon Profiler graph with a grid size of 32x62x32, measuring CPU utilization, time per frame, frames per second and memory usage*

52x102x52

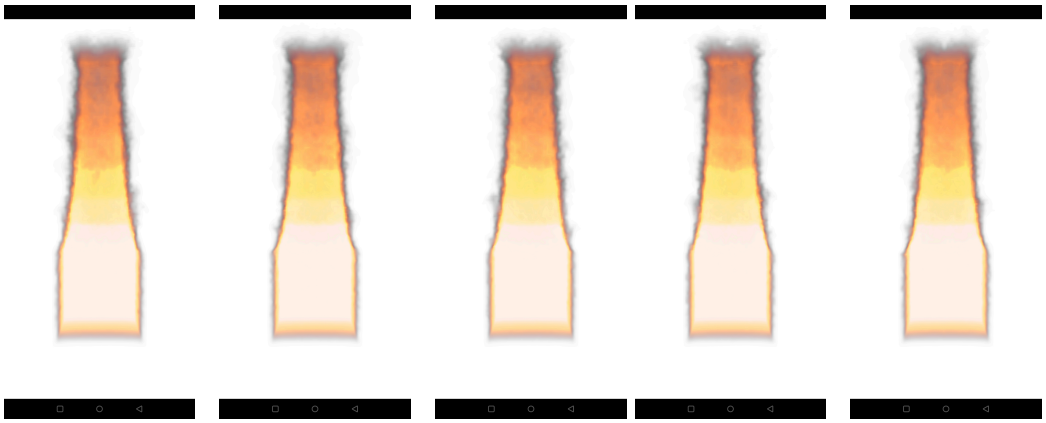

Figure A.7: *Visuals of the fire with a grid size of 52x102x52*

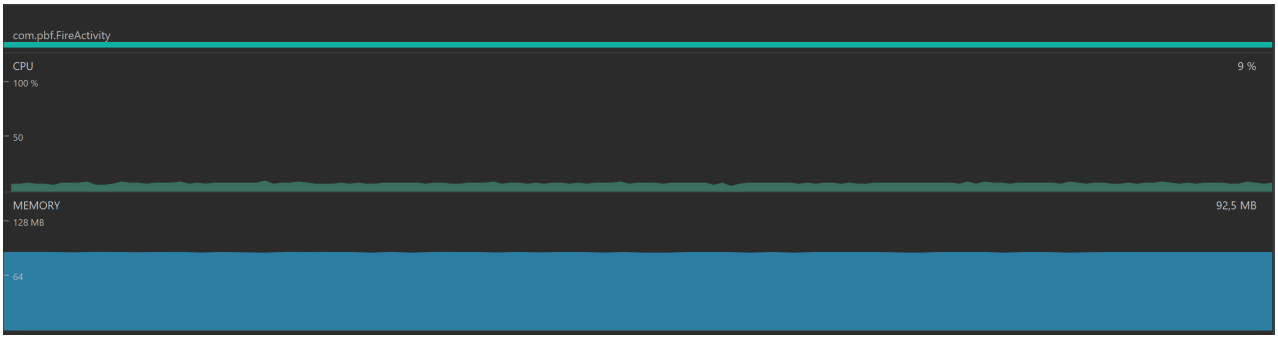

Figure A.8: *Android Profiler graph with a grid size of 52x102x52, measuring CPU utilization and memory usage in MB*

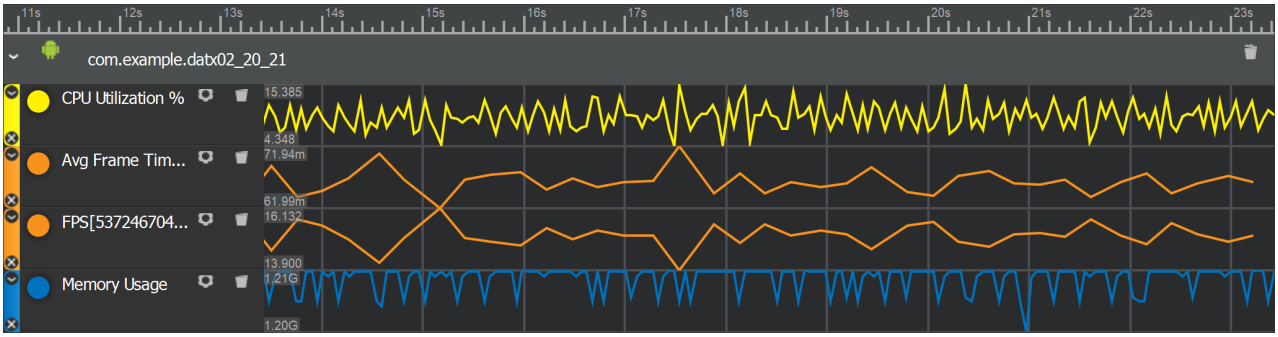

Figure A.9: *Snapdragon Profiler graph with a grid size of 52x102x52, measuring CPU utilization, time per frame, frames per second and memory usage*

72x142x72

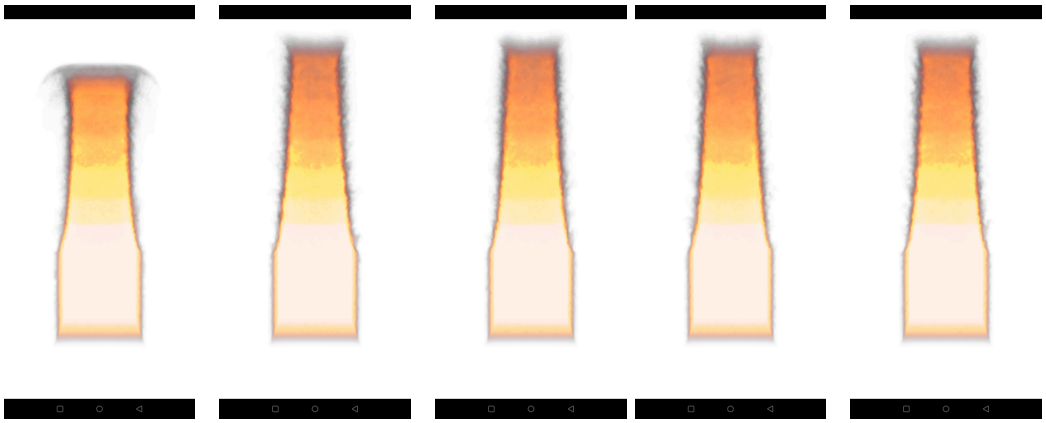

Figure A.10: *Visuals of the fire with a grid size of 72x142x72*

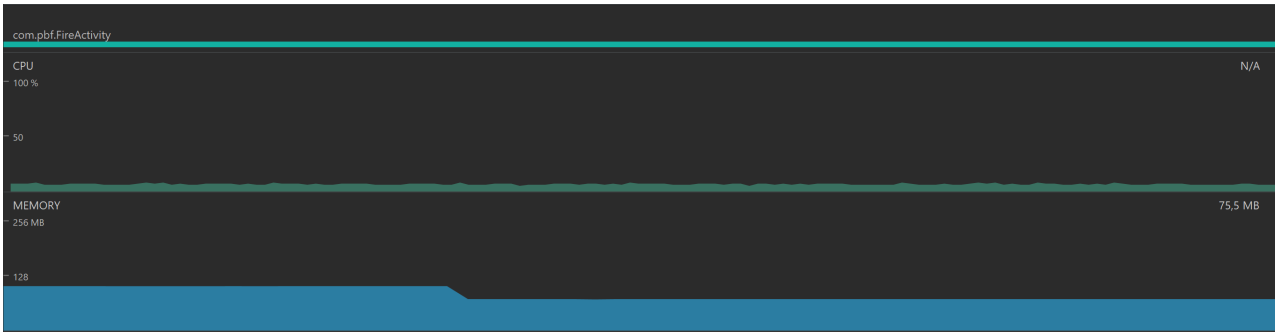

Figure A.11: *Android Profiler graph with a grid size of 72x142x72, measuring CPU utilization and memory usage in MB*

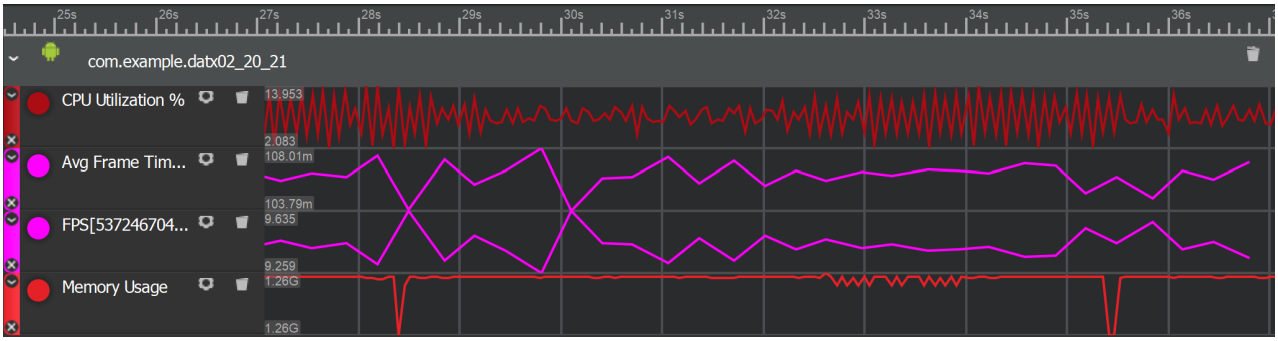

Figure A.12: *Snapdragon Profiler graph with a grid size of 72x142x72, measuring CPU utilization, time per frame, frames per second and memory usage*

12x42x12

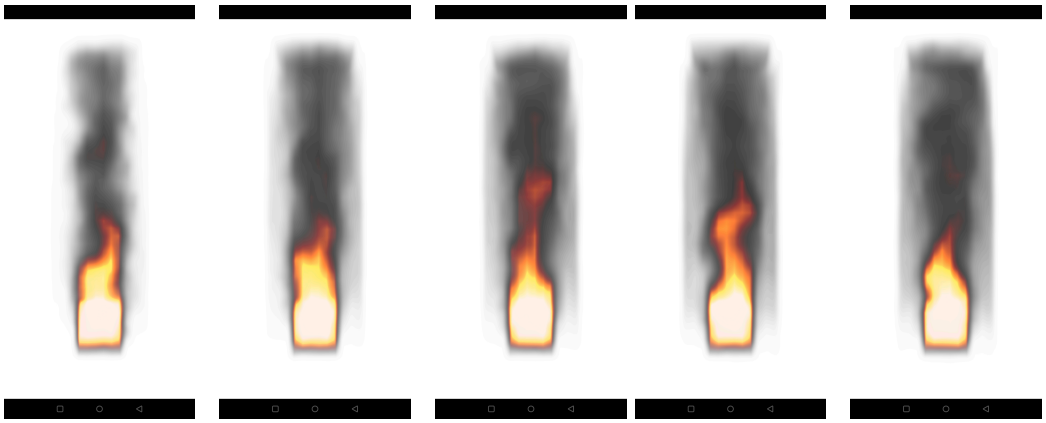

Figure A.13: *Visuals of the fire with a grid size of 12x42x12*

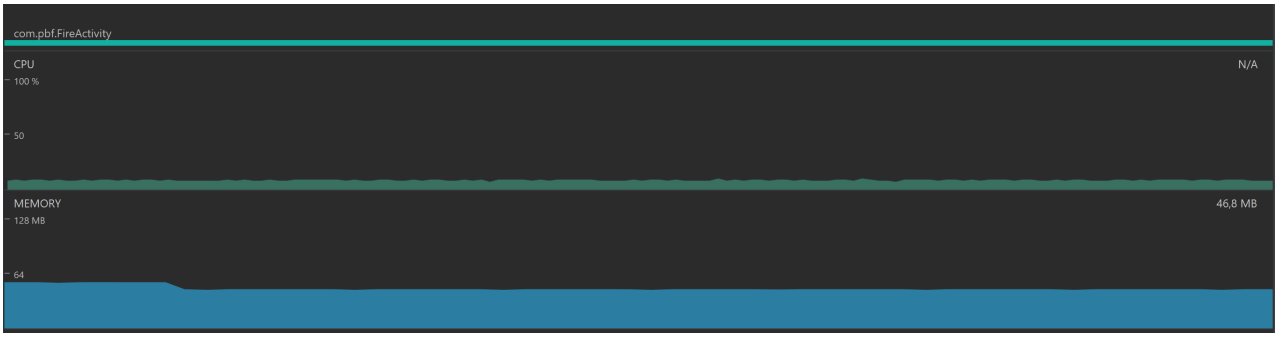

Figure A.14: *Android Profiler graph with a grid size of 12x42x12, measuring CPU utilization and memory usage in MB*

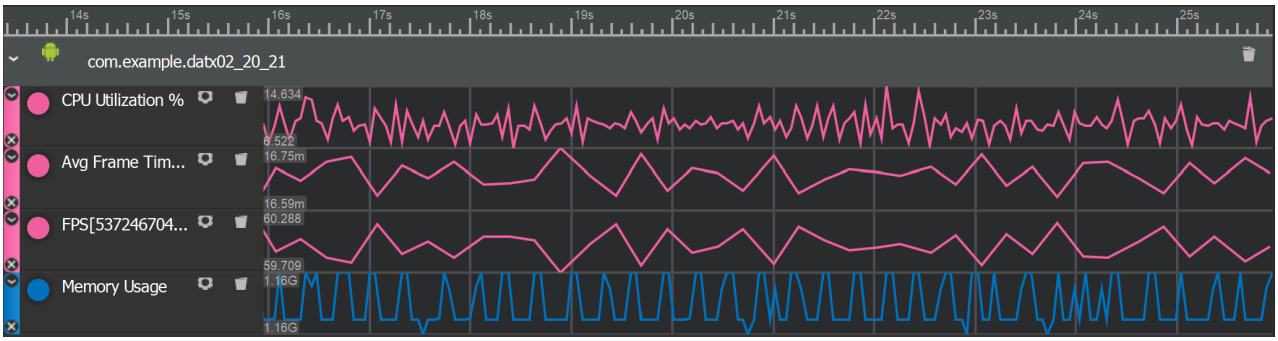

Figure A.15: *Snapdragon Profiler graph with a grid size of 12x42x12, measuring CPU utilization, time per frame, frames per second and memory usage*

32x122x32

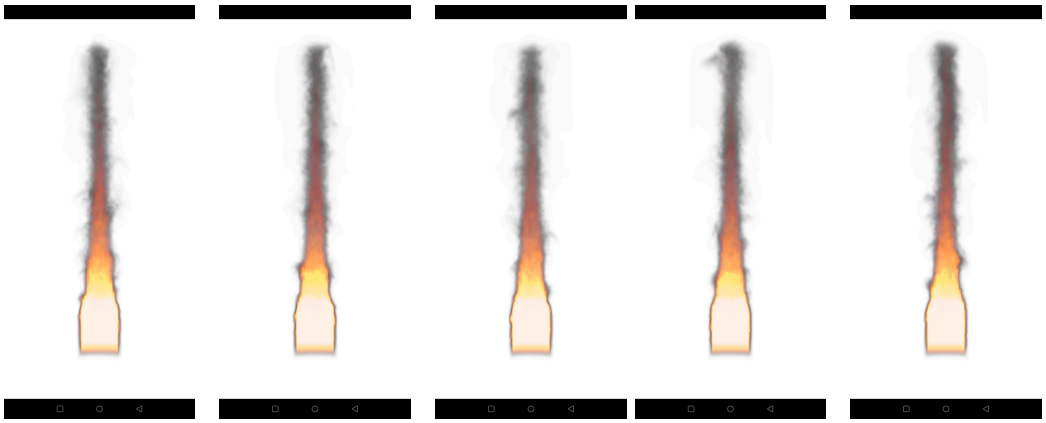

Figure A.16: *Visuals of the fire with a grid size of 32x122x32*

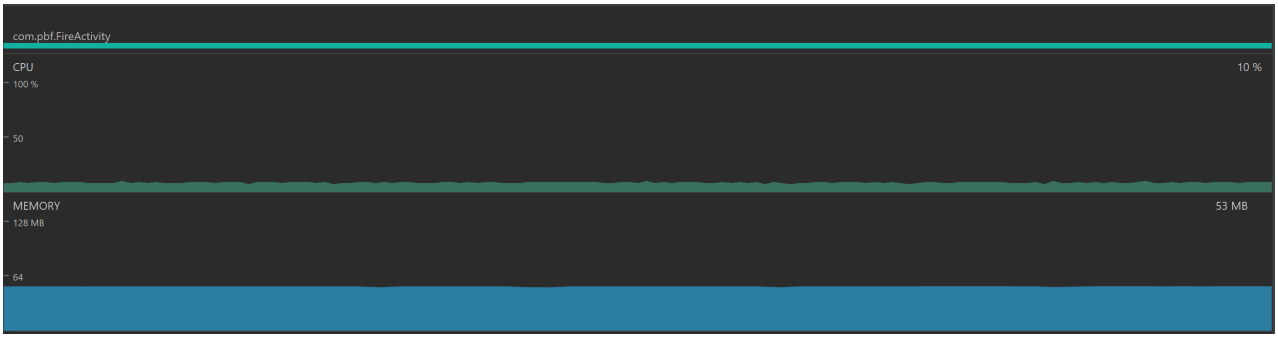

Figure A.17: *Android Profiler graph with a grid size of 32x122x32, measuring CPU utilization and memory usage in MB*

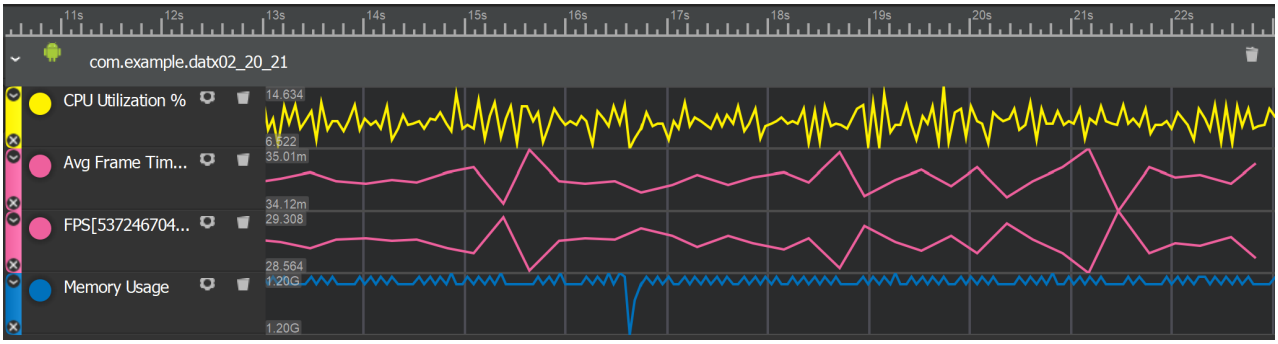

Figure A.18: *Snapdragon Profiler graph with a grid size of 32x122x32, measuring CPU utilization, time per frame, frames per second and memory usage*

52x202x52

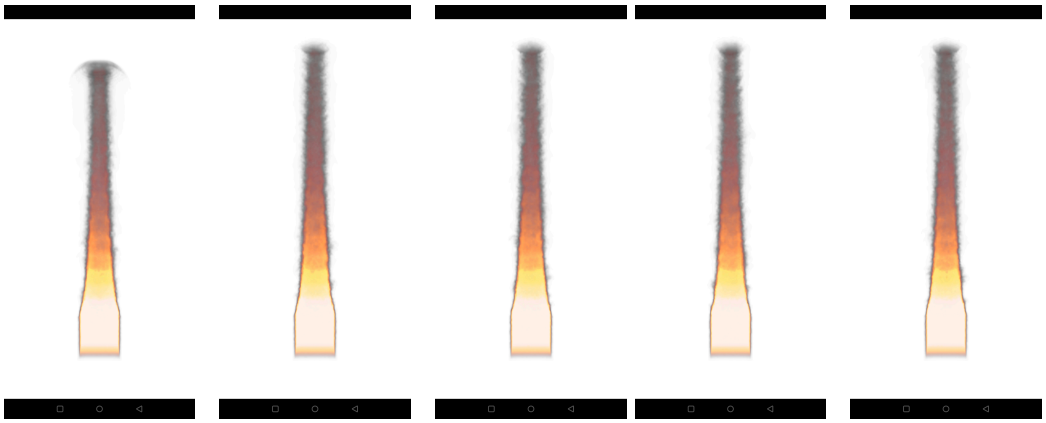

Figure A.19: *Visuals of the fire with a grid size of 52x202x52*

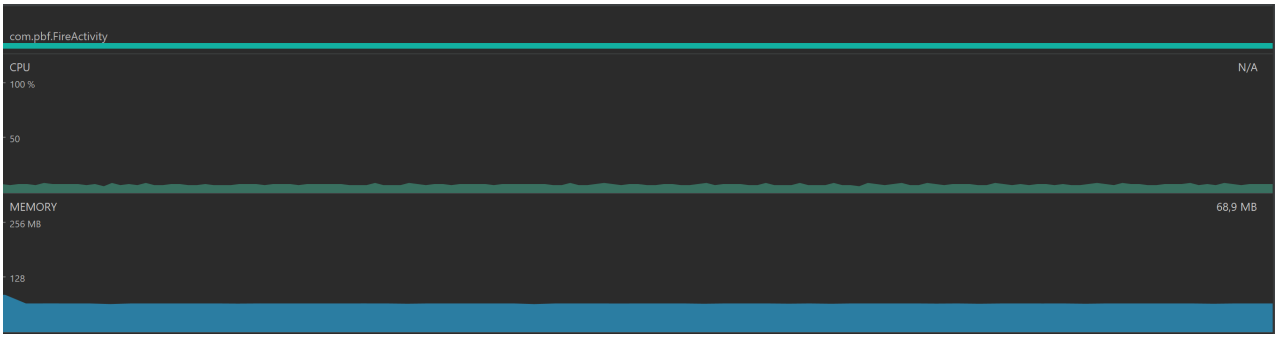

Figure A.20: *Android Profiler graph with a grid size of 52x202x52, measuring CPU utilization and memory usage in MB*

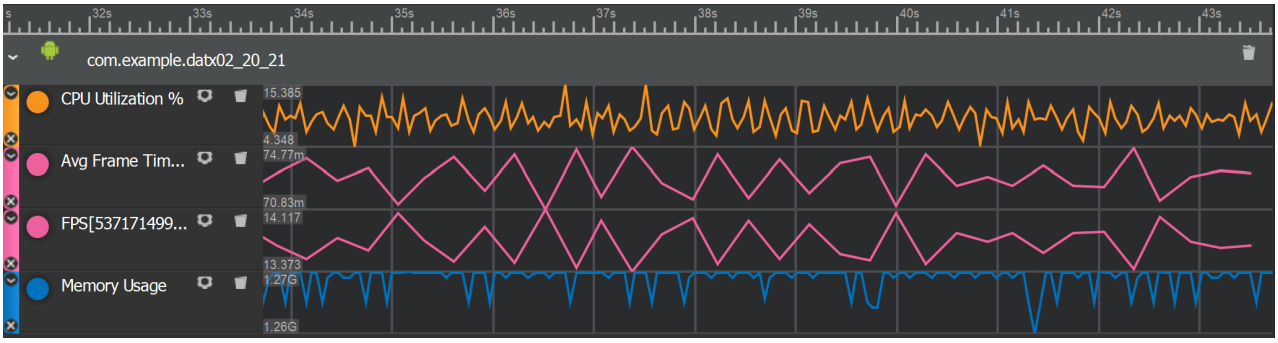

Figure A.21: *Snapdragon Profiler graph with a grid size of 52x202x52, measuring CPU utilization, time per frame, frames per second and memory usage*

72x282x72

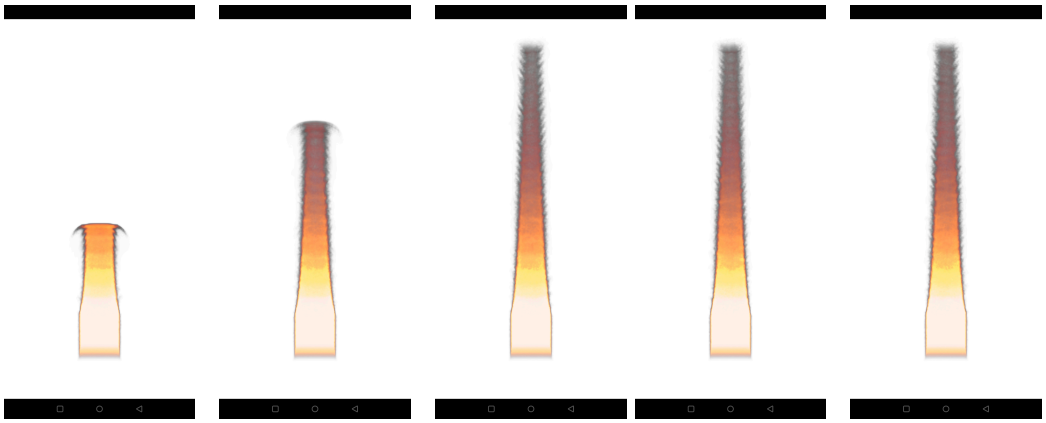

Figure A.22: *Visuals of the fire with a grid size of 72x282x72*

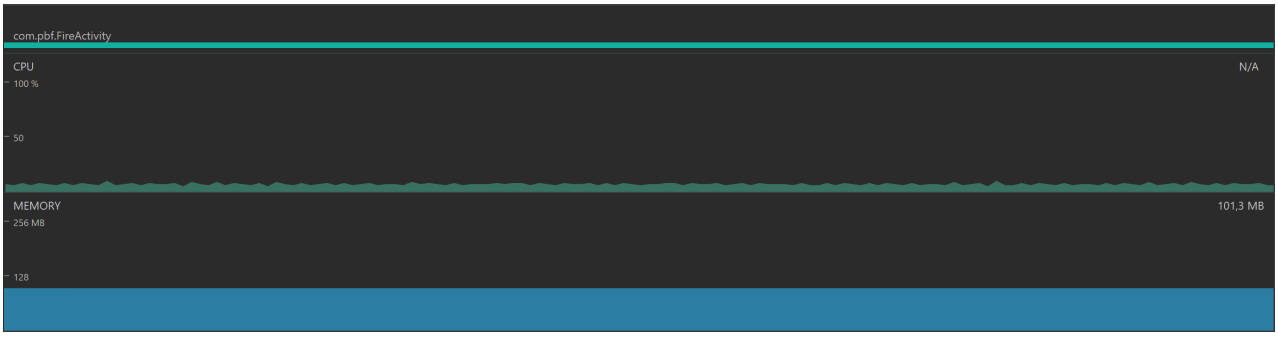

Figure A.23: *Android Profiler graph with a grid size of 72x282x72, measuring CPU utilization and memory usage in MB*

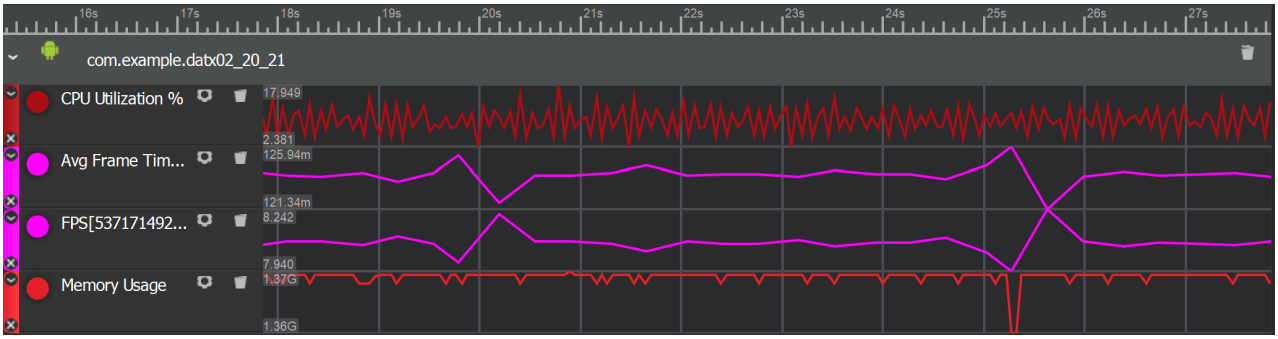

Figure A.24: *Snapdragon Profiler graph with a grid size of 72x282x72, measuring CPU utilization, time per frame, frames per second and memory usage*

B Navier-Stokes Performance and Visuals

In this Appendix, the tested resolutions of the Navier-Stokes approach in Section 6.2.2 and the visual results of every resolution are displayed. The scaling ratio used when simulating is 1x4x1.

12x42x12

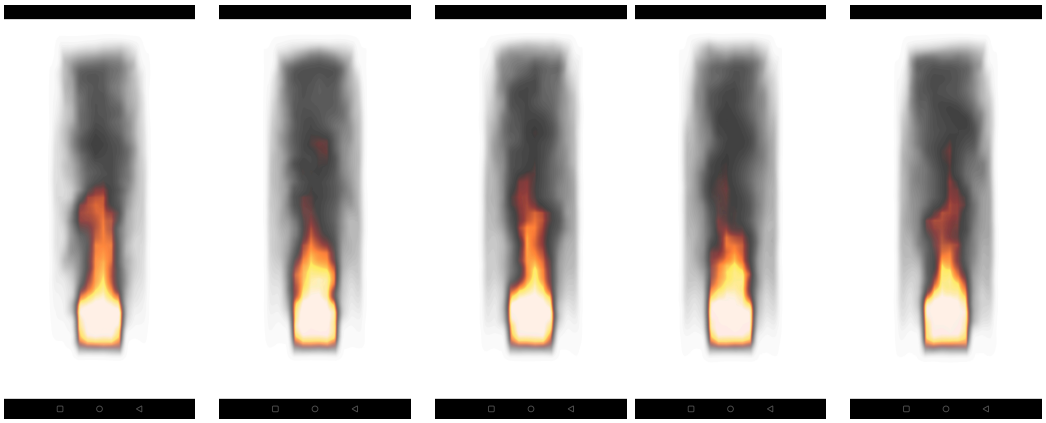

Figure B.1: *Visuals of the Navier-Stokes approach using a grid size of 12x42x12*

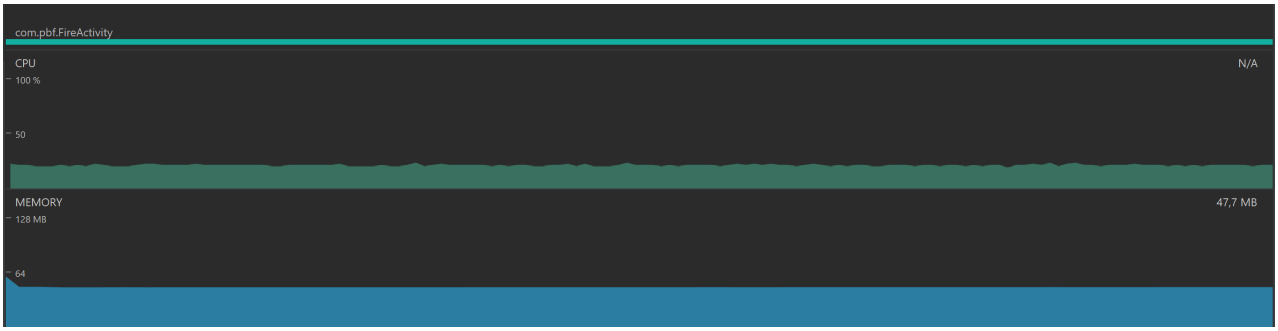

Figure B.2: *Graphs of CPU utilization and memory usage (MB) with the Navier-Stokes approach in Android Profiler*

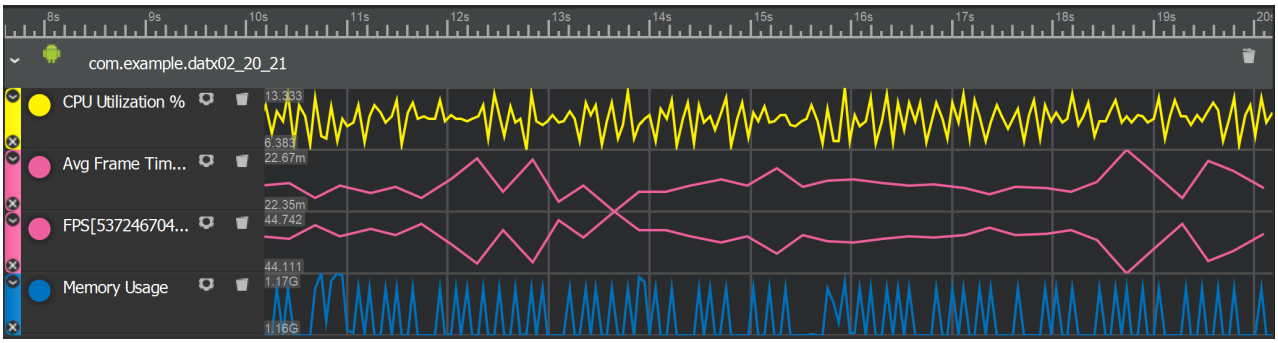

Figure B.3: *Graphs of CPU utilization, time per frame, frames per second and memory usage with the Navier-Stokes approach in Snapdragon Profiler*

32x122x32

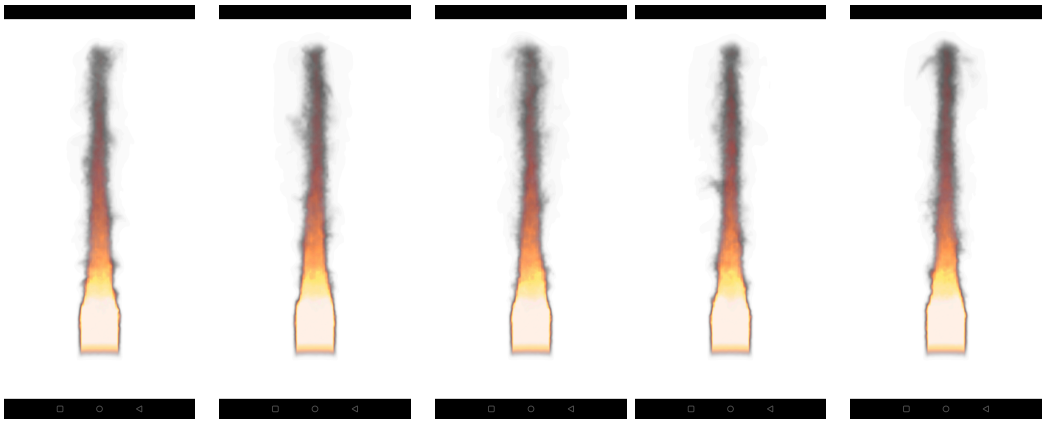

Figure B.4: *Visuals of the Navier-Stokes approach using a grid size of 32x122x32*

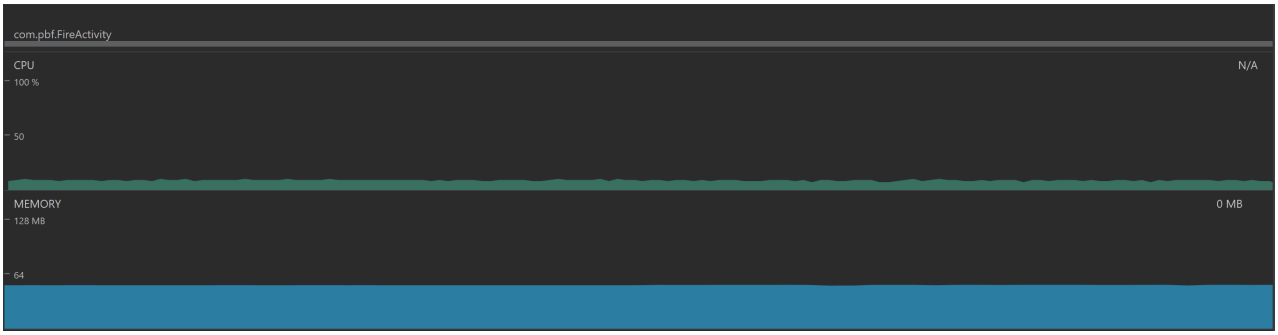

Figure B.5: *Graphs of CPU utilization and memory usage (MB) with the Navier-Stokes approach in Android Profiler*

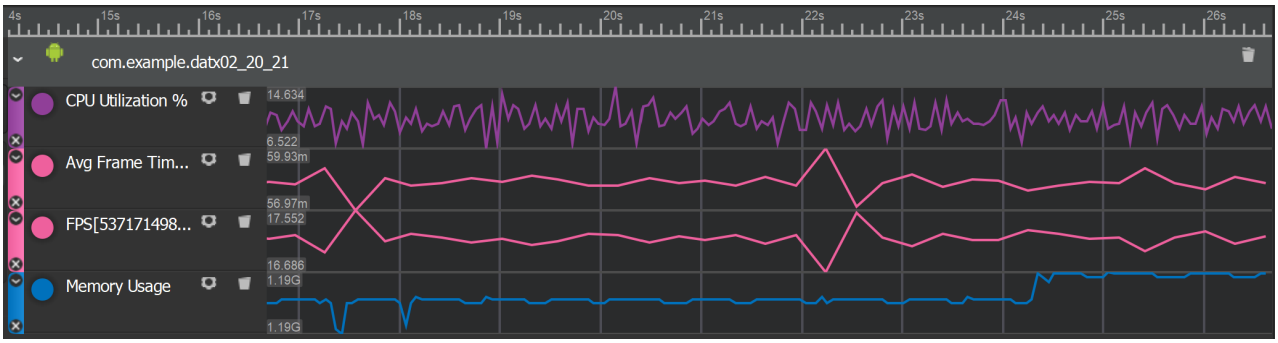

Figure B.6: *Graphs of CPU utilization, time per frame, frames per second and memory usage with the Navier-Stokes approach in Snapdragon Profiler*

52x202x52

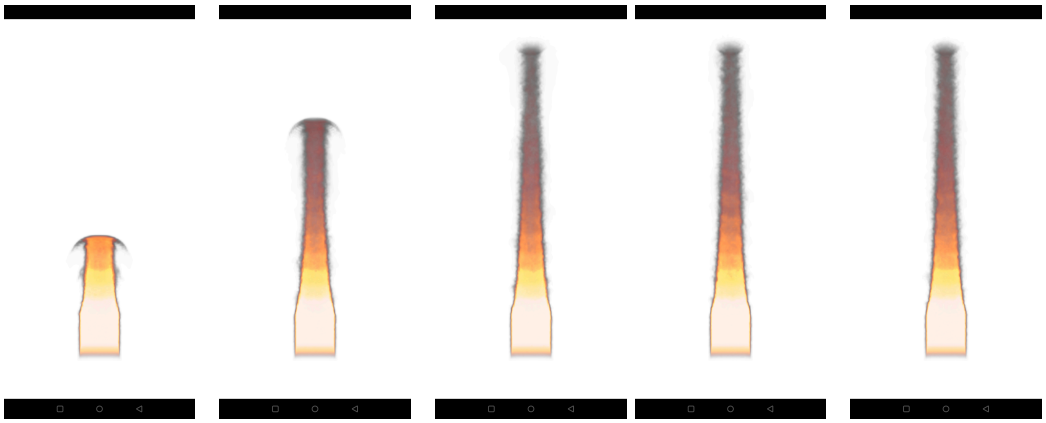

Figure B.7: *Visuals of the Navier-Stokes approach using a grid size of 52x202x52*

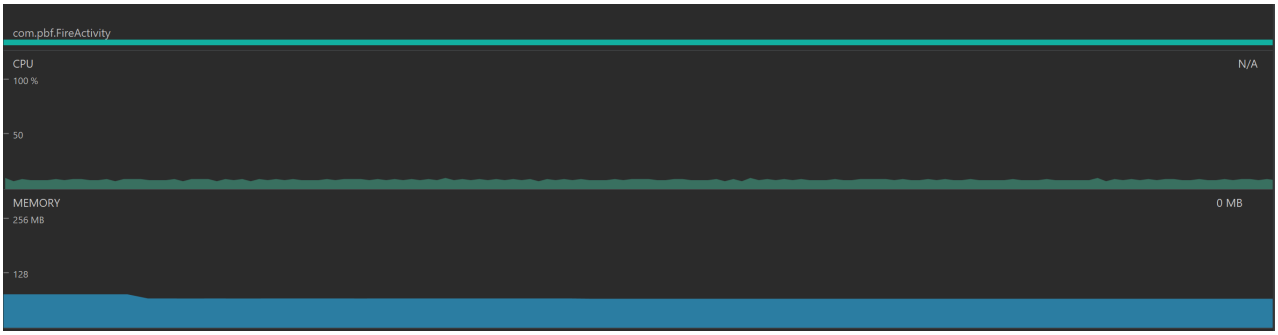

Figure B.8: *Graphs of CPU utilization and memory usage (MB) with the Navier-Stokes approach in Android Profiler*

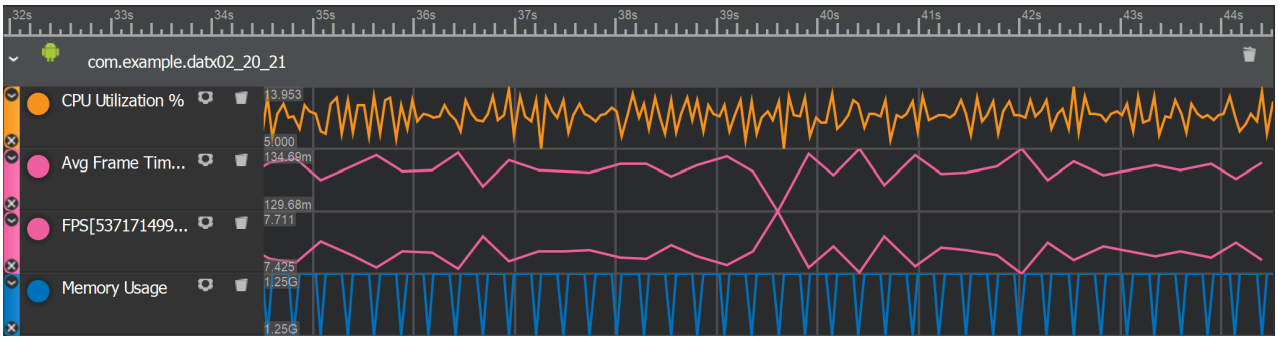

Figure B.9: *Graphs of CPU utilization, time per frame, frames per second and memory usage with the Navier-Stokes approach in Snapdragon Profiler*

72x282x72

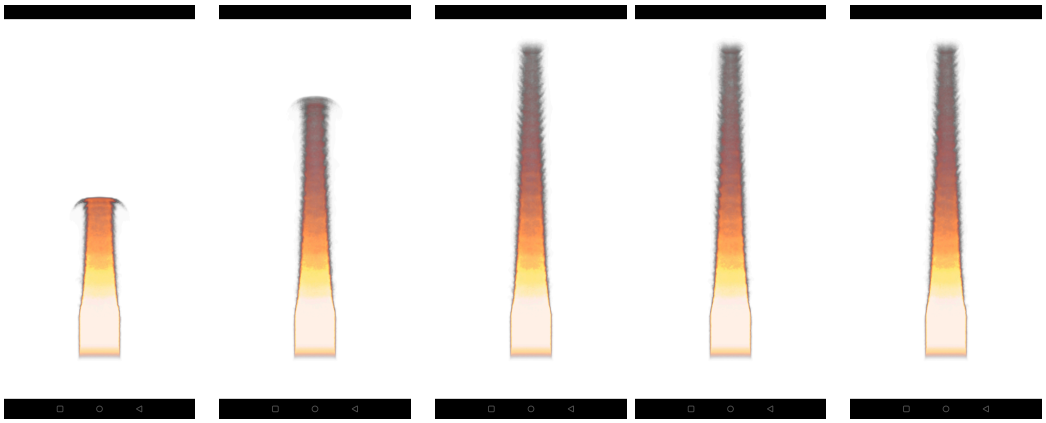

Figure B.10: *Visuals of the Navier-Stokes approach using a grid size of 72x282x72*

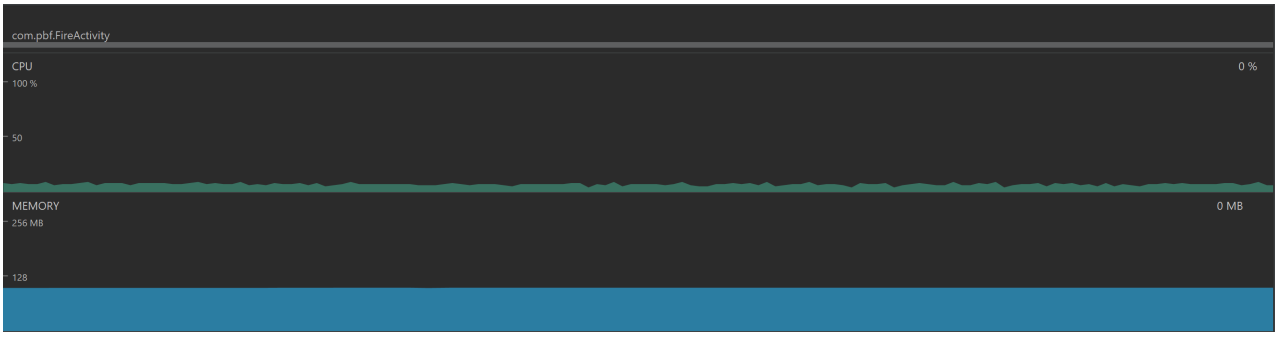

Figure B.11: *Graphs of CPU utilization and memory usage (MB) with the Navier-Stokes approach in Android Profiler*

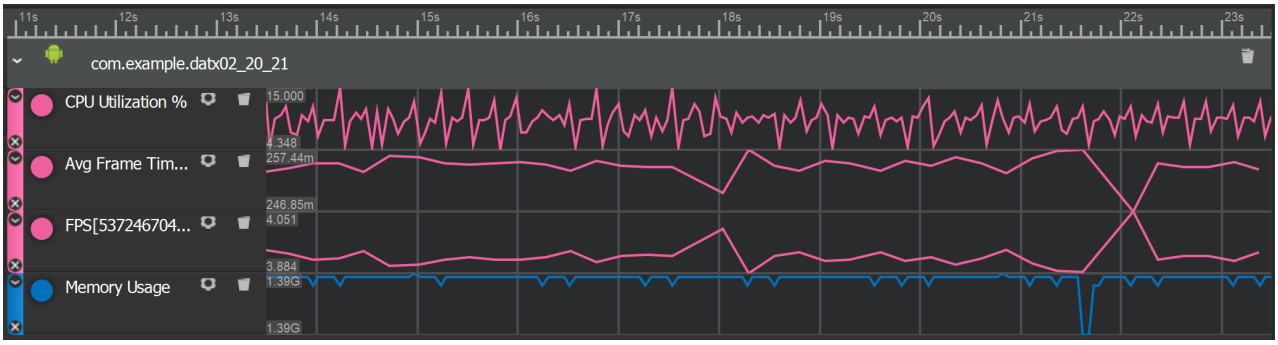

Figure B.12: *Graphs of CPU utilization, time per frame, frames per second and memory usage with the Navier-Stokes approach in Snapdragon Profiler*

C Performance of Wavelet Turbulence and the Visual Result

In this Appendix, the tested resolutions when using the Wavelet Turbulence in Section 6.2.4 is displayed, as well as the visual results of every resolution. The scaling ratio used when simulating is 1x4x1.

12x42x12

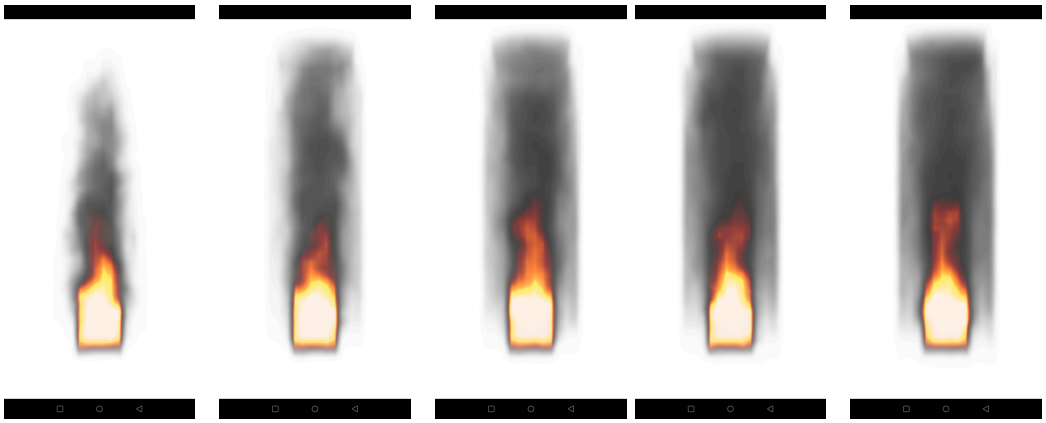

Figure C.1: *Visuals when using Wavelet Turbulence with a grid size of 12x42x12 for the velocity, and a grid size of 12x42x12 for the temperature and density*

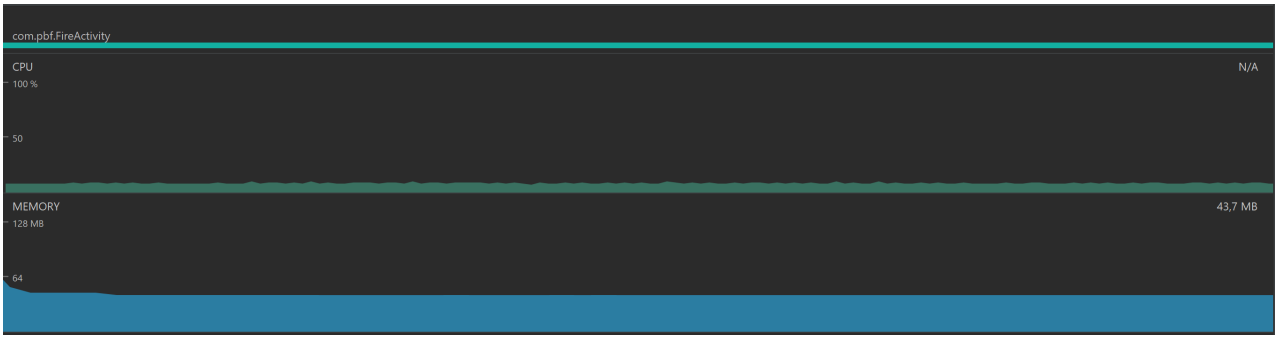

Figure C.2: *Graphs of CPU utilization and memory usage (MB) of the simulation with Wavelet Turbulence in Android Profiler*

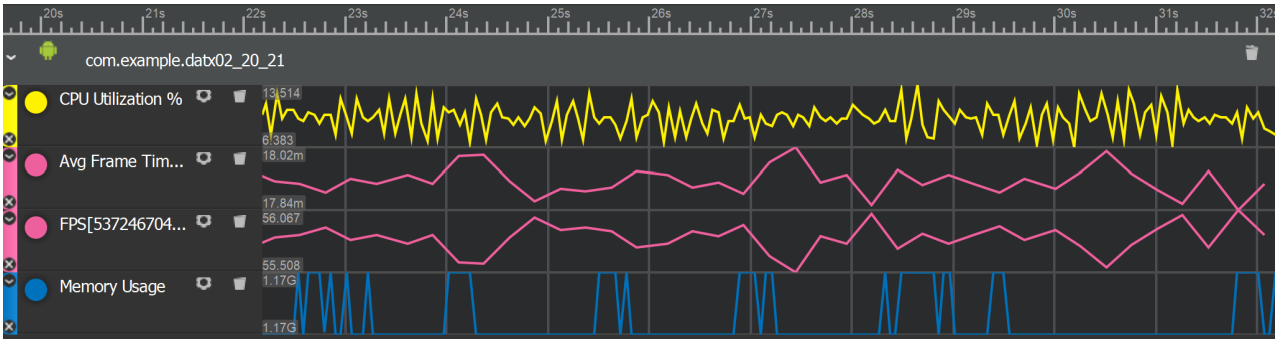

Figure C.3: *Graphs of CPU utilization, time per frame, frames per second and memory usage with the simulation with Wavelet Turbulence in Snapdragon Profiler*

32x122x32

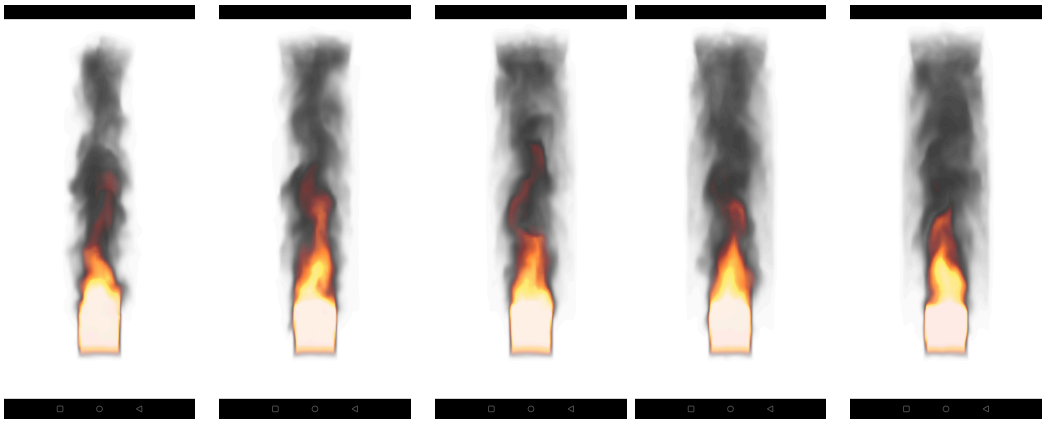

Figure C.4: *Visuals when using Wavelet Turbulence with a grid size of 12x42x12 for the velocity, and a grid size of 32x122x32 for the temperature and density*

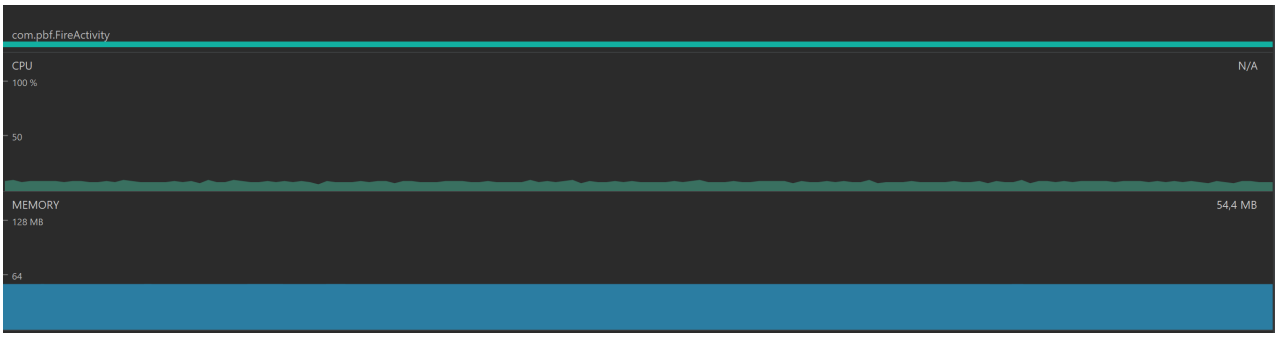

Figure C.5: *Graphs of CPU utilization and memory usage (MB) of the simulation with Wavelet Turbulence in Android Profiler*

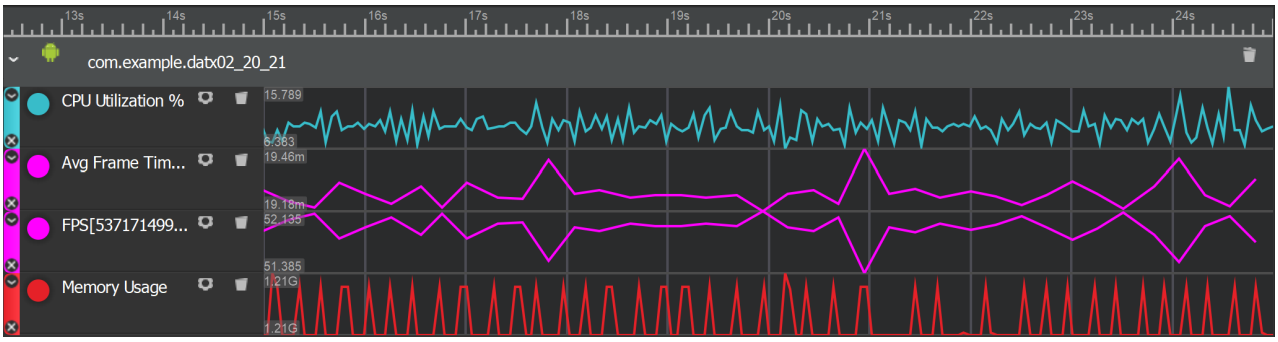

Figure C.6: *Graphs of CPU utilization, time per frame, frames per second and memory usage with the simulation with Wavelet Turbulence in Snapdragon Profiler*

52x202x52

Figure C.7: *Visuals when using Wavelet Turbulence with a grid size of 12x42x12 for the velocity, and a grid size of 52x202x52 for the temperature and density*

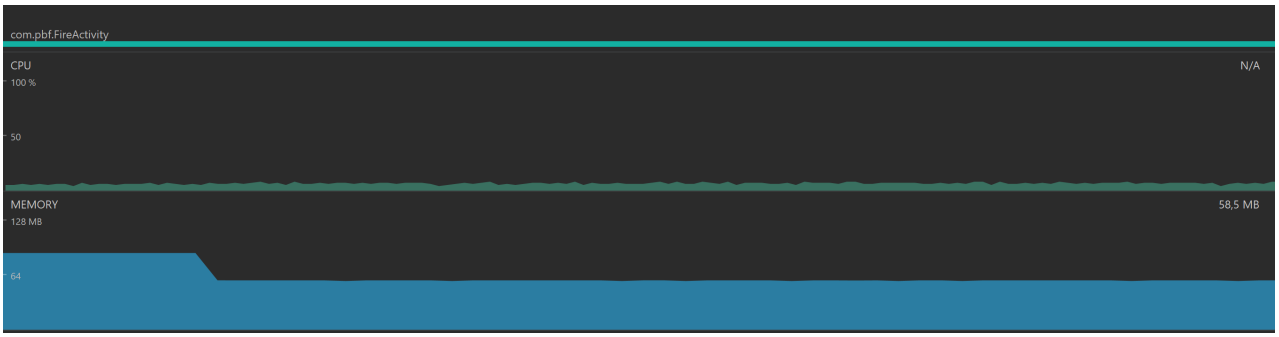

Figure C.8: *Graphs of CPU utilization and memory usage (MB) of the simulation with Wavelet Turbulence in Android Profiler*

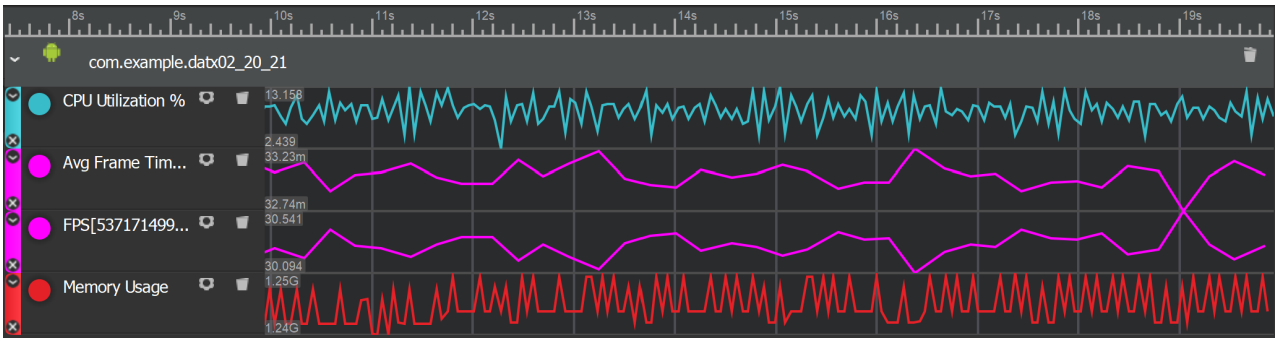

Figure C.9: *Graphs of CPU utilization, time per frame, frames per second and memory usage with the simulation with Wavelet Turbulence in Snapdragon Profiler*

72x282x72

Figure C.10: *Visuals when using Wavelet Turbulence with a grid size of 12x42x12 for the velocity, and a grid size of 72x282x72 for the temperature and density*

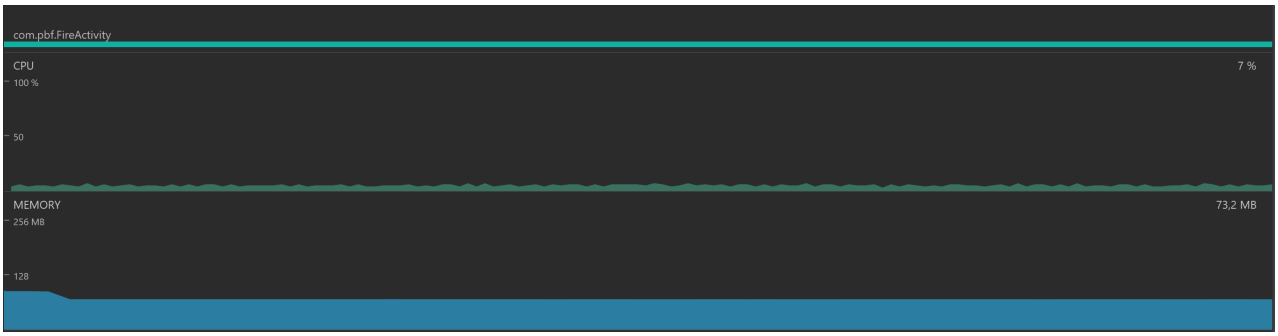

Figure C.11: *Graphs of CPU utilization and memory usage (MB) of the simulation with Wavelet Turbulence in Android Profiler*

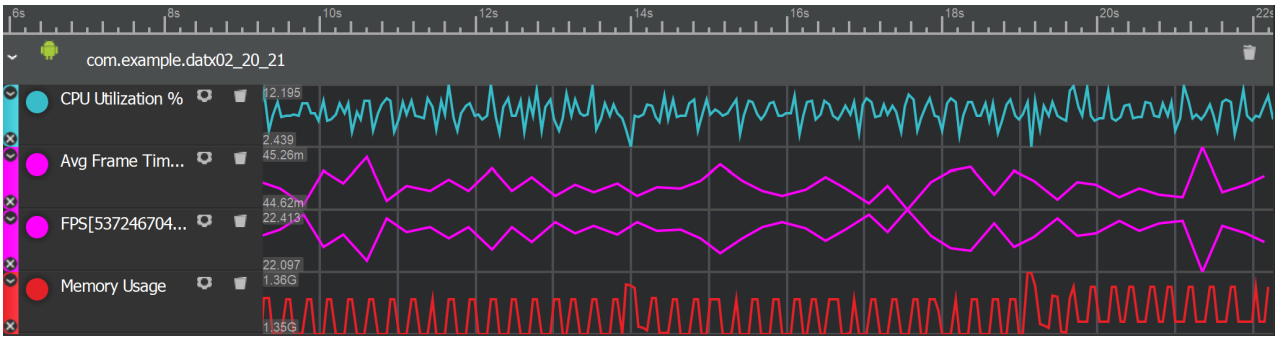

Figure C.12: *Graphs of CPU utilization, time per frame, frames per second and memory usage with the simulation with Wavelet Turbulence in Snapdragon Profiler*