university of groningen

faculty of science and engineering

Lagrangian Particle Tracking in Fluids Using Smart Devices

Bachelor Thesis

July 2024

Author:
Martin Opat

First Supervisor:
Dr. Christian Kehl

Second Supervisor:
Dr. Julian Köllermeier

Abstract

The demand for fluid simulations is growing, driven by the need for advanced solutions in climate adaptation, local water management, and economical-industrial use cases. Such applications underscore the critical role of accurate and localized fluid flow analysis in environmental and industrial scenarios.

This research aims to design and implement a fluid simulation using a compiled language to enhance efficiency and scalability. The transition to compiled languages is motivated by the need to reduce the computational overhead associated with interpreted languages, making the simulations more practical for various applications. Additionally, the emergence of powerful yet resource-constrained smart devices, alongside the proliferation of wireless devices, further expands the scope of these applications.

The subsequent software evaluation focuses on operational efficiency, examining processing speed and effectiveness under resource-limited conditions typical of smart devices with lower computational resources. This analysis demonstrates that fluid simulations involving up to half a million particles are feasible at interactive frame rates on such devices. The successful implementation underlines the potential for deploying advanced simulation tools directly on mobile platforms, providing a robust tool for local decision-making in environmental and industrial scenarios.

Acknowledgments

I would like to thank my supervisor, Dr. Christian Kehl, for his guidance and support throughout the research process and for providing the necessary resources and feedback. I would also like to thank my second supervisor Dr. Julian Köllermeier, for his insightful feedback.
The use of Grammarly was employed to proofread this document.

Data Privacy

This research involves the development and performance testing of a fluid simulation Android application. All testing and data collection were conducted locally and without the involvement of 3rd parties or 3rd party devices. Thus, no personal identifiable information or sensitive data from other individuals was collected.

The data necessary to evaluate the performance of the application is only collected by a version of the application that was tailored for testing purposes. The release version of the application does not collect any data.

The application requires access to files stored on the device. The user is prompted to manually pick the files and folders to which the application is granted read access. The application does not access files that the user has not explicitly selected. The data read from the files is used solely to run the fluid simulation and is not stored for any longer than is strictly necessary.

This research complies with relevant data protection regulations, including GDPR and the Data Privacy Policy of the University of Groningen.

For any concerns or questions regarding data privacy, please contact:
Martin Opat
m.opat@student.rug.nl
University of Groningen

Contents

1 Introduction **6**

2 Literature review **8**

3 Methods **10**
- 3.1 Data 10
- 3.2 Simulation method 10
  - 3.2.1 Runge-Kutta 4th order 10
  - 3.2.2 Particle advection 11
- 3.3 Taking measurements 12

4 Implementation **13**
- 4.1 Android Native C++ implementation 13
- 4.2 File input 13
- 4.3 Graphics 14
  - 4.3.1 Shader setup 14
  - 4.3.2 Particle rendering 14
  - 4.3.3 Vector field rendering 14
- 4.4 Simulation 15
- 4.5 Third-party libraries 16

5 Results **17**
- 5.1 Performance comparison 17
  - 5.1.1 Render time 17
  - 5.1.2 Simulation compute time 18
  - 5.1.3 File loading time 19
  - 5.1.4 Application cycle time 19
  - 5.1.5 Wall clock time 20
- 5.2 Application cycle composition 20
  - 5.2.1 Sequential method 21
  - 5.2.2 Parallel method 21
  - 5.2.3 GPU method 22
- 5.3 Memory usage 22

6 Discussion **24**
- 6.1 Performance comparison 24
- 6.2 Application cycle composition 25
- 6.3 Memory usage 25
- 6.4 Bottlenecks 26
- 6.5 Impact 27
- 6.6 Future work 28

7 Conclusion **29**

8 References **30**

Appendix A 32
Implemented visualization techniques 32

Appendix B 35
Raw measured data 35
Processed data 35
Error Analysis 35
Glossary 35
Acronyms 36

List of Figures

2.1 A simple example of a fluid simulation using the Smooth Particle Hydrodynamics algorithm [2]. Image from [12] 8
4.1 The communication between the Java-based implementation and the native C++ implementation. 13
4.2 The rendering pipeline of the application. 15
5.1 The graph on the left shows the render time for all three methods, while the graph on the right shows the same data plotted on a logarithmic scale. 17
5.2 The graph on the left shows the simulation time for all three methods, while the graph on the right shows the same data plotted on a logarithmic scale. 18
5.3 Simulation time per particle for the three methods plotted on a logarithmic scale. 18
5.4 File loading time for the three methods plotted on a logarithmic scale. 19
5.5 The graph on the left shows the application cycle time for all three methods, while the graph on the right shows the same data plotted on a logarithmic scale. 19
5.6 The graph on the left shows the wall clock time for all three methods, while the graph on the right shows the same data plotted on a logarithmic scale. The red dashed line represents the 60 fps threshold. 20
5.7 Composition of the application cycle for the sequential method. 21
5.8 Composition of the application cycle for the parallel method. . 21
5.9 Composition of the application cycle for the GPU method. . . . 22
5.10 Memory usage for the GPU method for 250 000 particles. 22
7.1 Images of the fluid simulation Android application for 250 000 particles. 29
A.1 Simple vector field visualization technique where vectors are rendered as straight lines colored based on the vector's direction. 32
A.2 3D Texture slicing visualization technique. 33
A.3 Line integral convolution visualization technique. 34

1 Introduction

In the 18th century, Joseph-Louis Lagrange developed a mathematical model to describe the motion of fluids. This approach treats the fluid as a set of discrete particles, each carrying a certain amount of mass, velocity, and possibly other relevant physical properties. Later, in the 19th century, Claude-Louis Navier and George Gabriel Stokes developed a set of equations to describe the motion of fluids. These equations are now known as the Navier-Stokes equations and are the basis for the Eulerian grid-based approach to fluid simulation [1]. The scope of application of these models was expanded by the development of computers in the 20th century. These applications include weather forecasting, nuclear reactor cooling supplies modeling, and modeling oil spills in marine environments. In the field of physics fluid simulations, many algorithms, such as Smoothed Particle Hydrodynamics [2], the Material Point Method[3], and Position-Based Fluids [4], have been developed to simulate the motion of fluids [1].

Today's technological landscape sees a widespread use of wireless, portable devices across various fields. These devices often face constraints in computational resources, impacting their ability to perform complex tasks such as fluid simulations in real-time. However, the need to produce such simulations is increasingly sought after in today's digital era. In such situations where portability and real-time data are essential, especially when precision is not vitally important, the ability to perform fluid simulations on portable devices is crucial. More information on this topic and the current state-of-the-art is provided in Section 2.

Smart device(s) typically have limited computational resources, which restrict the complexity of data structures and processes they can handle. This limitation is inherent to their design; unlike PCs, smart device(s)' system-on-chip architecture integrates essential components into one unit, preventing component upgrades. Therefore, addressing these constraints requires innovative software solutions.

A substantial portion of physics simulation software is developed using Python due to the language's accessibility and convenience. However, this choice proves inefficient for use in smart device(s) with low computational resources due to Python's high computational overhead and reduced performance [5].

This research aims to fill the gap between computational physics and computer science by developing a high-performance fluid simulation in a compiled language suitable for devices with limited resources. The main goal is to create such a fluid simulation application for Android devices. The research seeks to determine the maximum number of particles that can be accurately simulated at interactive framerates on these devices. In doing so, the research aims to challenge the current limitations and expand the capabilities of fluid simulations on smart device(s), thus impacting fields where real-time simulation on portable devices is critical.
The research aims to answer the following research questions:

1. **What is the maximum amount of traced particles that can be simulated in a Double-Gyre model on a smart device(s), while maintaining 60 frames per second?**
2. **What is the long-term operational efficiency of the Double-Gyre model simulation on smart device(s)?**

Upon fully answering the research questions, the research aims to make the following contributions to the field of fluid simulations:

1. **A new perspective on the performance of Euler-Lagrange fluid simulations on smart device(s).**
2. **A simulation and visualization software of fluid-suspended tracer particles deployable on Android platforms.**
3. **A conclusion on the computational feasibility, as well as the limits and constraints of such simulations on smart device(s).**

2 Literature review

As the demand for real-time fluid simulations grows, so does the need to optimize computational resources, especially in smart device(s). This section will discuss the current practices and challenges in implementing fluid simulations on such devices, discussing the transition from Python-based to more efficient compile language-based implementations.

Smart device(s) typically have lower computational power than modern PCs, which feature higher memory capacities and faster processors. Their system-on-chip architecture integrates essential components like the CPU and memory, limiting upgrade possibilities and affecting performance [6]. Despite these hardware limitations, there is an increasing demand to run complex simulations, such as Lagrangian and Eulerian models, on such devices for applications in emergency services, augmented reality, or hydroelectric power plants. However, many existing simulation models are not optimized for mobile devices. This lack of optimization is primarily due to using typical numerical programming languages like FORTRAN, MatLab, or Python. These languages are designed with principles that assume the availability of substantial memory and compute resources, which often conflict with the constraints of mobile platforms [5].

Attempts have been made to optimize the Python implementations, including Just-In-Time (JIT) compilation into C [7][8] and vectorization [9]. While these methods help significantly to improve performance, they are still less efficient than if the models were developed in compiled languages such as C or C++. With the optimizations implemented, the reduced efficiency can also be seen in the memory usage, which is still handled by the Python runtime environment [10].

Regardless of whether a simulation is developed in Python or C++, the underlying algorithms remain the same. Common algorithms used for fluid simulations are Smooth Particle Hydrodynamics (SPH), the Material Point Method (MPM), and Position Based Fluids (PBF) [11]. All three of these models are based on the Lagrangian approach to fluid simulation. Figure 2.1 shows a simple example of such a simulation.

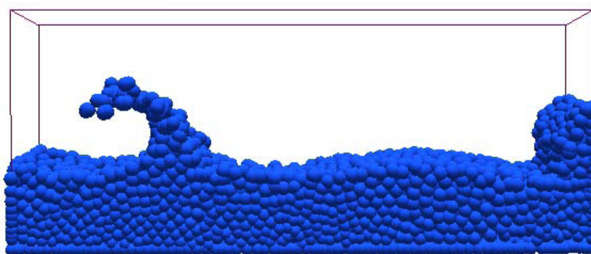


Figure 2.1: *A simple example of a fluid simulation using the Smooth Particle Hydrodynamics algorithm [2]. Image from [12]*

In SPH, the fluid is modeled as a set of particles, each representing a fluid volume with associated properties such as mass, position, and velocity. These particles interact based on their relative distances, employing a smoothing kernel function to approximate physical quantities and their gradients [2]. This function ensures that the influence of each particle decreases smoothly with distance, providing a mesh-free method to solve the fluid dynamics equations [13]. SPH is particularly effective in simulating complex fluid behaviors such as splashing and swirling motions.

The MPM method integrates both Lagrangian and Eulerian frameworks, facilitating simulations that involve large deformations and interactions between multiple phases of matter. Particles represent material points carrying mass and velocity, while a background Eulerian grid handles the computation of gradients and other properties. This dual approach leverages the advantages of particle dynamics for tracking material continuity and an Eulerian grid for numerical stability and efficient handling of large deformations [3].

In PBF, the interaction mechanism among particles is similar to SPH but emphasizes maintaining the correct density to enforce fluid incompressibility. Instead of integrating forces, PBF adjusts particle positions directly based on density constraints. This approach ensures that the simulated fluid maintains its incompressibility, making PBF particularly effective in real-time applications where rapid computations and robust handling of complex fluid interactions are crucial [4].

Another method, the Particle-In-Cell (PIC) method, combines the Lagrangian and Eulerian approaches by simulating a set of particles that interact with a spatially and temporally varying vector field defined over a fixed Eulerian grid. The particles are treated as entities possessing physical properties such as mass or electric charge. The particle's properties influence the vector field based on a specific weighting scheme, which helps determine the field's characteristics at each point. The PIC method is particularly effective in simulating the behavior of plasmas, where the interactions between particles and fields are essential to understanding the plasma's dynamics [14]. Despite its primary application in plasma simulations, the PIC method's foundational principles equally apply to fluid dynamics. This adaptability allows it to model various field values such as electric currents, density distributions, or fluid velocities using the Eulerian grid while tracing particles interacting with the grid in a Lagrangian frame.

The development and deployment of the above simulation methods on mobile devices represent a significant shift in computational science. Traditionally, the implementation of these simulations for mobile platforms has already been developed in low-level programming languages like C or C++ [15]. And yet, many of these implementation approaches offload heavy computations to remote High-Performance Computing platforms, utilizing the mobile device merely as a frontend [16]. Contrary to this common practice, the primary research goal of this thesis is to explore and develop methods that enable these complex simulations to be conducted entirely on smart device(s).

The discussed simulation methods each provide unique advantages and need to be chosen based on the specific requirements of the simulation task, such as the need for accuracy, computational efficiency, or the ability to handle particular fluid or material behavior. Furthermore, each method handles the physics of fluid and material interactions differently, illustrating that these are distinct approaches and not interchangeable. These distinctions make each method suitable for particular types of simulations, depending on the goals and constraints of the project.

The domain of Lagrangian fluid simulations on smart device(s) using native languages is underexplored and thus underdeveloped. However, existing open-source projects such as Ocean Parcels [17] provide a reference or a starting point for own approaches.

3 Methods

3.1 Data

The specific physics use case simulated in this research is the Double-Gyre model [18]. The Double-Gyre model is a well-known test case in computational fluid dynamics consisting of two counter-rotating gyres in a rectangular basin. It is commonly used to evaluate the performance of fluid simulations. The model's hydrodynamic velocity field is defined as an Eulerian grid. The grid is precomputed and stored in a NetCDF [19] dataset format. The dataset is loaded from files and used as input for the Lagrangian particle tracking simulation.

The dataset comprises a 3D hydrodynamic velocity field derived from the analytical solution of the Double-Gyre model [20], covering a period of one year. It is split into daily files, each file encapsulating the velocity field along one of the axes (x, y, or z). The spatial resolution of the dataset is specified at $539 \times 269 \times 28$ grid vertices in the x, y, and z directions, respectively, with uniform spacing between the grid points. Each data point is stored as a 32-bit floating-point number in SI units.

3.2 Simulation method

The simulation relies on a Lagrangian-Eulerian approach. The fluid flow is represented by tracer particles that move according to the velocity field defined on an Eulerian grid. The velocity field is interpolated from the Eulerian grid to the particle positions using trilinear interpolation. The particles advect through the fluid using the velocity field, and their positions are updated at each time step.

The numerical method employed in this research is the Runge-Kutta 4th order (RK4) method. RK4 is an iterative numerical method that offers higher accuracy in solving differential equations compared to more straightforward, first-order methods like Euler's method. The choice of RK4 is driven by its superior precision and robustness in maintaining stability under various simulation conditions. This method is particularly favored in computational fluid dynamics for its effectiveness in rapidly changing dynamic systems [21].

3.2.1 Runge-Kutta 4th order

The RK4 method addresses the initial value problem defined by:

$$\frac{dy}{dt} = f(t, y), \quad y(t_0) = y_0 \tag{3.1}$$

where y is the unknown function to be approximated, f is a known function, t is the independent variable, and y_0 is the initial value of y at t_0. The following equations define

the Runge-Kutta 4th order method [21]:

$$k_1 = hf(t_n, y_n) \tag{3.2}$$

$$k_2 = hf(t_n + \frac{h}{2}, y_n + \frac{k_1}{2}) \tag{3.3}$$

$$k_3 = hf(t_n + \frac{h}{2}, y_n + \frac{k_2}{2}) \tag{3.4}$$

$$k_4 = hf(t_n + h, y_n + k_3) \tag{3.5}$$

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{3.6}$$

$$t_{n+1} = t_n + h \tag{3.7}$$

where h is the step size, and y_n, and t_n are the approximations of y and t at the n-th time step, respectively. As we can see, in equations (3.2)-(3.7), this method aggregates the increments from four intermediary steps, which significantly enhances the accuracy of the solution.

3.2.2 Particle advection

Particle advection dictates the physics behind the movement of particles in a fluid. The following equation gives the advection operator in Cartesian coordinates [22]:

$$\mathbf{u} \cdot \nabla = u_x \frac{\partial}{\partial x} + u_y \frac{\partial}{\partial y} + u_z \frac{\partial}{\partial z} \tag{3.8}$$

where $\mathbf{u} = (u_x,\ u_y,\ u_z)$ is the velocity vector field. For a conserved quantity, we can define a new operator $\mathcal{A}$ as follows:

$$\mathcal{A} = \frac{\partial}{\partial t} + \mathbf{u} \cdot \nabla \tag{3.9}$$

We define the particles as simple points in space, i.e., tracer particles. With this assumption, we can represent such a tracer particle using the Dirac delta function $\delta^3(\mathbf{r}-\mathbf{r_p})$ where $\mathbf{r_p}$ is the position of the particle. Applying the operator $\mathcal{A}$ to the function $\delta^3(\mathbf{r}-\mathbf{r_p})$, results in the following advection equation:

$$\mathcal{A}\delta^3(\mathbf{r}-\mathbf{r_p}) = 0 \tag{3.10}$$

$$\tag{3.11}$$

or equivalently:

$$\frac{\partial \delta^3(\mathbf{r}-\mathbf{r_p})}{\partial t} + \mathbf{u} \cdot \nabla \delta^3(\mathbf{r}-\mathbf{r_p}) = 0 \tag{3.12}$$

To solve equation (3.12) numerically, we need to transform it into an initial value problem. We rearrange equation (3.12) as follows:

$$\frac{\partial \delta^3(\mathbf{r}-\mathbf{r_p})}{\partial t} = -\mathbf{u} \cdot \nabla \delta^3(\mathbf{r}-\mathbf{r_p}) \tag{3.13}$$

applying the product rule to the LHS gives:

$$\nabla \delta^3(\mathbf{r}-\mathbf{r_p}) \cdot \frac{d}{dt}(\mathbf{r}-\mathbf{r_p}) = -\mathbf{u} \cdot \nabla \delta^3(\mathbf{r}-\mathbf{r_p}) \tag{3.14}$$

$$\nabla \delta^3(\mathbf{r}-\mathbf{r_p}) \cdot \frac{d\mathbf{r_p}}{dt} = \mathbf{u} \cdot \nabla \delta^3(\mathbf{r}-\mathbf{r_p}) \tag{3.15}$$

$$\tag{3.16}$$

Integrating both sides over all positions $\mathbf{r}$ and all volume yields:

$$\int_{(V)} \int \nabla \delta^3(\mathbf{r} - \mathbf{r_p}) \cdot \frac{d\mathbf{r_p}}{dt} \, d\mathbf{p} \, dV = \int_{(V)} \int \mathbf{u} \cdot \nabla \delta^3(\mathbf{r} - \mathbf{r_p}) \, d\mathbf{p} \, dV \tag{3.17}$$

$$\int_{(V)} \delta^3(\mathbf{r} - \mathbf{r_p}) \cdot \frac{d\mathbf{r_p}}{dt} \, dV = \int_{(V)} \mathbf{u} \cdot \delta^3(\mathbf{r} - \mathbf{r_p}) \, dV \tag{3.18}$$

$$\frac{d\mathbf{r_p}}{dt} = \mathbf{u} \tag{3.19}$$

For some given initial condition $\mathbf{r_p}(t_0) = \mathbf{r_{p_0}}$ at time t_0, we can use the RK4 method to solve equation (3.19) numerically.

3.3 Taking measurements

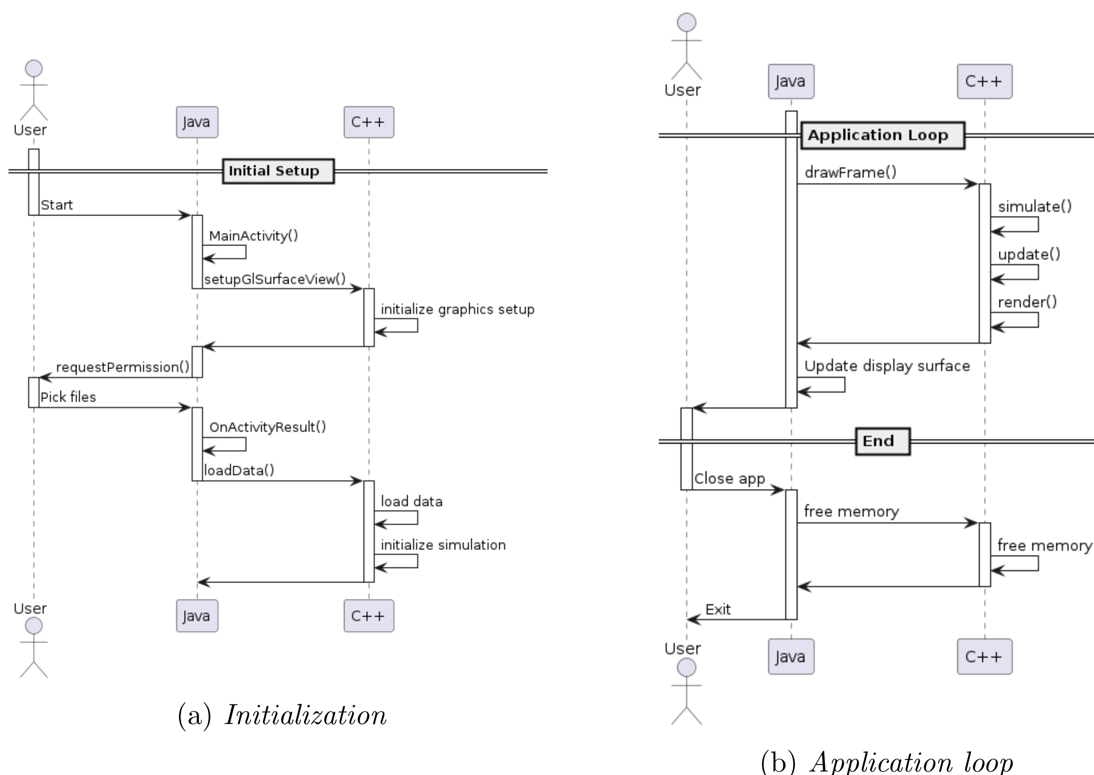
A series of time measurements across various operations are used to evaluate the performance of the Android application. These include frame rendering time (render time), particle advection simulation time (compute time), data set file loading time (load time), and wall clock time. We use OpenGL's built-in timer queries for render time to leverage the GPU's internal clock, whereas the `<time.h>` C library is utilized for compute and load times to monitor total CPU time. Wall clock time is obtained using the `<chrono>` C++ library, which measures the time spent in the application as measured by a stationary clock in the same inertial reference frame as the application.

All measurements are conducted on a Samsung Galaxy S23 Ultra, equipped with a Snapdragon 8 Gen 2 CPU and an Adreno 740 GPU, running Android version 14. The screen resolution of the device is set to 2 316px × 1 080px. The software is tested on the device under various computational loads by adjusting the particle counts at multiple levels: 1 500, 3 500, 7 500, 17 000, 37 000, 85 000, 190 000, 420 000, 950 000, 2 200 000, and 5 000 000 particles.
For each particle count, time measurements span a 5-minute period, with each type of time being sampled and averaged over all cycles within each second. This set of measurements is taken for three different simulation method implementations: sequential particle advection, parallel particle advection, and particle advection using the GPU. More about the implementations will be discussed in the next section. Additionally, the memory usage of the application is monitored using the Android Profiler tool. The collected data is processed and analyzed and, consequently, used to answer the research question by comparing the application's performance under different computational loads.

4 Implementation

4.1 Android Native C++ implementation

The implementation consists of two main parts: the Android Java-based implementation and the native C++ implementation. The Java-based implementation is a wrapper for the native C++ implementation to make it compatible with the Android platform. It extends the Android `MainActivity` class, which is the application's entry point. This class then handles the application's user interface, permissions, file input, display surfaces, and update calls. The native C++ implementation is the application's core, handling the fluid simulation, rendering, and data processing. The two parts communicate using the Java Native Interface (JNI). The Java-based implementation calls the native C++ functions to perform the fluid simulation and rendering. This communication is visualized in Figure 4.1.



(a) *Initialization*

(b) *Application loop*

Figure 4.1: *The communication between the Java-based implementation and the native C++ implementation.*

4.2 File input

The application requires a vector field to define the particles' movement. This vector field is read from NetCDF files located on the device.

However, accessing files on Android devices directly from native code is not straightforward. Therefore, the application uses the Android Java-based implementation to request permissions to access the file system and asks the user to select the files containing the vector field. Subsequently, the Java-based implementation passes the file descriptors to the native C++ implementation using JNI.

Additionally, reading all the time steps of the entire vector field at once would require a large amount of memory, which is unfeasible for smart device(s). Thus, the vector field is loaded one time step at a time. Time-wise linear interpolation is then performed between the two nearest time steps to obtain the vector field at the desired time. The application thus requires two sets of files for each time step: the files containing the time step before the desired time and the files containing the time step after the desired time. However, to avoid lagging when switching files, a third set of files is loaded in the background, containing the data that are two time steps ahead. Processing these files in the background and uploading them to the GPU if necessary ensures that the application can quickly switch between vector field files without having to wait for loading and processing when requested.

4.3 Graphics

The application uses OpenGL ES 3.2. This version provides access to compute and geometry shaders while still being compatible with most modern Android devices.

4.3.1 Shader setup

Two main shader programs are used in the rendering part of the application: the vector field rendering shader program and the particle rendering shader program. A diagram of the rendering pipeline is shown in Figure 4.2.

4.3.2 Particle rendering

The particle rendering shader program renders the particles as simple points in 3D space of constant size and color. The particles are defined as a flat array of floats, where each particle is represented by three consecutive floats - its position in 3D space. The particle positions are stored in a GPU Vertex Buffer Object (VBO). The application uses a Vertex Array Object (VAO) to store the VBO. The VAO is then used to render the particles. However, with this setup, every time the particles are updated outside the GPU, the VBO needs to be uploaded to the GPU again.

4.3.3 Vector field rendering

The vector field shader program renders the vector field as a set of lines in 3D space. The lines are colored based on the vectors' directions to make the directions visually distinguishable. The color is calculated in the geometry shader by mapping the vector's direction to the HSV color space and converting it to RGB.

Displaying every vector in the vector field can be computationally expensive, especially for large vector fields with millions of vertices. The application uses a downsampling technique to display only a subset of the vectors in order to reduce the computational cost. The subset, call it display vertices, is selected by only rendering every n_x-th vector in the x-direction, every n_y-th vector in the y-direction, and every n_z-th vector in the z-direction. The downsampling factors n_x, n_y, and n_z can be adjusted to balance the visual quality and performance of the application.

The vector field is defined as a flat array of floats, where each vector is represented by six consecutive floats - the position of the two points that define the vector in 3D space. Before rendering the vector field, the application linearly interpolates each vector in the

display vertices to obtain the vectors' positions at the desired time step. The interpolated vectors are then sent to the GPU for rendering.

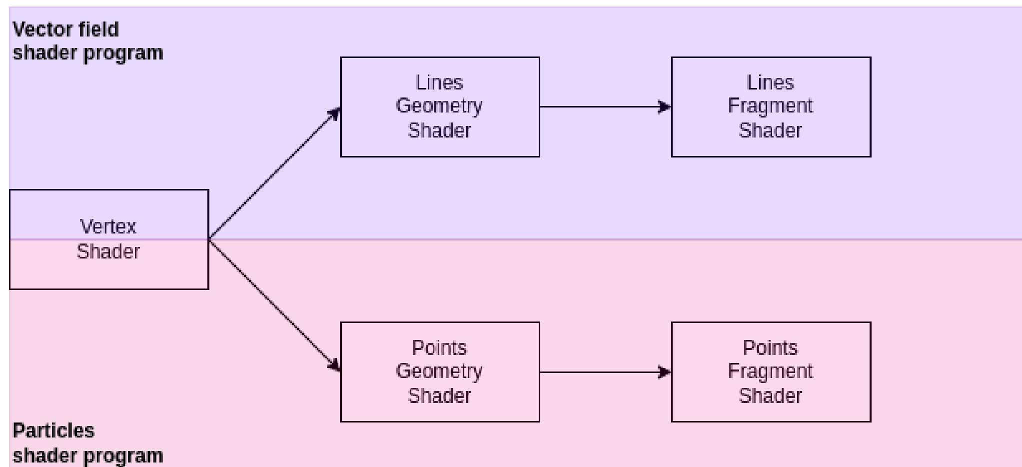


Figure 4.2: *The rendering pipeline of the application.*

4.4 Simulation

The application implements three distinct methods for simulating the particles. Sorted ascendingly by complexity, these methods are sequential particle advection, parallel particle advection, and particle advection using the GPU.
From equation (3.19), and equations (3.2) to (3.7), we can see that to perform a particle's advection, we only need to know the particle's position and the value of the velocity field at the particle's position. This means that the advection of each particle is independent of the advection of other particles. Thus, the advection of particles can easily be performed in parallel - the main idea behind the parallel particle advection method and the particle advection using the GPU method.

The sequential particle advection method uses the CPU to perform the advection of particles sequentially. The method iterates over all particles and calculates the new position of each particle one by one. It is the simplest and most straightforward method to implement.

The parallel particle advection method uses the CPU to perform the advection of particles in parallel. The method divides the particles into batches and assigns each batch to a separate CPU thread. Each thread then calculates the new position of the particles in its batch, and after all threads have finished, the new positions are updated. The application uses a thread pool to manage the threads and their tasks to avoid the overhead of constantly creating and destroying threads

The particle advection using the GPU method uses the GPU, instead of the CPU, to perform the advection of particles. Updating the particles' positions means performing the same arithmetic operations on each particle. Thus, the particle advection is a perfect use case for Single Instruction, Multiple Data (SIMD) parallel processing. The GPU is designed to handle SIMD parallelism efficiently, making it an ideal candidate for this task. Therefore, the particle advection in this method is implemented using the OpenGL ES compute shader. The compute shader is executed in a grid of workgroups, where each workgroup contains multiple threads that execute the same code on different data.
A Shader Storage Buffer Object (SSBO) is used to mediate the reading of old particle

positions and the writing of new particle positions. The SSBO is a VBO that stores the particles' positions but allows for both read and write operations in the compute shader. Since the SSBO stores the particles' positions in this method, they can be directly used to render the particles. The implementation uses an updated VAO that stores the SSBO on top of the VBO. As before, the VAO is then used to render the particles. This setup eliminates the overhead of CPU-GPU communication at every simulation time step.

4.5 Third-party libraries

The application uses the OpenGL Mathematics (GLM) library for linear algebra operations. The library provides a wide range of vector and matrix operations functions, making it ideal for the application's needs. The application also uses the NetCDF C++ library to read the vector field from NetCDF files. The library provides functions to read and write NetCDF files, making accessing the vector field data easy. Additionally, the application adapts a popular thread pool library to manage threads efficiently. The library provides functions to create and manage a thread pool as well as assign tasks to threads. The functionality to wait for all threads to finish their tasks had to be implemented.

The application uses the Android Native Development Kit (NDK) to compile the C++ code and the third-party libraries. Header-only libraries like GLM are included directly in the C++ code, while the NetCDF C++ library and its dependencies were compiled separately as shared libraries and linked to the application.

5 Results

5.1 Performance comparison

In this section, the performance of the three simulation methods is compared. The performance is measured in terms of render time, simulation compute time, file loading time, application cycle time, and wall clock time, as described in the section 3.3

5.1.1 Render time

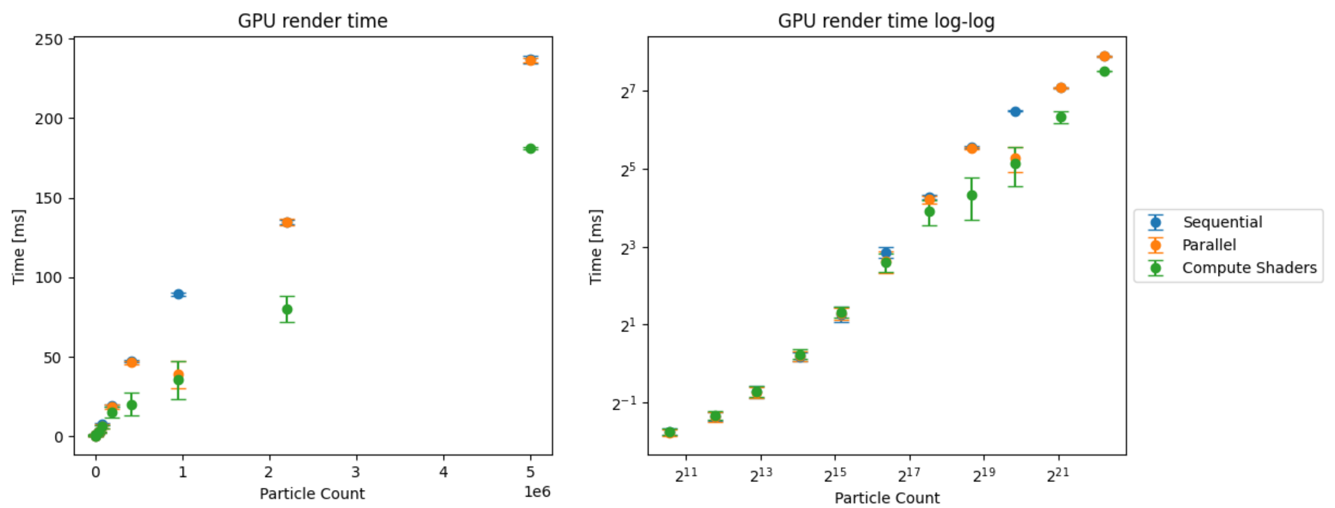


Figure 5.1: *The graph on the left shows the render time for all three methods, while the graph on the right shows the same data plotted on a logarithmic scale.*

The render time across three computational methods is depicted in Figure 5.1, with each graph representing the average time per application cycle over numerous cycles.
Notably, the graph on the right illustrates minimal variation in render times across methods for smaller particle counts, i.e., fewer than 2^{15} particles. All three methods show a direct proportionality between render time and particle count. However, the GPU method outperforms the others at higher particle counts, as highlighted by the left graph in Figure 5.1. This graph also indicates that all methods' initial render time (y-intercept) is nearly zero.

5.1.2 Simulation compute time

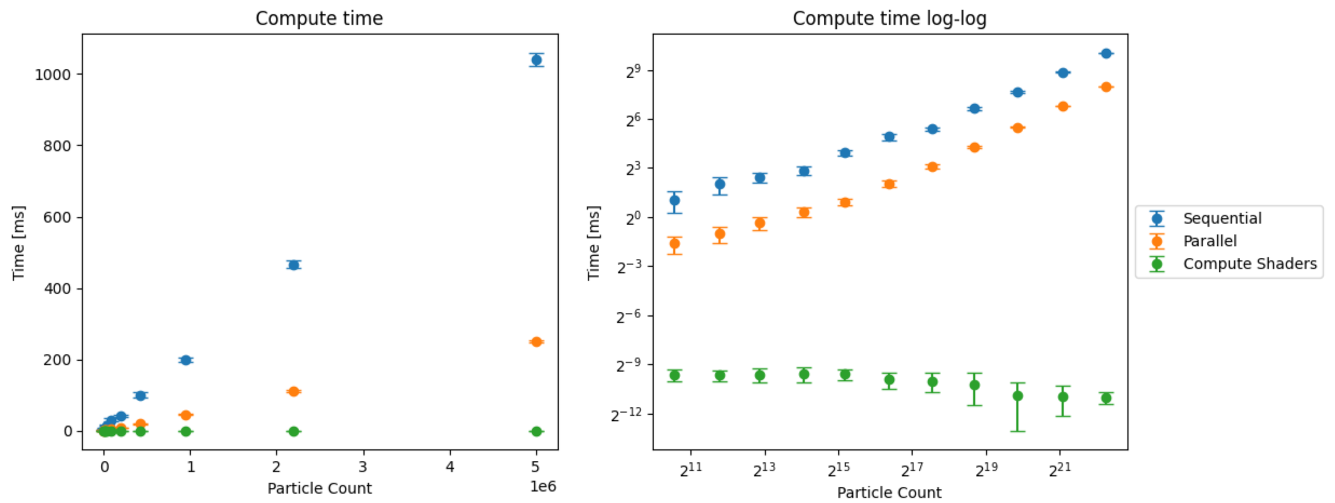


Figure 5.2: *The graph on the left shows the simulation time for all three methods, while the graph on the right shows the same data plotted on a logarithmic scale.*

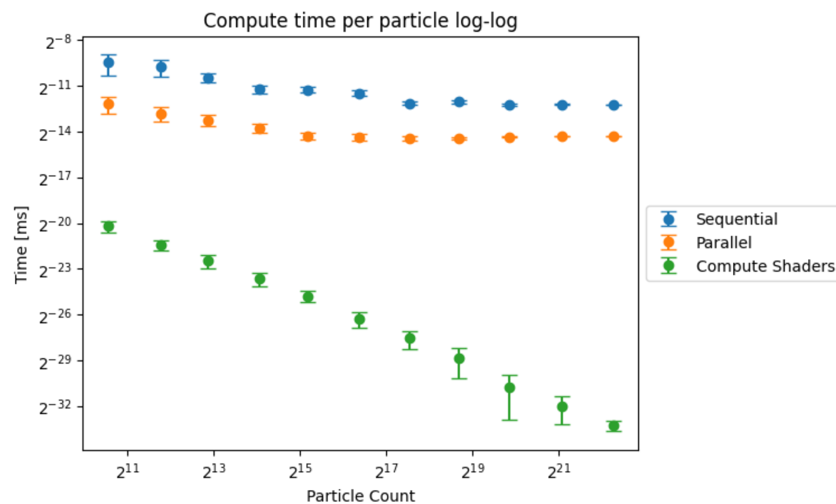


Figure 5.3: *Simulation time per particle for the three methods plotted on a logarithmic scale.*

Figure 5.2 examines the simulation compute time for the same three methods, further detailed per particle in Figure 5.3. The GPU method consistently achieves the fastest simulation times, surpassing both the parallel and sequential methods. While the sequential and parallel methods exhibit a linear increase in compute time, the GPU method maintains a nearly constant time across different particle counts. Additionally, the slope of the trend for the parallel method is considerably lower than that of the sequential method. As particle counts increase, both the sequential and parallel methods show a slight reduction in simulation time per particle, which then stagnates beyond 2^{17} particles. Conversely, the GPU method reduces its simulation time per particle as the count increases and continues to decrease without stagnating.

5.1.3 File loading time

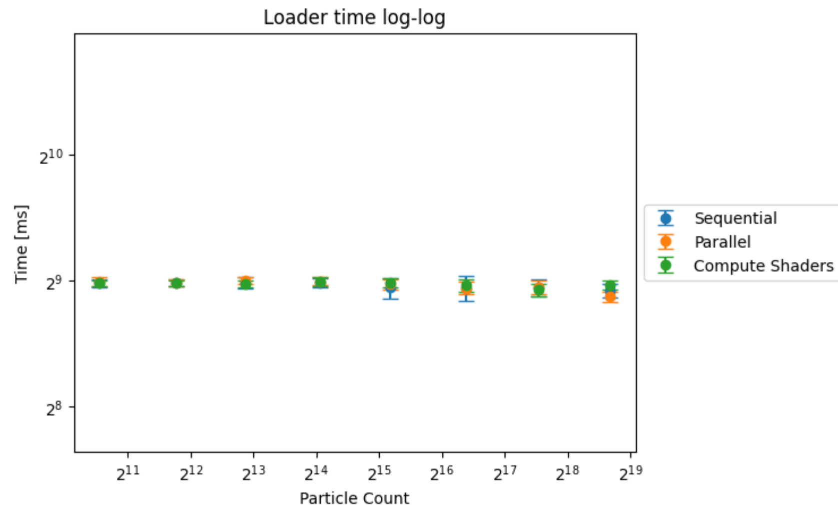


Figure 5.4: *File loading time for the three methods plotted on a logarithmic scale.*

The graph in Figure 5.4 shows the time it takes to load, process, and upload the data for all (three) files necessary for preparing the next simulation day. As we can see, the value constantly remains at around 2^9 milliseconds for all three methods, and does not change with the number of simulated particles.

5.1.4 Application cycle time

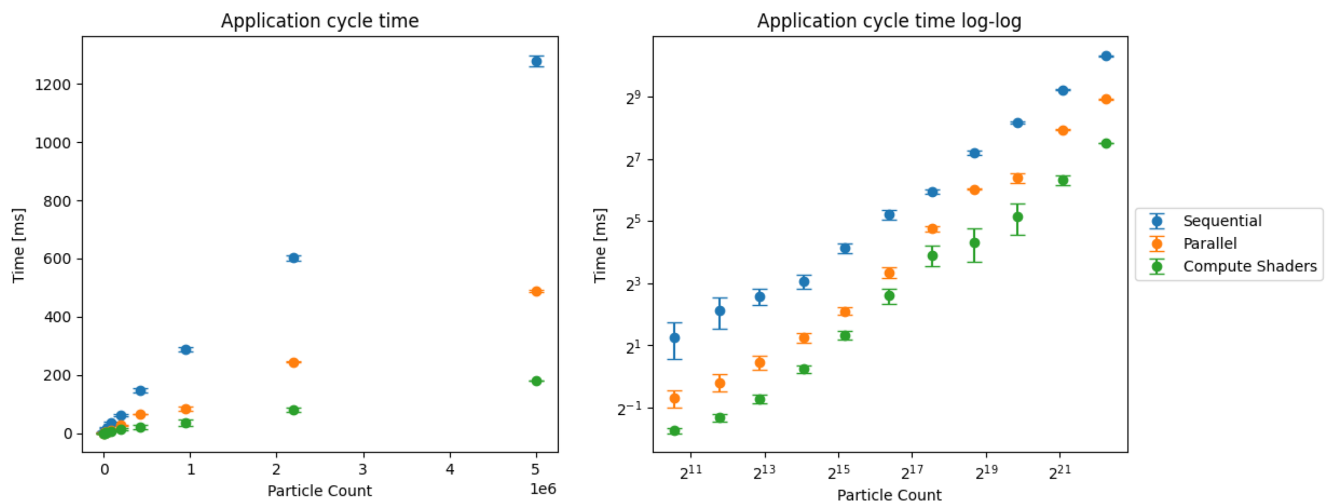


Figure 5.5: *The graph on the left shows the application cycle time for all three methods, while the graph on the right shows the same data plotted on a logarithmic scale.*

The graphs in Figure 5.5 display the application cycle time for the three methods. This time is the average time for a single application cycle as measured by the CPU and GPU internal timers. We can see from both graphs that all three methods experience a linear increase in application cycle time as the number of particles increases. However, the rate of this increase is different for each method - lowest for the GPU method, followed by the parallel method, and then the sequential method.

5.1.5 Wall clock time

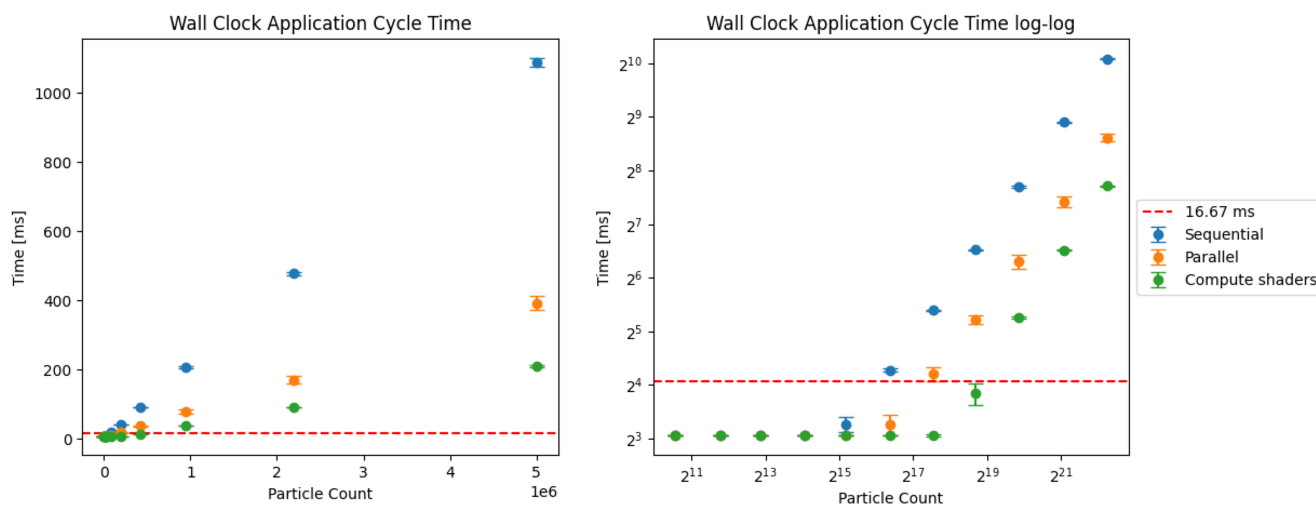


Figure 5.6: *The graph on the left shows the wall clock time for all three methods, while the graph on the right shows the same data plotted on a logarithmic scale. The red dashed line represents the 60 fps threshold.*

Figure 5.6 presents the average wall clock time measurements for completing a single application cycle across the three methods. In both graphs, we observe that the overall trends for large particle counts mirror those noted in the previous subsection, where CPU and GPU internal timers were used. However, the right graph shows that the wall clock time remains constant for the initial data points, with each method diverging at different particle thresholds: the sequential method at approximately 2^{15} particles, the parallel method at about 2^{16} particles, and the GPU method not until roughly 2^{18} particles. This graph also highlights the maximum particle counts at which each method maintains an interactive frame rate of 60 fps, translating to a wall clock time of approximately less than 16.67 milliseconds. The GPU method sustains this up to 2^{19} particles, significantly higher than the parallel method at 2^{16} and the sequential method at 2^{15}.

5.2 Application cycle composition

This section shows the composition of the application cycle for the three methods. The application cycle is divided into the following parts: render time, simulation compute time, and file loading time. All times were measured by the CPU and GPU internal timers and averaged over many cycles.

The graphs in Figures 5.7, 5.8, and 5.9 illustrate the composition of the application cycle for the sequential, parallel, and GPU methods, respectively. Across all methods, while file load time initially impacts the application cycle, it decreases in significance as particle counts increase. This trend allows other components to influence the cycle's composition more prominently at higher particle counts.

5.2.1 Sequential method

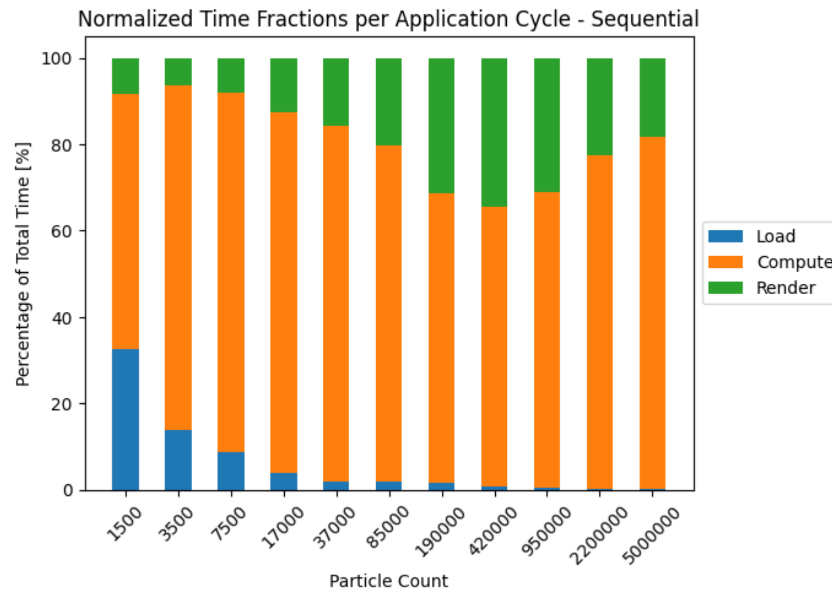


Figure 5.7: *Composition of the application cycle for the sequential method.*

The graph in Figure 5.7 showcases the composition of the application cycle for the sequential method. We can see that the compute time is the most significant part of the application cycle. We can observe that the render time's fraction keeps growing for as long as the load time's fraction keeps decreasing. However, after the load time becomes negligible, the render time's fraction begins to decline while the compute time's fraction keeps increasing.

5.2.2 Parallel method

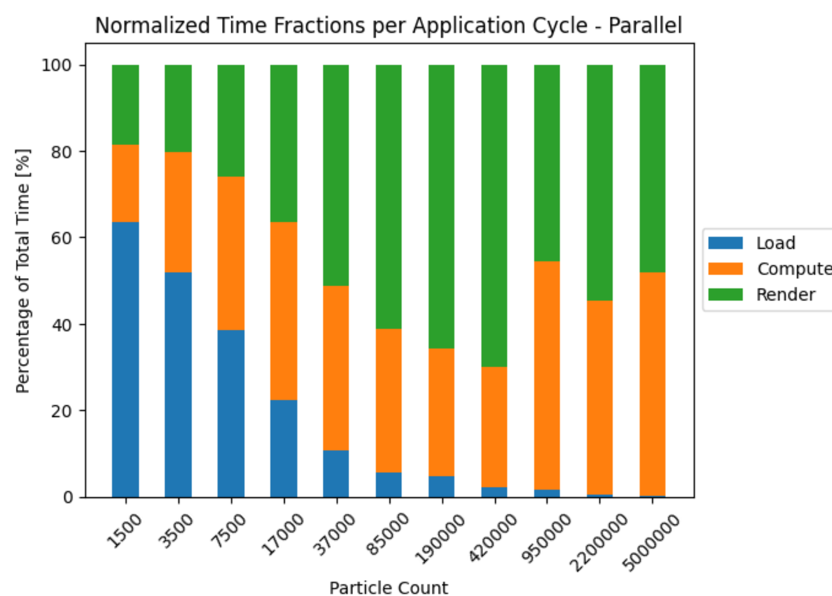


Figure 5.8: *Composition of the application cycle for the parallel method.*

The graph in Figure 5.8 shows the composition of the application cycle for the parallel method. The compute time forms a significantly smaller part of the application cycle

compared to the sequential method. The render time dominates the application cycle for particle counts between 3 700 and 420 000. For particle counts larger than 420 000, the compute time and the render time are approximately equal in size. The render time's fraction grows and declines similarly to the sequential method, but the compute time's fraction grows at a slower rate.

5.2.3 GPU method

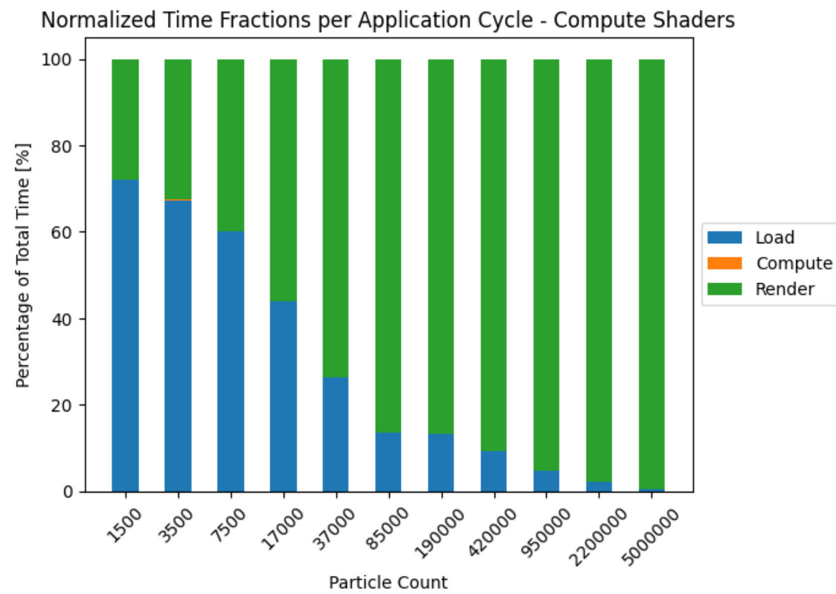


Figure 5.9: *Composition of the application cycle for the GPU method.*

The composition of the application cycle for the GPU method is displayed in the graph in Figure 5.9. For large particle counts, the render time forms the most significant part of the application cycle. The compute time is the smallest fraction of the application cycle for all measurements and is practically negligible in comparison. We also observe that the render time's fraction keeps growing for all particle counts measured.

5.3 Memory usage

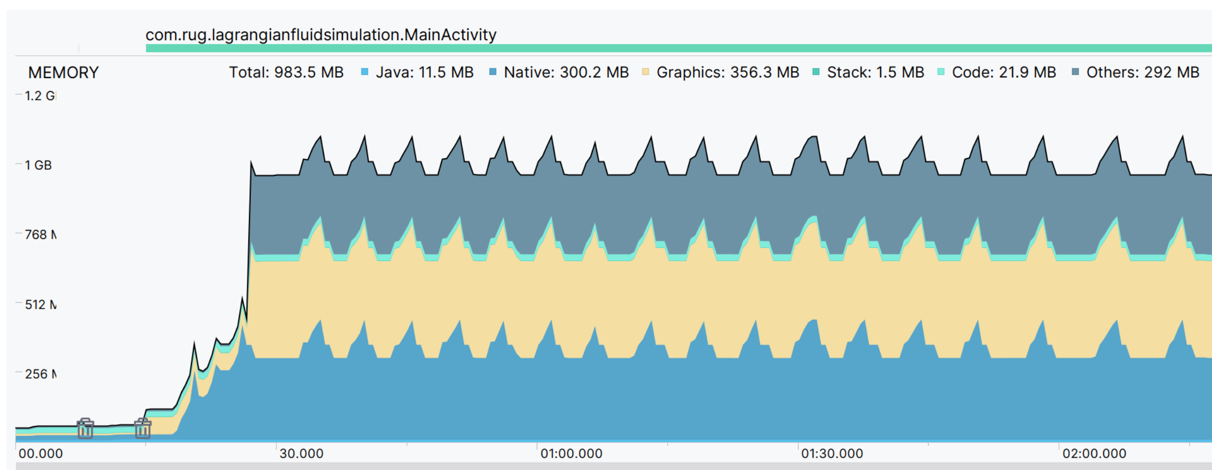


Figure 5.10: *Memory usage for the GPU method for* 250 000 *particles.*

Overall memory utilization of the application over time is depicted in Figure 5.10. The graph shows the memory consumption associated with the GPU method when simulating 250 000 particles. Memory usage is divided into multiple components, with Native allocations, Graphics, and Other allocations being the most significant. From the graph in Figure 5.10, we see that initial memory usage is minimal; however, it rapidly increases, exhibiting four distinct peaks. Subsequently, the memory usage stabilizes at approximately 1 GB with perpetual peaks in the memory of the Native allocations. The memory usage of the other categories remains stable with negligible fluctuations.

6 Discussion

6.1 Performance comparison

As depicted in Figure 5.1, the render time demonstrates a linear relationship with particle count, being directly proportional to the number of displayed vertices. The constant number of displayed vectors ensures that increasing render times are solely attributed to the particle count. From the graph on the left in Figure 5.1, we observed that the y-intercept is near zero. This can be attributed to the constant overhead of OpenGL ES draw calls being negligible compared to the time it takes to render all objects. The time it takes to render the vector field is also insignificant, especially for large particle counts, due to the vector field's reduction of display vertices. Particularly for the testing, the vector field grid was reduced from $539 \times 269 \times 28 = 4\ 059\ 748$ vertices to $36 \times 18 \times 6 \approx 3\ 609$ vertices. This reduction is the reason why the render and compute times are very similar for small particle counts, where the number of rendered vertices for the vector field is comparable to those for the particles. The GPU method becomes more efficient for large particle counts due to another factor: advecting the particles on the GPU means that the particle data does not need to be uploaded from the CPU to the GPU, lowering CPU-GPU communication time.

Figure 5.2 illustrates that the GPU method outperforms both parallel and sequential methods, with nearly constant simulation times across varying particle counts. This is attributed to the GPU's capacity for efficient SIMD operations. In contrast, the sequential method, lacking such optimizations, shows the poorest performance. The graph on the right in Figure 5.2 shows that the GPU method's simulation time is almost constant for all particle counts. Figure 5.3 also nicely demonstrates this phenomenon, as the simulation compute time per particle decreases as the number of particles increases. Unfortunately, the efficiency of the GPU method is limited by the bottleneck discussed in section 6.4.

Load times remain constant regardless of particle count, as demonstrated by the data in Figure 5.4. This consistency is expected given that the vector field data loading and processing is not dependent on the number of simulated particles.

Application cycle time, represented in Figure 5.5, affirms the overall efficiency of the GPU method, which exhibits the lowest times across all particle counts. The parallel method ranks second, followed by the sequential method. The application cycle time displayed in Figure 5.5 is the sum of the render time and the simulation time. Thus, the results are consistent since the GPU method performed the best in both render and simulation times.

Lastly, the graph in Figure 5.6 shows that the GPU method is the most efficient for all measured particle counts also in terms of wall clock time. This time covers all operations, including those unrelated to the test application, due to the nature of the wall clock timer, which captures the globally-elapsed system time, including background system activities. During the measurements, efforts were made to keep the number of background processes to a minimum, but eliminating all other processes completely is unachievable.

In order to maintain a 60 fps frame rate, each of the three simulation modes has a distinct limit in particle count. These limits are approximately 2^{19} particles (around half a million) for the GPU method, 2^{16} particles (between 50 000 and 100 000) for the parallel method,

and 2^{15} particles (below 50 000) for the sequential method. Particle counts smaller or equal to these limits ensure the application can handle the computational load within the desired frame rate.

The application cycle's wall clock time measurements are also affected by v-sync, which limits the refresh rate to 120 Hz. This limitation means that the application cycle's wall clock time can not be lower than $\frac{1}{120\text{ Hz}} \approx 8.33$ milliseconds, as even if the application cycle is shorter, the next update call will not be issued earlier than 8.33 milliseconds after the previous one. This delay explains why the wall clock time measurements in Figure 5.6 appear constant for small particle counts.
Decoupling the application cycle and refresh rate does not make sense for a real-time application for devices with limited computational resources, as pre-computing frames ahead of time is not feasible.

6.2 Application cycle composition

The sequential method's application cycle composition in Figure 5.7 shows that the simulation compute time is the most significant part of the application cycle for all measured particle counts. This observation is expected, as the sequential method performs all calculations in a naive sequential manner.
Further analysis of Figure 5.7 reveals that the render time's fraction increases as long as the load time's fraction decreases. This behavior is expected, as the render time is directly proportional to the number of particles while the load time is constant. However, when the load time's fraction becomes negligible, the render time's fraction starts to decrease since the render time grows slower than the simulation time.

In contrast, the application cycle composition for the parallel method, illustrated in Figure 5.8, reflects a reduced fraction of the simulation compute time due to the method's parallel processing capabilities.

Lastly, for the GPU method's application cycle composition depicted in Figure 5.9, the fraction of the simulation compute time remains negligible for all measured particle counts. This observation is expected since we saw the simulation time being constant in Figure 5.2. The constant value of the simulation time also explains why the fraction of the render time only increases with the number of particles. The overwhelming dominance of the render time in the application cycle composition for large particle counts demonstrates how the render time and its related operations are the main bottleneck for the GPU method.

6.3 Memory usage

The memory usage of the application is measured only for the GPU method, as it is the most performant of the three methods. Additionally, the most significant contributor to memory consumption is vector field data, which is handled with minor differences across the three methods. Consequently, the memory usage of the GPU method serves as a representative indicator of the application's overall memory characteristics. The associated graph in Figure 5.10 illustrates this usage pattern. At the beginning of the measurement, the app was not yet fully initialized, waiting for the user to select the vector field files. This initialization process is the reason for the initial memory usage spike. We can observe that four sharp increases in memory usage constitute this initial memory spike. The

first three correspond to the loading process, as detailed in section 4.2, where the sharp increases correlate with loading three distinct sets of vector field files into memory. After the data is loaded, the application instantiates all the classes and objects needed for the simulation, creating the fourth spike. Following this, memory consumption stabilizes at approximately 1 GB. Additional periodic spikes are noted throughout the simulation's duration, coinciding with the loading of subsequent vector field files. The memory consumption data indicates that the application's memory usage remains consistent with time. Thus, looking back at the secondary research question, the application has good long-term operational efficiency.
The composition of the application's memory usage can be categorized into three main components: Graphics, Native, and Others. The Graphics component includes memory used for rendering the vector field and particles, which are managed by the GPU and OpenGL ES directly. The Native component includes memory used for running the actual simulation and its associated data, which are managed outside the Java Virtual Machine and directly utilize system-level resources. The 'Others' category captures miscellaneous or unclassified memory usages, such as temporary data buffers and system caches not immediately related to Java or Native operations.

6.4 Bottlenecks

A discrepancy can be observed between the application cycle time measured by the internal CPU and GPU timers shown in Figure 5.5 and the application cycle's wall clock time measurements shown in Figure 5.6. This discrepancy is partially caused by the exclusion of the load time from the measurements in Figure 5.5. However, since the loading happens asynchronously with the rest of the application, the load time is distributed over many application cycles. It thus cannot be the sole cause of the discrepancy.
One particular bottleneck was observed, which is the leading cause of the discrepancy. This bottleneck is the synchronization and swapping between the different types of buffers used by the GPU. The bottleneck is especially noticeable in the GPU method, which inherently utilizes more buffers. Thus, while only synchronization and swapping between the display and surface buffers are needed for the sequential and parallel methods, the GPU method requires more steps. Specifically, the GPU method necessitates synchronization and swapping between the SSBO, which the compute shader writes into the updated particle positions, and the VBO read by the vertex shader for rendering. Even though the SSBO and the VBO are physically the same data blocks, they are used in a different context by the two shader programs, and thus synchronization is needed. This intensive memory swapping and synchronization requirement essentially make the simulation on the GPU an I/O-bound computing problem rather than purely compute-bound. This bottleneck is also the main reason the GPU method's performance degrades with increasing particle counts, highlighting the critical impact of internal I/O operations on the overall performance.

Additionally, one more bottleneck is observed in the GPU method. The bottleneck is especially noticeable when the particles are randomly distributed in a dense grid. In this case, two major factors contribute to the bottleneck.
Firstly, the particles in the thread groups are far apart in memory. The GPU utilizes peak memory bandwidth when multiple SIMD unit threads, i.e., warps, access consecutive memory locations simultaneously. When threads in a warp access data far apart in memory, the memory controller might need to issue multiple memory transactions to gather the data for all threads in the warp instead of a single transaction for aligned,

consecutive memory accesses.
Secondly, the grid interpolation is performed on the GPU, which requires a lot of different vector field vertices to be accessed for each particle when the particles in the thread groups are far apart. If a warp accesses data points that are spread far apart, the likelihood of these data points being in the same cache line is low, leading to poor utilization of the cache. This increases cache misses, where the GPU looks for data in the cache but does not find it and thus has to go back to slower, global memory reads. There is no simple way to fix this bottleneck, as the particles would need to be sorted in a way where nearby particles are in the same thread group. For example, an efficient tree-like data structure on GPU architectures would need to be implemented to sort the particles in linear or sublinear time. Otherwise, the fix becomes the new bottleneck. Such a data structure, however, does not exist [23].

Unfortunately, with the current implementation, the time impact of either of the bottlenecks cannot be measured directly using the internal CPU and GPU timers. The current implementation uses the `MainActivity` class with JNI to call the native C++ function. This means the Java code creates and manages the EGL context responsible for the buffer synchronization and swapping. However, the internal CPU and GPU timers are only available in the native C++ code, which does not provide a way to measure the GPU execution time of Java function calls. A potential solution is to use the `NativeActivity` class, which moves the total control of the app to the native C++ code. However, with this solution, not only would the EGL context need to be explicitly handled by the native C++ code, but all event handling and other Android-specific features also need to be handled explicitly. This switch is not arbitrary; the resulting project would suffer in maintainability and expandability.

6.5 Impact

The application allows the user to load different vector fields. The only conditions to be met are for the data to be in a glsNetCDF format and split into separate files per velocity component per timestep. Due to this flexible implementation, many different kinds of vector fields can possibly be displayed and used for simulation.

The user can interact with the application by touching the device's screen. Currently, camera rotation and zooming are supported. These simple interactions allow the user to view the simulation from different angles and distances. This way, the application can visually represent the vector field and simulate the particle movement in this vector field. The application can also be used to study the behavior of particles in different vector fields or to observe how the particles behave in the vector field over time.

The application supports adding UI elements like menu bars or modal dialogs by defining these components in the XML layout files. The behavior and functionality of these elements can be developed within the Java context, which can further mediate the functionality to the native C++ code if required. Additionally, the application provides the implementation basis for the UI, enabling users to configure simulation parameters, such as the number of particles, the displayed vector field density, or the simulation accuracy. Parameters entered by the user into the UI elements could be read by the application and passed to the appropriate class constructors. This setup shows the simplicity of integrating functions like 'Make Screenshot' and 'Toggle Vector Field' into broader applications.

However, the implementation of the UI itself is out of the scope of this project, as the specific requirements for the UI will vary depending on the application scenario.

6.6 Future work

Possibly the most crucial future work is to minimize the bottlenecks described in section 6.4. Some optimization of these bottlenecks has already been done, such as reducing the number of buffer swaps and synchronizations by efficiently handling buffer data and drawing calls. However, further optimization would improve the application's performance. Especially for the bottleneck affecting the most efficient GPU method, the application could be extended to handle better the uneven load distribution caused by non-consecutive data accesses in the compute shader in the case of chaotic particle distributions in dense vector fields.

Different types of vector field visualization methods were experimented with during the project. These include line integral convolution and 3D texture slicing. Details about the individual experimental visualization methods are located in Appendix A. Ultimately, the measurements were taken using the simplest method - displaying both the vector field and the particles using OpenGL ES primitive types.
Instanced rendering is not beneficial for a simplistic approach like this, as rendered objects do not share many of the same attributes across instances. However, instanced rendering could become beneficial for future, more complex visualization methods. Further experimentation on the feasible vector field visualization methods needs to be done. The application could even be extended to support multiple visualization methods simultaneously, allowing the user to switch between them. These might include the already mentioned methods and other methods, such as direct volume rendering.

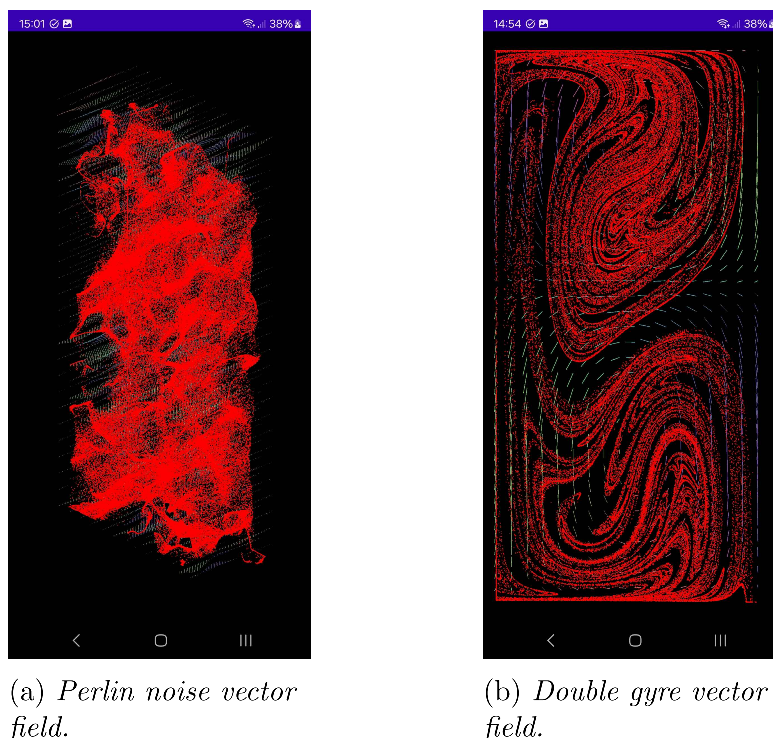
7 Conclusion

This research project focused on developing a fluid simulation Android application optimized for smart device(s), addressing the growing requirement for on-demand accessible fluid simulations across various applications and fields. By bridging the gap between computational physics and computing science, an efficient fluid simulation was created using a compiled language, C++, and leveraging the OpenGL ES graphics API. Example images of the developed application are shown in figure 7.1.

The simulation employs a Lagrangian-Eulerian approach, solving the particle advection numerically as defined by equation 3.19 within a vector field defined as an Eulerian grid. The advection equation is numerically solved using the Runge-Kutta 4th order method, and the simulation is implemented via three distinct approaches: sequential particle advection, parallel particle advection, and GPU-based particle advection.

The performance evaluations for all implementations under various computational loads revealed that GPU-based particle advection is the most effective method. Utilizing compute shaders enables efficient simulation of approximately up to $2^{19} \approx 500\,000$ particles on a Samsung Galaxy S23 Ultra. Moreover, the memory usage evaluation demonstrates that the GPU method maintains a consistent memory footprint of approximately 1 GB after initialization, thereby demonstrating the application's long-term operational efficiency in managing resources on smart device(s). Nonetheless, challenges remain, particularly with the rendering phase and the necessary buffer swapping and synchronization, which continue to act as bottlenecks.

In summary, simulating particles on smart device(s) is feasible and shows promising results, especially when compute shaders are utilized.



(a) *Perlin noise vector field.*

(b) *Double gyre vector field.*

Figure 7.1: *Images of the fluid simulation Android application for* 250 000 *particles.*

8 References

[1] J. Blazek, *Computational Fluid Dynamics: Principles and Applications*, 3rd. Butterworth-Heinemann, 2015, ISBN: 9780128011720. DOI: 10.1016/C2013-0-19038-1.

[2] L. Moubin and L. Gui-rong, *Smoothed particle hydrodynamics: A meshfree particle method.*, 2003. [Online]. Available: http://search.ebscohost.com.proxy-ub.rug.nl/login.aspx?direct=true&db=nlebk&AN=134095&site=ehost-live&scope=site.

[3] C. Jiang, *The Material Point Method for the Physics-Based Simulation of Solids and Fluids*. UCLA, 2015. [Online]. Available: https://escholarship.org/uc/item/8090m32r.

[4] D. Kim, *Fluid Engine Development*. CRC Press, 2016, ISBN: 9781498719933.

[5] F. Zehra, M. Javed, D. Khan, and M. Pasha, *Comparative analysis of c++ and python in terms of memory and time*, Preprints, 2020. DOI: 10.20944/preprints202012.0516.v1.

[6] M. P. Singh and M. K. Jain, *Evolution of processor architecture in mobile phones*, International Journal of Computer Applications, vol. 90, no. 4, pp. 34–39, 2014. DOI: 10.5120/15564-4339.

[7] S. Park, S. An, and B. So, *Boosting the performance of python-based geodynamic code using the just-in-time compiler*, Geophysics and Geophysical Exploration, vol. 24, no. 2, pp. 35–44, 2021. DOI: 10.7582/GGE.2021.24.2.35.

[8] C. Kehl, E. van Sebille, and A. Gibson, *Speeding up python-based lagrangian fluid-flow particle simulations via dynamic collection data structures*, 2021. arXiv: 2105.00057.

[9] P. Mora, G. Morra, and D. A. Yuen, *A concise python implementation of the lattice boltzmann method on hpc for geo-fluid flow*, Geophysical Journal International, vol. 220, no. 1, pp. 682–702, 2020, ISSN: 0956-540X. DOI: 10.1093/gji/ggz423.

[10] J. Akeret, L. Gamper, A. Amara, and A. Refregier, *Hope: A python just-in-time compiler for astrophysical computations*, Astronomy and Computing, vol. 10, pp. 1–8, 2015, ISSN: 2213-1337. DOI: 10.1016/j.ascom.2014.12.001.

[11] R. Bridson, *Fluid simulation for computer graphics*. CRC press, 2015. DOI: 10.1201/9781315266008.

[12] M. Kelager, *Lagrangian fluid dynamics using smoothed particle hydrodynamics*, Department of Computer Science. University of Copenhagen, p. 59, 2006. [Online]. Available: http://glowinggoo.com/sph/bin/kelager.06.pdf.

[13] J. Monaghan, *Smoothed particle hydrodynamics*, Annu. Rev. Astron. Astrophys, vol. 543, p. 74, 1992. DOI: 10.1146/annurev.aa.30.090192.002551.

[14] S. Markidis, P. Henri, G. Lapenta, *et al.*, *The fluid-kinetic particle-in-cell method for plasma simulations*, Journal of Computational Physics, vol. 271, pp. 415–429, 2014, Frontiers in Computational Physics, ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2014.02.002.

[15] J. Lu, J. Piao, and S. Kim, *Optimizing on real-time fluid 3d effect in mobile environment*, in *Proceedings of the 9th International Conference on Ubiquitous Information Management and Communication*, Bali, Indonesia: Association for Computing Machinery, 2015, ISBN: 9781450333771. DOI: 10.1145/2701126.2701147.

[16] J. R. Wilson and K. C. Gramoll, *Viscous fluid dynamics app for mobile devices using a remote high performance cluster*, in *2015 ASEE Annual Conference & Exposition*, Seattle, Washington: ASEE Conferences, 2015. DOI: 10.18260/p.25042.

[17] P. Delandmeter and E. van Sebille, *The parcels v2.0 lagrangian framework: New field interpolation schemes*, Geoscientific Model Development, vol. 12, no. 8, pp. 3571–3584, 2019. DOI: 10.5194/gmd-12-3571-2019.

[18] E. Simonnet, M. Ghil, and H. Dijkstra, *Homoclinic bifurcations in the quasi-geostrophic double-gyre circulation*, English, Journal of Marine Research, vol. 63, no. 5, pp. 931–956, 2005, ISSN: 0022-2402. DOI: 10.1357/002224005774464210.

[19] R. K. Rew and G. P. Davis, *Netcdf: An interface for scientific data access*, IEEE Computer Graphics and Applications, vol. 10, no. 4, pp. 76–82, 1990. DOI: 10.1109/38.56302.

[20] S. C. Shadden, F. Lekien, and J. E. Marsden, *Definition and properties of Lagrangian coherent structures from finite-time Lyapunov exponents in two-dimensional aperiodic flows*. 2005, vol. 212, pp. 291–292. DOI: 10.1016/j.physd.2005.10.007.

[21] W. H. Press, *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007, pp. 907–915, ISBN: 9780521880688.

[22] D. R. Durran, *Numerical methods for fluid dynamics: With applications to geophysics*. Springer Science & Business Media, 2010, vol. 32, pp. 12–13, 358–390. DOI: 10.1007/978-1-4419-6412-0.

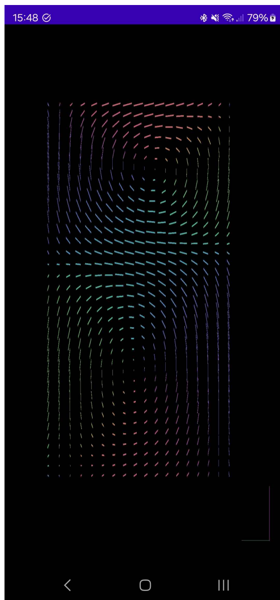
[23] N. Fauzia, L.-N. Pouchet, and P. Sadayappan, *Characterizing and enhancing global memory data coalescing on gpus*, in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, San Francisco, CA, USA, 2015, pp. 12–22. DOI: 10.1109/CGO.2015.7054183.

[24] J. R. Taylor and W. Thompson, *An introduction to error analysis: the study of uncertainties in physical measurements*. Springer, 1982, vol. 2, pp. 45–154, ISBN: 9780935702750.

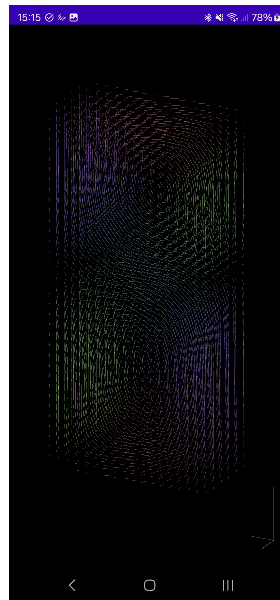
Appendix A

Implemented visualization techniques

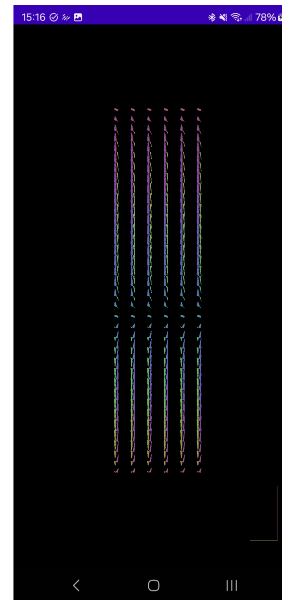
The different vector field visualization techniques implemented for the fluid simulation Android application are presented in this appendix. Figure A.1 shows the simple vector field visualization technique that is used to measure the performance (see section 4.3.3). In contrast, Figure A.2 presents a visualization technique where the outline of the vector field is displayed as an opaque cuboid. In both methods, the colors are based on the vectors' directions and mapped using an HSV mapping.



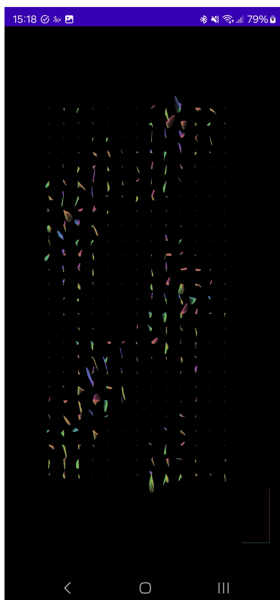
(a) *Double gyre field x-y plane view.*



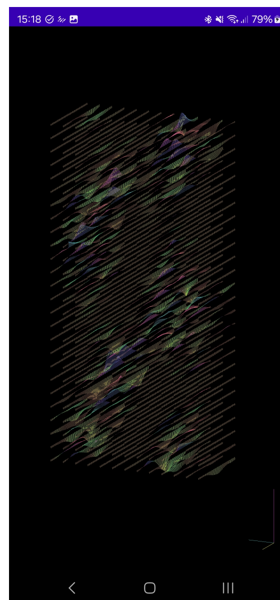
(b) *Double gyre field rotated.*



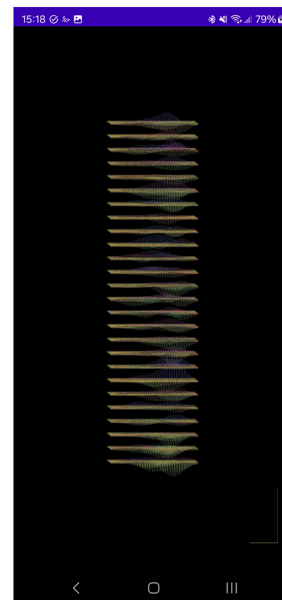
(c) *Double gyre field z-y plane view.*



(d) *Perlin noise field x-y plane view.*



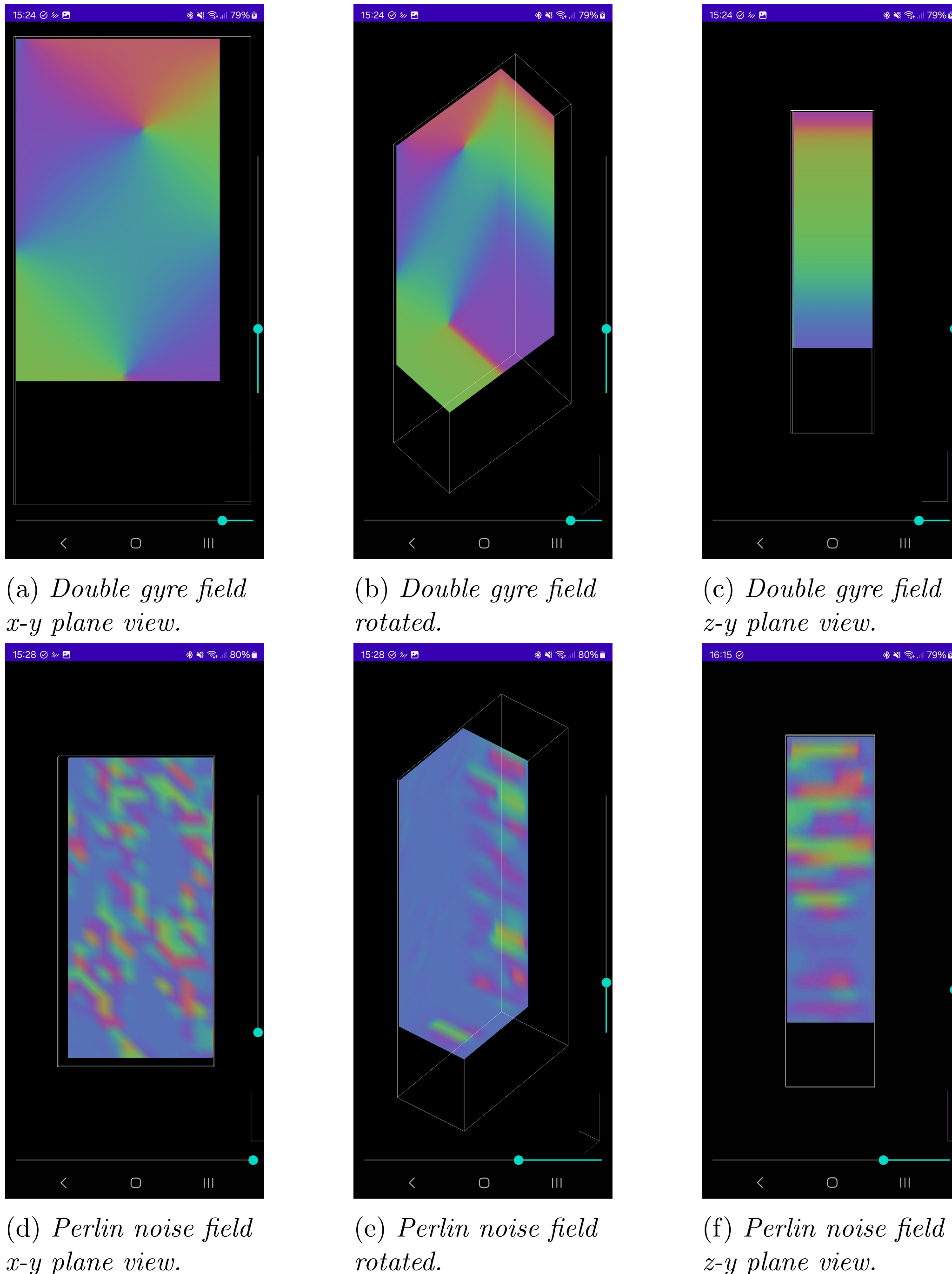
(e) *Perlin noise field rotated.*



(f) *Perlin noise field z-y plane view.*

Figure A.1: *Simple vector field visualization technique where vectors are rendered as straight lines colored based on the vector's direction.*

The visualization method depicted in Figure A.2 performs a 3D texture interpolation on the graphical fragments to render the opaque cuboid. The cuboid can be sliced using clipping planes orthogonal to the x and y axes. The position of the clipping planes can be adjusted using the sliders on the bottom and right sides of the screen.



(a) *Double gyre field x-y plane view.*

(b) *Double gyre field rotated.*

(c) *Double gyre field z-y plane view.*

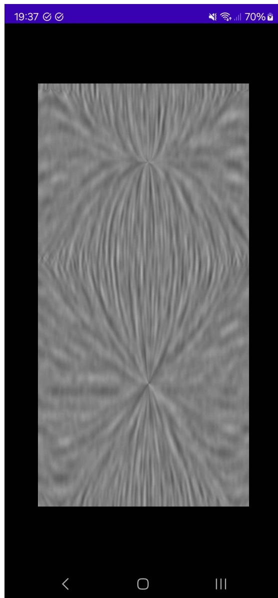
(d) *Perlin noise field x-y plane view.*

(e) *Perlin noise field rotated.*

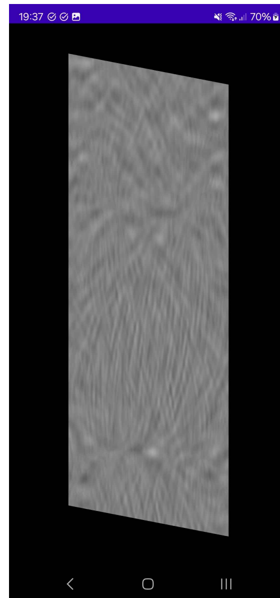
(f) *Perlin noise field z-y plane view.*

Figure A.2: *3D Texture slicing visualization technique.*

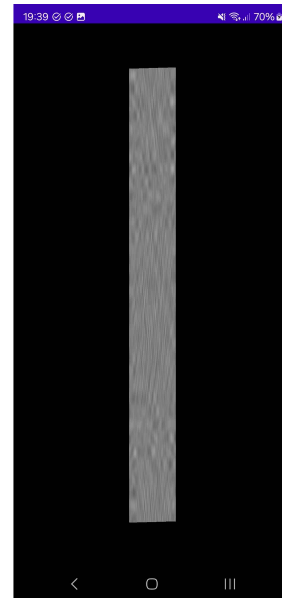
Additionally, line integral convolution visualization is shown in Figure A.3. This method integrates a grey-scale Perlin noise texture over the vector field's Eulerian vertices, resulting in the rendered images.



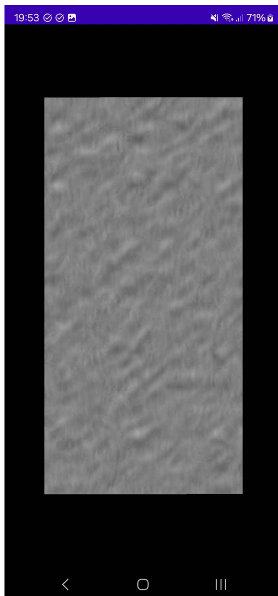
(a) *Double gyre field x-y plane view.*



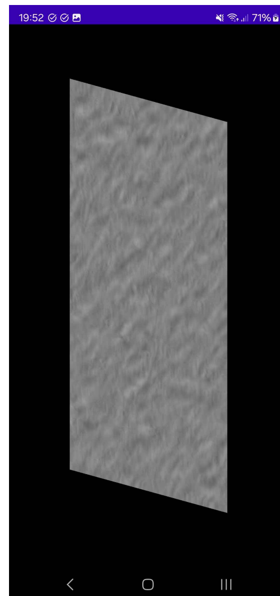
(b) *Double gyre field rotated.*



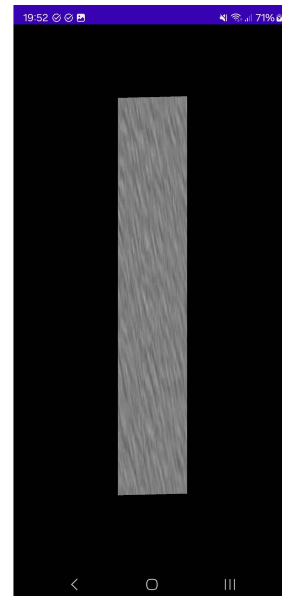
(c) *Double gyre field oblique z-y plane view.*



(d) *Perlin noise field x-y plane view.*



(e) *Perlin noise field rotated.*



(f) *Perlin noise field oblique z-y plane view.*

Figure A.3: *Line integral convolution visualization technique.*

Appendix B

Raw measured data

The data gathered during the measurements is too large to be included in this document. However, the raw data is available upon request or can be found in the measurements branch in the repository of this project: https://github.com/MartinOpat/Lagrangian-fluid-simulation-for-Android/tree/measurements

Processed data

Similarly to the raw data, the processed data can be requested or found in the measurements branch in the repository of this project: https://github.com/MartinOpat/Lagrangian-fluid-simulation-for-Android/tree/measurements

Error Analysis

This section outlines the error analysis formulas used during data processing. For further detail, see [24].

Error Propagation

For error propagation, Gauss' Law of Error Propagation was used:

$$\Delta f(x_1, x_2, ..., x_n) = \sqrt{\left(\frac{\partial f}{\partial x_1}\Delta x_1\right)^2 + \left(\frac{\partial f}{\partial x_2}\Delta x_2\right)^2 + ... + \left(\frac{\partial f}{\partial x_n}\Delta x_n\right)^2} \tag{B.1}$$

where f is the function of the variables $x_1, x_2, ..., x_n$, and $\Delta x_1, \Delta x_2, ..., \Delta x_n$ are the errors in the variables $x_1, x_2, ..., x_n$ respectively.

Standard Error in the Mean

The standard error in the mean was calculated using the following formula:

$$\text{Standard deviation}: \ \sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \mu)^2}{n}} \tag{B.2}$$

$$\text{Standard error in the mean: SE} = \frac{\sigma}{\sqrt{n}} \tag{B.3}$$

where n is the number of measurements, x_i is the i-th measurement, and μ is the arithmetic mean of all the measurements.

Glossary

Android Is a mobile operating system based on open-source software such as a modified Linux kernel version. 5, 6, 7, 12, 13, 14, 27, 29, 32, 36

GLM Is a header-only C++ mathematics library for graphics software based on the OpenGL Shading Language (GLSL) specifications. 16

JNI Is a framework that allows Java code running in the Java Virtual Machine to call and be called by native applications and libraries written in other languages such as C or C++. 13, 27

NDK Is a set of tools that allow for the use of native languages, like C or C++, in Android applications. 16

NetCDF The Network Common Data Form is a data abstraction for storing and retrieving multidimensional data. It provides a machine-independent format for representing scientific data [19]. 10, 13, 16

OpenGL API Is a cross-language, cross-platform application programming interface for rendering 2D and 3D graphics. 36

OpenGL ES Is a subset of the OpenGL API designed for embedded systems. 14, 15, 24, 26, 28, 29

smart device(s) Is a device implementing a system-on-a-chip architecture, integrating essential components into one unit, preventing component upgrades. 6, 7, 8, 9, 14, 29

SSBO Is a buffer object that allows for read and write operations in a shader program. 15, 16, 26

VAO Is an object that encapsulates the state needed to render geometry. 14, 16

VBO Is a buffer object that stores an array of data in the GPU's memory. 14, 16, 26

Acronyms

CPU Central Processing Unit. 8, 12, 15, 16, 19, 20, 24, 26, 27

fps Frames Per Second. 5, 20, 24

GPU Graphics Processing Unit. 4, 5, 12, 14, 15, 16, 17, 18, 19, 20, 22, 23, 24, 25, 26, 27, 28, 29, 36

MPM Material Point Method. 8, 9

PBF Position Based Fluids. 8, 9

PIC Particle-In-Cell. 9

RK4 Runge-Kutta 4th order. 10, 12

SIMD Single Instruction, Multiple Data. 15, 24, 26

SPH Smoothed Particle Hydrodynamics. 8, 9