# Performance Optimization and Visualization for a Fluid Dynamics Simulation

**CS351 CSE Project**

**Final Report**

**Samuel Stark**

Supervisor: Dr. Matt Leeke

Department of Computer Science

University of Warwick

May 2021

## Abstract

Using CFD programs to simulate fluids has become an incredibly important element of many research areas and industrial applications such as weather forecasting, animation, and vehicle design. Complex simulations may require large & expensive systems to complete, and even then may take multiple hours to finish. Making simulations run faster on relatively cheap GPU hardware would lower the barrier to entry, and help users work effectively by reducing their iteration times.

User efficiency may also be improved with real-time in-situ visualization, where data is visualized and displayed in parallel with the simulation. For advanced visualization features to be included, or to combine many different visualization features, an implementation must be efficient enough to not delay the simulation work.

The goal of this project is to implement a real-time tightly-coupled in-situ visualized fluid simulation. While real-time in-situ visualizations exist in games, this implementation provides novelty by focusing on accuracy over graphical fidelity. A simulation is ported to the GPU to improve performance, and optimized for the CUDA platform. The visualization is implemented from scratch with Vulkan, using techniques from the games industry for efficient rendering. This implementation is evaluated against the original simulation and other simulation/visualization programs, and the impact of an advanced visualization is shown to be negligible compared to the simulation.

*Keywords: Fluid Simulation, CUDA, Vulkan, GPU, In-Situ, Visualization*

## Acknowledgements

Many people contributed to this project's success, and there are a few in particular I'd like to thank. Firstly my project supervisor, Dr Matthew Leeke, has been a fantastic help throughout the project's development. He initially helped steer the project in the right direction, decided the title, and gave me pointers on how to research. He's always been quick to give insightful feedback, even when responding to late-night emails, and sets a high academic standard that I will strive to reach throughout the rest of my career.

Next, I'd like to thank the Second Assessor, Dr Sam Agbroko, for attending my presentation and marking this report. His questions and feedback for the presentation informed the direction of this report, especially the emphasis on novelty.

Thirdly, Dr Gavin Stark has always been willing to lend me an ear and bounce my ideas around. Finally, the rest of my friends and family have all been a great support throughout the project, for which I am very grateful.

# Contents

# List of Figures

# List of Tables

# CHAPTER 1

# Introduction

Developing equations and mathematical constructs that model natural phenomena has been a large research space for centuries. As digital computers have developed, programs have been built to use these equations and find the results much faster than previously possible[58]. Computational Fluid Dynamics (CFD) programs are programs that simulate fluid flow in some form, usually using the Navier-Stokes equations (reproduced in Eqs. (2.1.1) and (2.1.2)). These fluid simulations have a variety of uses, including in aerodynamics[1], fire spread modelling[2], and in the entertainment industry (albeit with a focus on artistic input rather than physical accuracy[3]).

If the required fluid simulations are large, in-situ visualization is an effective method. Rather than storing simulation output to huge datafiles before visualizing them, visualization is done in parallel with the simulation[4]. The rest of the simulation output does not need to be stored, reducing storage requirements significantly. In-situ methods can be described as tightly-coupled, loosely-coupled, or as a hybrid between the two. Tightly coupled visualizations share memory directly with the simulation on the same machine, and loosely coupled visualizations have independent visualization machines that receive simulation data over a network.

Both configurations reduce the required storage space, but they have separate advantages and disadvantages.

Most cases generally do not require simulations at interactive speeds, except for those found in the games industry. While the games industry does use fluid simulation[5][1], many uses do not precisely integrate the Navier-Stokes equations but approximate them using a Lagrangian method[6]. An exception to this is [7], which uses a Jacobi solver for the Navier-Stokes equations. This is used to simulate character interaction with different substances floating on the water surface[7], not to simulate large blocks of water. By and large, interactive speeds and precise simulation for large fields are not pursued together.

## 1.1 Motivation

The 2020 Advanced Computer Architecture coursework presented a fluid simulation and tasked the students with optimizing it for a 6-core Intel i5-8500 CPU[59]. The original code ran very slowly, taking 80 seconds to simulate 10 simulation-seconds. After optimizations, the code performed the same simulation in just 1.26 seconds, 64x faster than the original and 7.9x faster than real-time[60].

This original simulation purposefully limited itself in some aspects, such as iteration count for an equation solver, which prevented it from converging to an accurate solution for the test data. Students were also explicitly prevented from accelerating the simulation using a GPU, which could have made it much faster as each simulation phase is embarrassingly parallel.

Another limitation was that the simulation state could only be visualized once the full simulation had completed, instead of in real-time, even though the final simulation was fast enough. This made the results much more difficult to understand, especially for people who don't understand the underlying code or mathematics.

[1]As these methods all share the simulation and visualization memory, they are tightly-coupled in-situ visualizations.

## 1.2 Project Aims

This project has three overarching goals: to port the original simulation to the GPU, use the speedup to increase the simulation accuracy, and implement a real-time tightly-coupled in-situ visualization. The combined simulation-visualization is the core contribution of this project, referred to as "the program" throughout the rest of this report.

While real-time in-situ visualizations exist in games, where graphical fidelity is the priority, there has not been an attempt to implement one in an industrial or academic context to the researcher's knowledge. The main novelty of this project is the combination of an accurate simulation with real-time visualization methods that aim to communicate important data instead of just looking pretty.

## 1.3 Stakeholders

The main stakeholders are the researcher and the project supervisor. Both stakeholders are invested in the project due to personal interest, and in the case of the researcher the effect this project has on final year grades.

# CHAPTER 2

# Research

Research was a key element of this project. Implementing and optimizing a fluid simulation without losing correctness required a complete understanding of the mathematics the sim evaluates. From this base, optimizations were applied to speed up the program without accidentally changing the result. As GPU-based optimization is a wide-reaching problem that has been investigated before, researching optimizations that others have found effective was the first step taken before designing the program. Implementing the visualization required investigation of the current state-of-the-art visualization methods, along with specific methods for building GPU-based visualizations.

This chapter details all research that was performed while building the program. All the equations related to a simulation 'tick' are stepped through in the first section. The next section collates research into the possible optimizations for the GPU implementation, and the final section details the current status of visualizations and some techniques used in the final program.

## 2.1 An Example Simulation Tick

The 1998 book "Numerical simulation in fluid dynamics: a practical introduction"[8] defines a basic structure for a discrete simulated timestep (a.k.a. a "tick") and provides a sample guide to implementing it in Fortran or C. This was used as the base of the original simulation and continues to be the base of this project. This section will explain the general structure of the simulation as defined in [8].

The simulation described specifically simulates "*laminar* flows of *viscous, incompressible fluids*"[8] in 2D. *Laminar* flows can be treated as separate layers of particles that can slide past each other, which interact solely through friction forces. The opposite of this is Turbulent flow, where particles may move between layers due to small friction forces[8]. This adds extra viscosity (the turbulent eddy viscosity, as covered in more detail in [9]) which is much more difficult to accurately model.

*Incompressible* fluids have a uniform density across the entire flow, which greatly simplifies the calculations. This property can be assumed for low-velocity gases and most liquids[8].

*Viscous* fluids have high internal friction forces that will eventually bring a moving fluid to rest. The viscosity is controlled by a parameter known as the Reynolds number $Re$[10], which is constant over the fluid. As $Re \rightarrow 0$ the viscosity of the fluid approaches infinity, and as $Re \rightarrow \infty$ the fluid becomes *inviscid*, i.e. not viscous. Using high $Re$ this sim could be used to simulate inviscid fluids, although it is important for the fluid to still be laminar and incompressible.

Any forces acting throughout the bulk of the fluid i.e. gravity can be simulated using the $g = (g_x, g_y)$ vector. However, the 2D variant of the simulation has been used in this project for top-down simulations with a level plane, so this is left unused.

### 2.1.1 The Simulation Variables

The simulation solves for three variables: horizontal velocity $u$, vertical velocity $v$, and pressure $p$. These variables are related by the Navier-Stokes momentum and continuity equations, which can be written as follows:

$$\begin{aligned}\frac{\partial u}{\partial t} + \frac{\partial p}{\partial x} &= \frac{1}{Re}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) - \frac{\partial (u^2)}{\partial x} - \frac{\partial (uv)}{\partial y} + g_x, \\ \frac{\partial v}{\partial t} + \frac{\partial p}{\partial y} &= \frac{1}{Re}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) - \frac{\partial (uv)}{\partial x} - \frac{\partial (v^2)}{\partial y} + g_y\end{aligned} \tag{2.1.1}$$

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \tag{2.1.2}$$

The values of the simulated quantities at tick $\#n$ are represented by $u^{(n)}, v^{(n)}, p^{(n)}$. These values are discretized by evaluating them at points on a staggered grid (see Fig. 2.1.1). This grid is indexed by $i$ in the x-direction and $j$ in the y-direction. It is important to note the variables $u, v$ represent the current velocity of the fluid within each grid space, **not** the velocity of the grid cells themselves. The grid does not move at any point during the simulation.

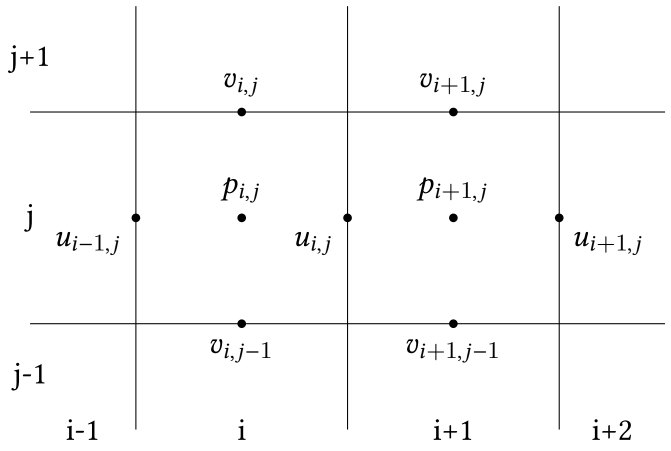


Figure 2.1.1: Discretization points for each variable on the staggered grid[8]

Each of the variables is located at a different position on the grid cell. Horizontal velocity $u_{i,j}$ is at the midpoint of the right cell edge, vertical velocity $v_{i,j}$ is at the midpoint of the top cell edge, and pressure $p_{i,j}$ is at the midpoint of the cell. This is used to solve odd-even decoupling[11]: for a fluid at rest (i.e. $u = v = 0$) the continuous solution is that the pressure $p$ is a constant across the grid. However, were this to be discretized using central differences with all variables in the same locations, it would also be possible for a checkerboard of pressure values to form, and for oscillation to take place[8]. Staggering the variables prevents this. Perić, Kessler and Scheuerer[12] show that this is also preventable through colocated grids, where a single grid is used for all variables and the velocities of each side of the cell are found using interpolation. These cell sides are implicitly staggered relative to the pressure and so avoid this problem.

To allow for derivatives to be accurately calculated for cells on the edges of the grid, boundary cells are added around each grid. The cells on the edges of any obstacles in the simulation are also marked as boundary squares. For a finite domain of size $(imax, jmax)$ this leads to a final grid size of $(imax+2)$ by $(jmax+2)$, where valid fluid values fall in the ranges $i \in \{1..imax\}$, $j \in \{1..jmax\}$.

The physical dimensions of each grid space are represented by $\delta x, \delta y$. This allows the derivatives of $u$ and $v$ to be calculated by finding the centred differences.

$$\left[\frac{\partial u}{\partial x}\right]_{i,j} := \frac{u_{i,j} - u_{i-1,j}}{\delta x}, \quad \left[\frac{\partial v}{\partial y}\right]_{i,j} := \frac{v_{i,j} - v_{i,j-1}}{\delta y} \tag{2.1.3}$$

The partial derivatives for pressure $\partial p/\partial x, \partial p/\partial y$ are found in the same way. The remaining derivatives, including second derivatives and $\partial uv/\partial x, \partial uv/\partial y$, can also be discretized by taking the difference across midpoints of their respective dimensions[13].

### 2.1.2 Overall Simulation Structure

Each simulation tick can be split into multiple stages, shown in Fig. 2.1.2. These stages are described in detail in the following sections.

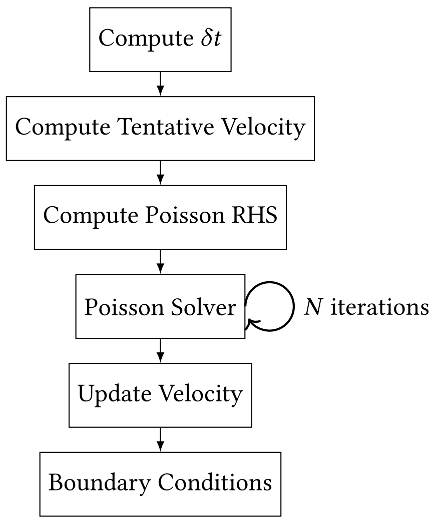


Figure 2.1.2: Stages of a Simulation Tick

### 2.1.3 Timestep Calculation

Each simulation tick simulates a discrete amount of time known as a timestep $\delta t$. This timestep is not a fixed value, and typically one would want to select as large a timestep as possible, but there are constraints on its maximum value which depend on the simulation state.

As the derivatives are calculated between adjacent grid points, it is impossible to accurately simulate a timestep where fluid moves between non-adjacent grid cells. To prevent this, the timestep $\delta t$ is calculated from the fluid velocities to make it unlikely.

$$\delta t = \tau * \min\left(\frac{Re}{2}\left(\frac{1}{\delta x^2}+\frac{1}{\delta y^2}\right)^{-1}, \frac{\delta x}{|u_{max}|}, \frac{\delta y}{|v_{max}|}\right) \tag{2.1.4}$$

Because the new velocities calculated in this tick may be larger than $u_{max}$ and $v_{max}$, the safety factor $\tau \in [0,1]$ is used to ensure the timestep is large enough to account for it[14].

### 2.1.4 Tentative Velocity

The final values of $u$ and $v$ are defined as

$$\begin{aligned}
u^{(n+1)} &= u^{(n)} + \delta t \left[ \frac{1}{Re} \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) - \frac{\partial (u^2)}{\partial x} - \frac{\partial (uv)}{\partial y} + g_x - \frac{\partial p}{\partial x} \right] \\
v^{(n+1)} &= v^{(n)} + \delta t \left[ \frac{1}{Re} \left( \frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right) - \frac{\partial (uv)}{\partial x} - \frac{\partial (v^2)}{\partial y} + g_y - \frac{\partial p}{\partial y} \right]
\end{aligned} \tag{2.1.5}$$

However, as these depend on the partial derivatives of $p$, which itself depends on velocity, they cannot be solved analytically. Variables $f$ and $g$, for horizontal and vertical "tentative velocity", are introduced to remove the dependency on $p$.

$$\begin{aligned}
f^{(n)} &:= u^{(n)} + \delta t \left[ \frac{1}{Re} \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) - \frac{\partial (u^2)}{\partial x} - \frac{\partial (uv)}{\partial y} + g_x \right] \\
g^{(n)} &:= v^{(n)} + \delta t \left[ \frac{1}{Re} \left( \frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right) - \frac{\partial (uv)}{\partial x} - \frac{\partial (v^2)}{\partial y} + g_y \right]
\end{aligned} \tag{2.1.6}$$

$$\begin{aligned}
u^{(n+1)} &= f^{(n)} - \delta t \frac{\partial p^{(n+1)}}{\partial x} \\
v^{(n+1)} &= g^{(n)} - \delta t \frac{\partial p^{(n+1)}}{\partial y}
\end{aligned} \tag{2.1.7}$$

### 2.1.5 Solving the Poisson Equation with SOR

For continuity to be achieved, the final velocity values must fulfil the continuity equation (Eq. (2.1.2)), the time discretization of which is shown below:

$$\frac{\partial u^{(n+1)}}{\partial x} + \frac{\partial v^{(n+1)}}{\partial y} = 0 \tag{2.1.8}$$

This means that the total amount of fluid entering a cell in tick $n+1$ is equal to the amount of fluid leaving, which must be the case otherwise the amount of fluid per cell wouldn't be constant and the fluid would become compressed.

Substituting the formulae in Eq. (2.1.7) into this relation and rearranging gives

$$\frac{\partial^2 p^{(n+1)}}{\partial x^2} + \frac{\partial^2 p^{(n+1)}}{\partial y^2} = \frac{1}{\delta t} \left( \frac{\partial f_{i,j}^{(n)}}{\partial x} + \frac{\partial g_{i,j}^{(n)}}{\partial y} \right) \tag{2.1.9}$$

The right-hand side of this equation is constant for timestep $n$, so can be precalculated and assigned to the variable $rhs$.

$$rhs_{i,j} := \frac{1}{\delta t}\left(\frac{\partial f_{i,j}^{(n)}}{\partial x} + \frac{\partial g_{i,j}^{(n)}}{\partial y}\right) \tag{2.1.10}$$

$$\frac{\partial^2 p^{(n+1)}}{\partial x^2} + \frac{\partial^2 p^{(n+1)}}{\partial y^2} = rhs_{i,j} \tag{2.1.11}$$

Discretizing this gives

$$\frac{p_{i+1,j}^{(n+1)} - 2p_{i,j}^{(n+1)} + p_{i-1,j}^{(n+1)}}{(\delta x)^2} + \frac{p_{i,j+1}^{(n+1)} - 2p_{i,j}^{(n+1)} + p_{i,j-1}^{(n+1)}}{(\delta y)^2} = rhs_{i,j} \tag{2.1.12}$$

and taking the simplest boundary conditions[8]

$$p_{0,j} = p_{1,j}, \qquad p_{i_{max}+1,j} = p_{i_{max},j} \qquad j \in \{1..j_{max}\} \tag{2.1.13}$$

$$p_{i,0} = p_{i,1}, \qquad p_{i,j_{max+1}} = p_{i,j_{max}} \qquad i \in \{1..i_{max}\} \tag{2.1.14}$$

$$f_{0,j} = u_{0,j}, \qquad f_{i_{max},j} = u_{i_{max},j} \qquad j \in \{1..j_{max}\} \tag{2.1.15}$$

$$g_{i,0} = v_{i,0}, \qquad g_{i,j_{max}} = v_{i,j_{max}} \qquad i \in \{1..i_{max}\} \tag{2.1.16}$$

resolves the equation to:

$$\frac{\epsilon_{i,j}^{E}(p_{i+1,j}^{(n+1)} - p_{i,j}^{(n+1)}) - \epsilon_{i,j}^{W}(p_{i,j}^{(n+1)} - p_{i-1,j}^{(n+1)})}{(\delta x)^2} + \frac{\epsilon_{i,j}^{N}(p_{i,j+1}^{(n+1)} - p_{i,j}^{(n+1)}) - \epsilon_{i,j}^{S}(p_{i,j}^{(n+1)} - p_{i,j-1}^{(n+1)})}{(\delta y)^2} = rhs_{i,j} \tag{2.1.17}$$

where $\epsilon_{i,j}^{\{N,S,E,W\}}$ represents the boundary squares. The equation shown on page 11 is for North, but the definition extends to other directions.

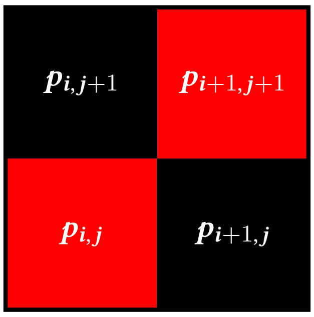


Figure 2.1.3: Example checkerboard pattern used for red/black splitting

$$\epsilon^N_{i,j} = \begin{cases} 0 & \text{The square directly north of } i,j \text{ is a boundary} \\ 1 & \text{The square directly north of } i,j \text{ is } \textit{not} \text{ a boundary} \end{cases} \tag{2.1.18}$$

Over the whole grid, this results in a linear system of equations over the inputs $p_{i,j} \quad \forall \quad i \in \{1..i_{max}\}, j \in \{1..j_{max}\}$. These can be decoupled by partitioning $p$ into red and black squares with a checkerboard pattern (see Fig. 2.1.3). As each cell's equation only depends on the adjacent values, iterations of Successive Over-Relaxation (SOR) can be performed on red and black in turn to reach a final value[1][16]:

$$\beta_{i,j} := \frac{\omega}{\left(\frac{\epsilon^E_{i,j}+\epsilon^W_{i,j}}{(\delta x)^2} + \frac{\epsilon^N_{i,j}+\epsilon^S_{i,j}}{(\delta y)^2}\right)} \tag{2.1.19}$$

$$p^{it+1}_{i,j} := (1-\omega)p^{it}_{i,j} + \beta_{i,j} * \left(\frac{\epsilon^E_{i,j}p^{it}_{i+1,j} + \epsilon^W_{i,j}p^{it}_{i-1,j}}{(\delta x)^2} + \frac{\epsilon^N_{i,j}p^{it}_{i,j+1} + \epsilon^W_{i,j}p^{it}_{i,j-1}}{(\delta y)^2} - rhs_{i,j}\right) \tag{2.1.20}$$

These iterations are continued until the L2 norm[17] of the residuals (the difference between the left-hand side as calculated and the expected right-hand side of Eq. (2.1.20) for each cell) falls below a specific tolerance [2] [8].

[1]This could equally be done without partitioning $p$, but the partitioning splits the SOR into separate phases which can then be parallelized. Normal SOR cannot be parallelized[15].

[2]In the original simulation this tolerance was relative to the L2 norm of $p$, although this was not directly specified by the book.

### 2.1.6 Final Velocity Calculations

Once the final values of $p$ have been calculated the velocity values $u, v$ can be found with Eq. (2.1.7). The boundary conditions for velocity must then be applied. There are four relevant types of boundary condition[3], which are applied depending on the type of boundary.

1. No-Slip condition - no fluid penetrates the boundary, and fluid does not move past it i.e. the boundary applies friction.
2. Free-Slip condition - fluid may not penetrate the boundary, but no friction is applied. Only tangential velocity is preserved for adjacent fluids.
3. Inflow - fluid is flowing in constantly, so the velocity is set to a constant value.
4. Outflow - velocity perpendicular to the surface is preserved and fluids may flow out.

[3]The book specifies five, including a Periodic Boundary Condition, which the original simulation did not support.

## 2.2 Optimization

Optimizing simulations is important in all cases, even those that are not real-time, as it allows the engineers using the software to iterate faster on their designs. When the extra constraint of real-time speeds is added, it becomes even more important. This section explores how CFD simulations started moving to the GPU, the previous optimizations applied at the CPU level , and the new optimizations to be applied to the GPU version.

### 2.2.1 Background

One of the first papers on optimizing a CFD simulation was released in 1995[18]. This paper considered the effect of automatic compiler parallelization and optimization of a full CFD program, and the steps a programmer must take to guide the compiler to e.g. avoid false sharing. The program was only executed on the CPU, as General Purpose GPU computing (GPGPU) had not yet taken hold.

GPGPU was first used for CFD simulations in 2004 with [19]. This used the “fragment shading” stage of the GPU rendering pipeline to perform the computation, as standalone “compute” pipelines were only exposed by APIs from 2007 onwards. Such APIs include CUDA (2007)[41], OpenCL (2008)[54], DirectX’s DirectCompute (2009)[20], and OpenGL 4’s compute shaders (2012)[55].

Since 2007, using GPGPU for CFD has become a large topic of study, as investigated in detail by [21]. We will take advantage of this to optimize the simulation speed to the point that a simulation can be both performed and visualized in real-time, which many other programs do not achieve.

### 2.2.2 Previous Work

As work on CFD progressed some optimizations were developed that change the simulation pipeline and provide an overall speedup. Some of these were adapted into the original CPU simulation[60] and carry over into the CUDA version.

Given the definition of $\beta$ in Eq. (2.1.19), the value of $\beta_{i,j}$ does not change over the course of the simulation and so can be precalculated before the simulation starts. Additionally, if it can be guaranteed that for every boundary square $p = 0$, which can be done either by never updating their pressure values or by updating them with $\beta_{i,j} = 0$, then $\epsilon_{i,j}$ doesn't need to be evaluated during the simulation at all. These optimizations increased the runtime speed of the Poisson evaluation by 2.24x[4], and they have been kept in the CUDA program.

The book states an alternate solution where $\epsilon$ is set to 1 at all times and pressure values on boundaries are copied from adjacent fluid squares[8]. Using this method may prevent noncontinuous starting velocities from producing nonphysical pressure values. Due to timing constraints this was not implemented for the simulation, but could be done in the future.

As stated in Section 2.1.5, red/black SOR is used to iteratively solve the Poisson equation. In the initial CPU simulation the values $(f, g, p, rhs)$ for red and black data were stored in the same arrays. This was problematic as data of the same colour was never contiguous, and any iteration looking for just red values would get a cache line with both colours, leading to half of each cache line being wasted. To fix this, red and black data is split into separate arrays before starting the Poisson solver. This has been carried over into the CUDA implementation.

The CPU simulation used OpenMP[61] to automatically parallelize the Poisson solver (and other program elements) by column. That is, each thread was given a group of columns to process. This was not needed in the CUDA version as each GPU kernel is implicitly parallelized over many GPU threads.

The CPU simulation included optimizations exploiting properties of the original code, such as floating-point precision, to speed up calculations while producing identical results. These optimizations include using fused multiply-add[22] in some places (but not all), precalculating divisions with double-precision floats, and skipping the residual calculation phase altogether. As this project was focused on im-

[4]The $\beta$ precalculation increased speed by 1.4x, and the removal of $\epsilon$ increased speed by 1.6x.[60]

proving upon the accuracy and speed of the original simulation, instead of producing bit-identical results, some of these optimizations were removed or made redundant.

### 2.2.3 New Optimizations

CPUs and GPUs have very distinct designs, so when moving programs between them it's important to acknowledge optimization techniques that apply to one and not the other. This section sums up the research into optimizing CUDA programs, which were applied when designing/implementing the final program.

CUDA devices combine groups of 32 threads into a 'warp' and execute them concurrently[41]. If the threads in a warp attempt to access multiple words in the same cache line, the access is *coalesced*[42] and only one cache line needs to be fetched for the warp to continue. Otherwise, if the accesses all touch different cache lines, every cache line needs to be fetched before execution can continue for any of the threads. This was accounted for when structuring GPU work.

The CUDA C Programming Guide[43] states that read-only memory can be read into a special data cache using the `__ldg()` intrinsic. The compiler may insert this automatically if it detects that data *must* be read-only, which is preferable to inserting it manually. Applying the `const` and `__restrict__` qualifiers on pointers to read-only data acts as a strong hint for the compiler to add the intrinsic. Diarra[23] found that introducing these qualifiers where possible led to large speedups in pointer heavy applications, and while our case may not use many pointers this should still be implemented wherever possible. This optimization was verified by checking for the presence of `ld.global`[44] instructions in the PTX assembly of compiled GPU kernels.

The CPU simulation used Intel AVX and SSE instructions[62] to calculate four Poisson values at once[5]. At the start of the project it was believed that CUDA cores could use SIMD on four-element floating-point vectors, and that this could

[5]Vectors of eight were tried but were found to be slower than four.

be exploited similarly, but this is not the case. The only SIMD instructions CUDA allows are for integer vectors of 2x16-bit or 4x8-bit[45]. As the simulation operates on 32-bit signed floating-point, these instructions are not suitable.

Calculating the simulation timestep and calculating the residual for a Poisson iteration both require a reduction over large blocks of data. Highly parallel GPU optimizations have already been studied extensively, so it is trivial to implement a fast generic reduction kernel. In [24] seven kernels are described, in ascending order of speed, and the second kernel was used in the CUDA simulation.

CUDA Graphs[46] are a CUDA feature that reduce CPU overhead by invoking a large amount of GPU work all at once, rather than with individual invocations. Running a CUDA graph will always use the same arguments as when initially recorded, so any work that depends on data that changes quickly (such as a timestep) might not be suitable. Profiling was used to determine when CUDA graphs are suitable, then applied only in these cases to avoid overcomplicating the program.

### 2.2.4 Conclusion

The original CPU simulation used both algorithmic optimizations (such as precomputing $\beta$) and CPU architecture-specific optimizations (multithreading, floating-point tricks, and vectorization). Algorithmic optimizations map well to CUDA unlike some architecture-specific optimizations, but CUDA makes up for these with its own suite of optimizations. As with the original CPU simulation, profiling and inspecting the compiled program was required to check the optimizations provided the expected benefits.

## 2.3 Visualization

To develop a visualization with similar computational intensity to the current state-of-the-art, the current state-of-the-art must first be investigated. This section covers some previous visualizations, including the capabilities built-in to the original program, and shows the visual techniques used in Autodesk CFD. The common practice of combining different techniques is investigated, and an algorithm for massively parallel particle simulation is shown.

### 2.3.1 Background

One of the earliest CFD interactive visualizations was in 2002, which had a simulation running slower than real-time on a separate computer to the real-time visualization[25]. Decoupling the simulation speed from the visualization speed allowed for high framerates to be achieved for the user interface, but any changes made from the user interface had a delay of 0.5 seconds before being reflected in the simulation. This qualifies as a loosely-coupled in-situ visualization because simulation data is streamed to a separate visualization system while the simulation completes.

VTK is a significant open-source toolkit, first introduced in 1993[26], which powers multiple tools such as ParaView and VisIt. Both of these programs support tightly-coupled in-situ visualization of an external simulation with plugins, but the VTK base is not well suited to in-situ integration[4].

Autodesk CFD is a closed-source tool that integrates a simulation with a standard visualization. It's based on ALGOR FEA, created by ALGOR Inc which was acquired by Autodesk in 2008[63]. The latest iteration is targeted at the manufacturing industry, unlike VTK, and does not support in-situ visualization.

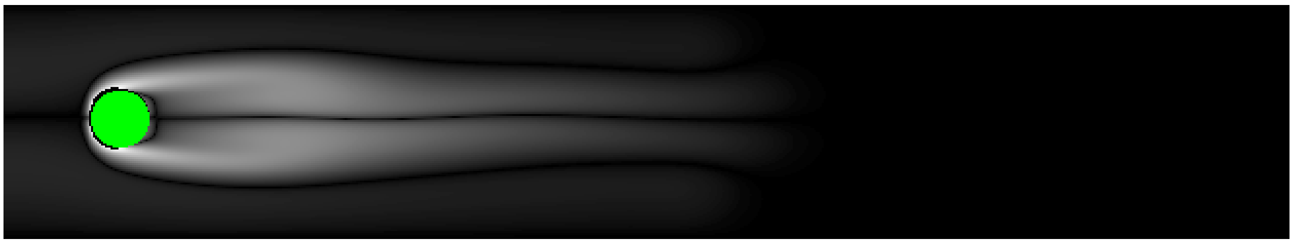
Both of the above examples are not built for in-situ visualization, but these programs still represent the industry standard for visualization capabilities. A novel element of this project is building a program from the ground up for in-situ visualization, and the visualization components will be based on Autodesk.

### 2.3.2 Previous Work

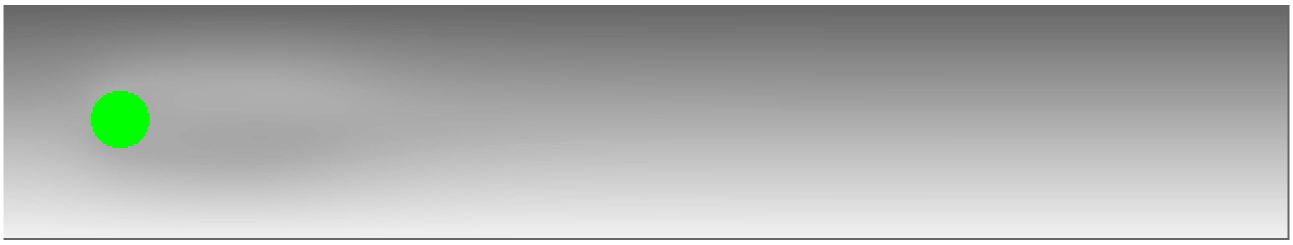
The original simulation[59] included a simple image visualizer for a static state, which evaluated one of two quantities over the grid and produced a `.ppm` image with the result. These quantities were Vorticity ($\zeta$), the strength of vortical (a.k.a. rotational) motion at each point in the grid; and Stream Function ($\psi$), the contours of which define streamlines. Streamlines are lines that are parallel to the velocity vector at each point, allowing the long-term flow of particles to be represented with a single line, and thus in a static image[64]. The quantities are defined by Eqs. (2.3.1) and (2.3.2), as specified in [8]. Examples of these modes are shown in Fig. 2.3.1.

$$\zeta(x, y) := \frac{\delta u}{\delta y} - \frac{\delta v}{\delta x} \tag{2.3.1}$$

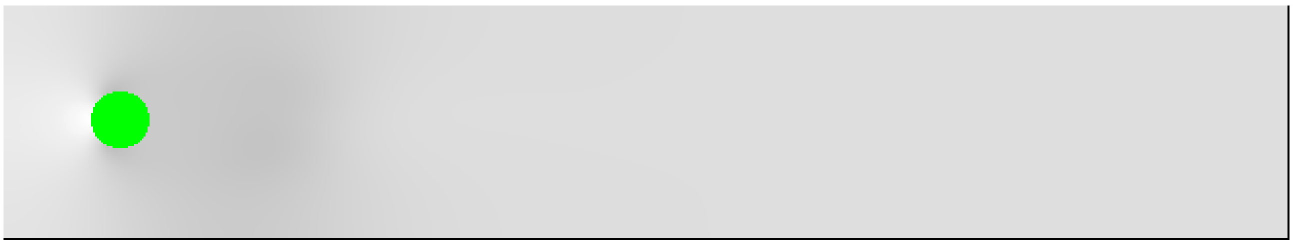
$$\frac{\delta \psi(x, y)}{\delta x} := -v, \qquad \frac{\delta \psi(x, y)}{\delta y} := u \tag{2.3.2}$$



(a) Vorticity $\zeta$



(b) Stream Function $\psi$



(c) Pressure $p$

Figure 2.3.1: Examples of the three outputs available from the original visualization

The vorticity image in Fig. 2.3.1a competently shows which areas of the grid contain particle movement. Unfortunately, near the edges of the obstacle circle (shown in green) the edges are black, implying no movement or rotation, which is incorrect and also a distracting artefact for the viewer. These are due to the imprecise nature of the original code, which only uses the differences to the East and South to find $\zeta$. This breaks down when the squares in these directions are boundaries, and the program defaults to zero. A better solution would be to take the central difference whenever possible and to fall back to using only one side when adjacent to a boundary. This would mean the only points where this breaks down are where a square is surrounded by boundaries on opposite sides, which is very unlikely and would also likely break other areas of the simulation.

The Stream Function visualization (Fig. 2.3.1b) is nearly impossible to visually parse, which makes sense as the velocity information is encoded in the differences between adjacent squares and not directly in the colours. The Stream Function is not intended to be directly visualized but instead used to find streamlines, which can be visualized directly.

During program development, a third mode was added which directly visualized the pressure values to aid in debugging, but this was not a very useful visualization as seen in Fig. 2.3.1c. Pressure is only ever referenced in the Navier-Stokes equation (and subsequently the algorithm) as a relative value. However, the simulation in practice ends up increasing all cells by a small amount each iteration. This overall increase in pressure values is ignored by the simulation, but the visualization doesn't adjust for it. In this example, the pressure values have all increased so even the lowest pressure value is a mid-grey. If the program simulated for too long, the pressure values would become too high and the visualization would be entirely white. This should be accounted for in the visualization, but also implies the base simulation is unstable. The simulation stability will be touched on in the Results and Evaluation (Chapters 9 and 10).

### 2.3.3 Current State-of-the-Art

The nature of this project required the visualizations to be rebuilt from the ground up, both to work on the GPU correctly and to synchronize with the concurrent simulation. This allowed new visualization features to be added. As a starting point, the supported visualization features of Autodesk CFD 2019 were investigated, and are outlined in Table 2.3.1. This section will explain these features in more detail.

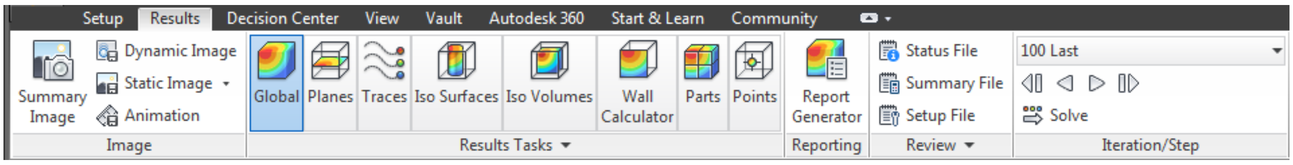


Figure 2.3.2: Results Ribbon for Autodesk CFD 2019[65, Results Visualization]

| Tool | Type |
|---|---|
| Global Controls | Visual |
| Planes | Visual |
| Iso Surfaces | Visual |
| Iso Volumes | Visual |
| Particle Traces | Visual |
| Wall Calculator | Text |
| Parts | Text |
| Points | Text |

Table 2.3.1: Summary of results tools from Autodesk CFD

Autodesk CFD is a 3D simulation and visualization program, typically involving one or more 3D surfaces (a 'model') and simulating the fluid movement around these surfaces. The simulation process creates multiple quantities, including scalar quantities Speed, Temperature, Pressure; and Vector quantities such as Velocity.

**Global Controls**[65, Global Controls] visualize selected quantities on all model surfaces. A scalar quantity can be displayed by changing the surface colour according to a scale (Fig. 2.3.3). A vector quantity can be displayed by creating small arrows on the surfaces that represent the quantity's direction and magnitude. The range of displayed values can be changed for both the scalar and vector quantities.

A key limitation of this method is it only shows the quantities on the surface, not

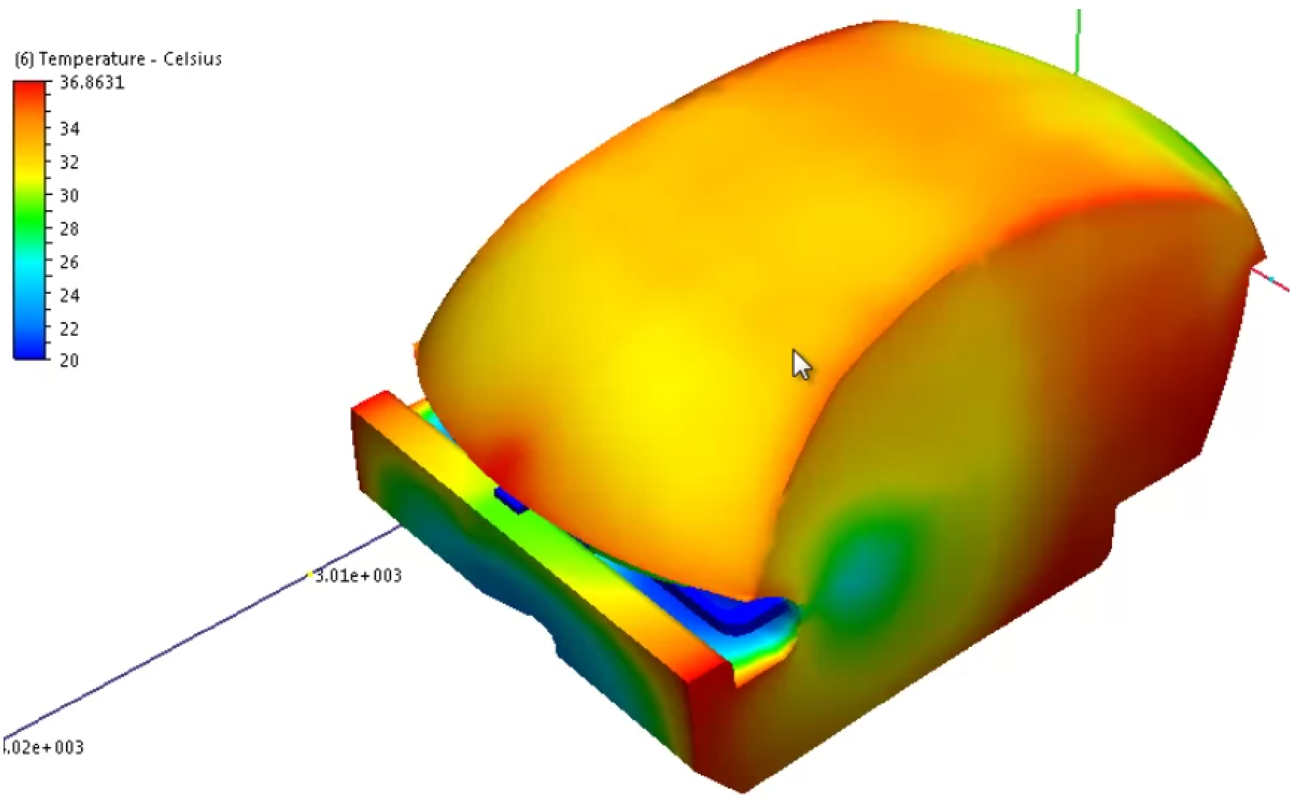


Figure 2.3.3: Using the Global Controls to visualize a scalar result[66]

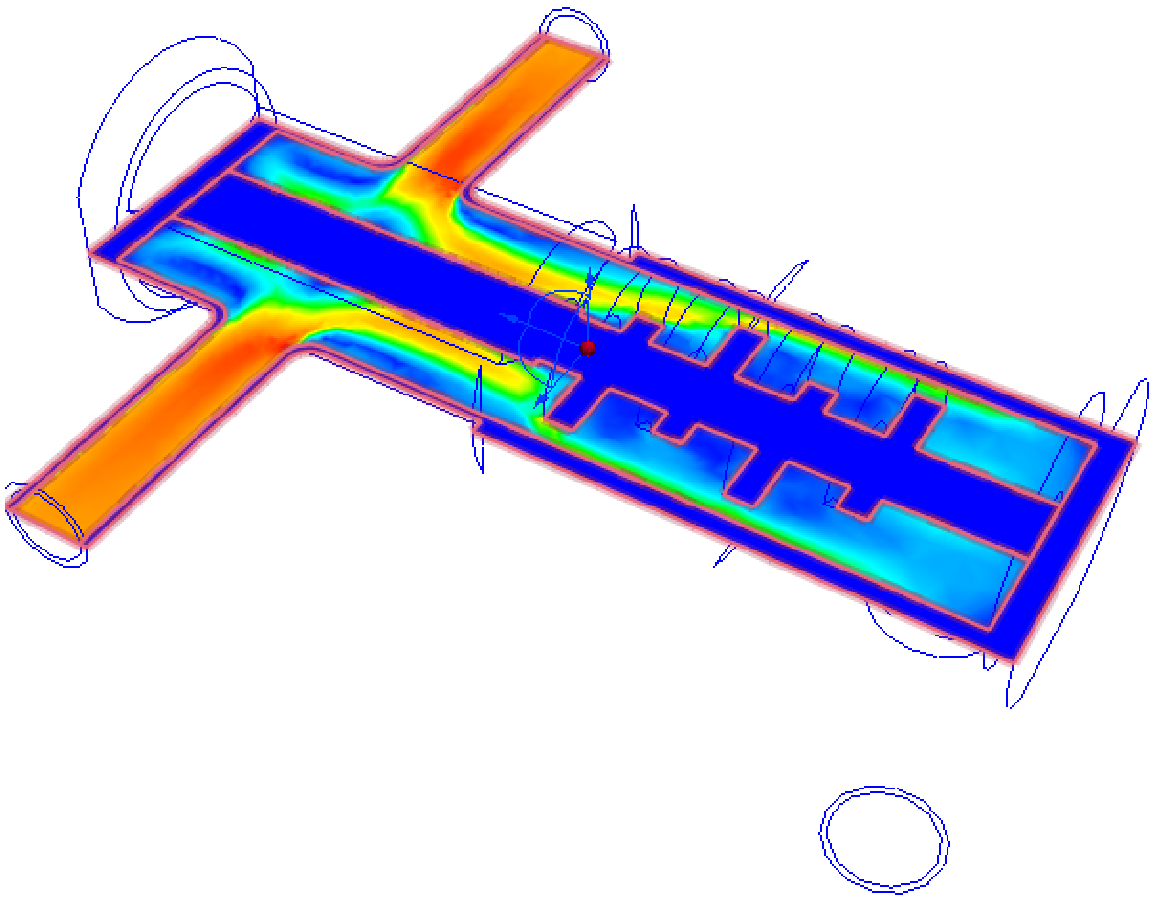
the quantities inside the fluid. Result Planes, Iso Surfaces, and Iso Volumes address this by calculating new surfaces/volumes which visualize quantities.

**Results Planes**[65, Planes] define a flat cross-section of the model which can visualize a single scalar quantity and a single vector quantity (Fig. 2.3.4).

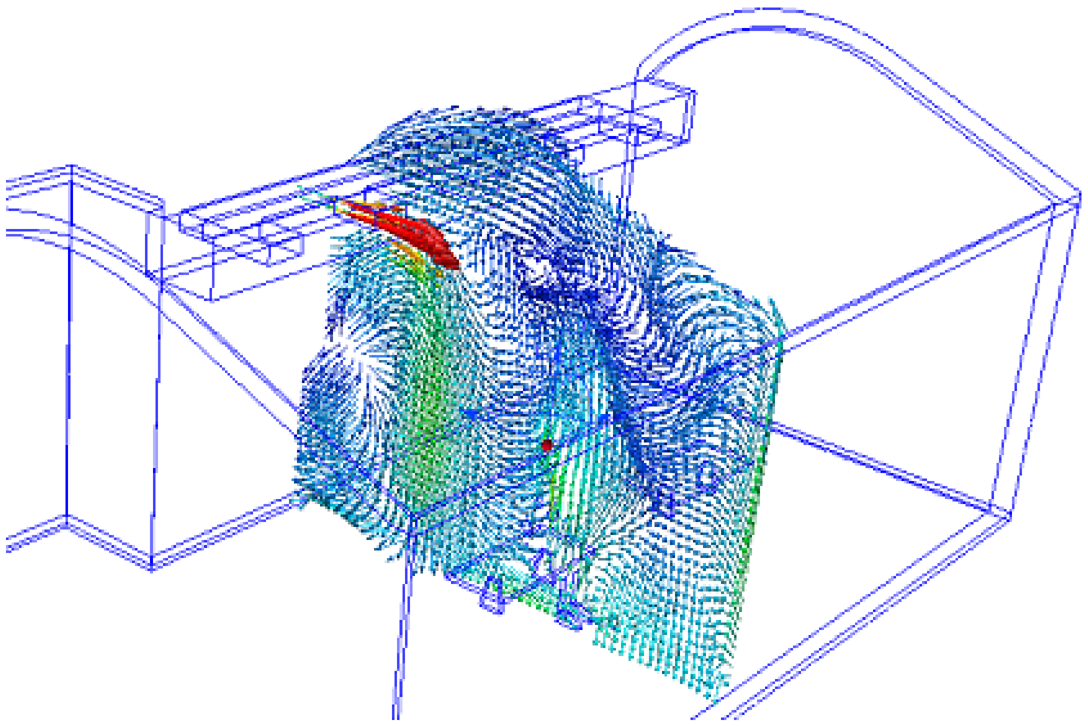
**Iso surfaces** define a surface based on the value of a result scalar quantity, e.g. $T = 5\,^{\circ}\mathrm{C}$. This surface can then display a separate scalar and vector quantity, just like a Result Plane (Fig. 2.3.5).

**Iso volumes** are similar to an isosurface but define a volume based on a value range ($0\,^{\circ}\mathrm{C} \leq T \leq 5\,^{\circ}\mathrm{C}$). A vector quantity can be displayed at points within the volume, and the surfaces of the volume can show a scalar quantity.

**Particle Traces** show the path particles would take through the flow after being emitted at certain points (or 'seeds'). Autodesk allows these paths to be traced forwards or backwards, allows particles to be simulated both with mass and without, and allows a variety of particle trails to be shown (Fig. 2.3.6).



(a) Scalar Quantity



(b) Vector Quantity

Figure 2.3.4: Result Planes displaying different types of Quantity

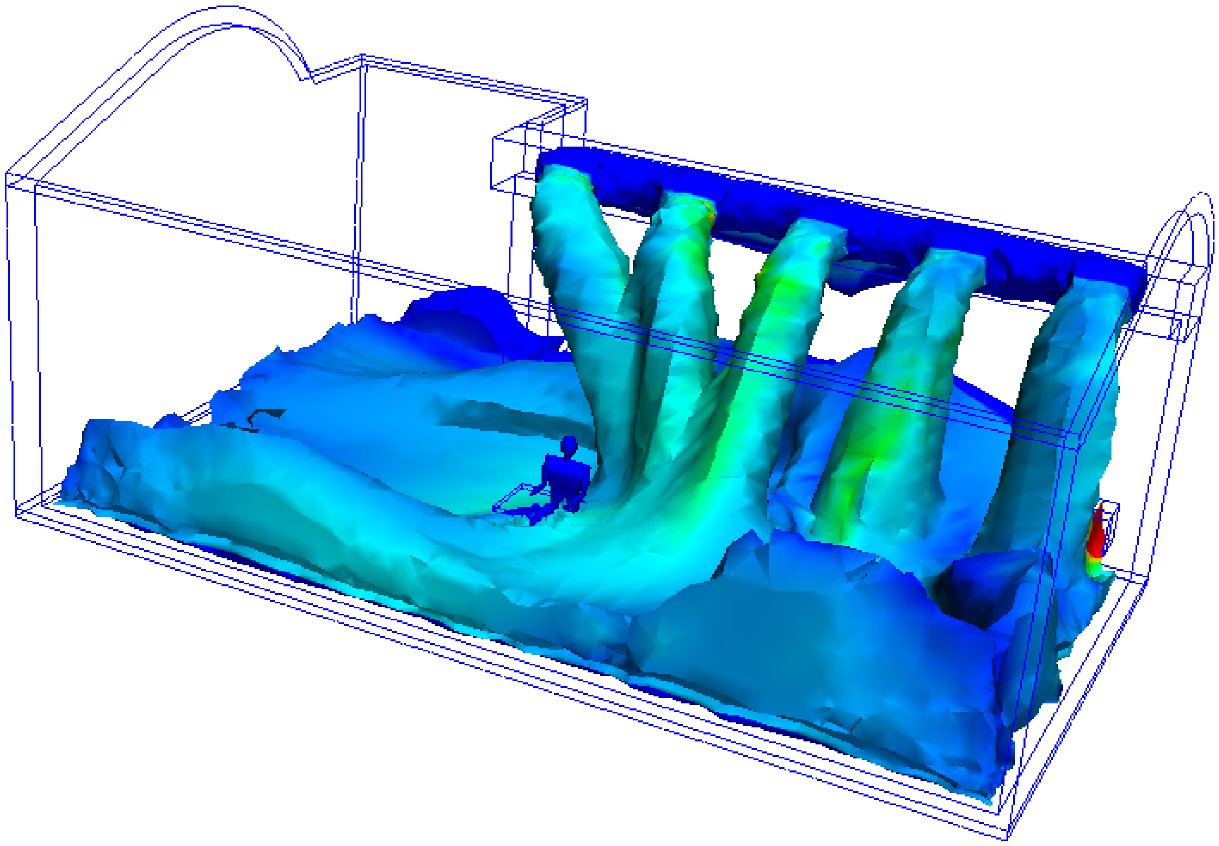
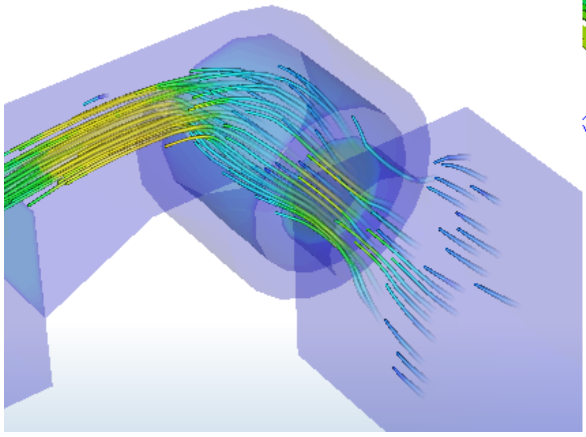
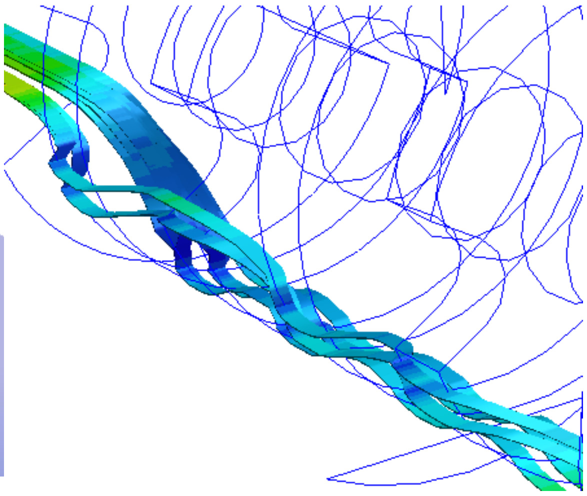


Figure 2.3.5: Example of an Isosurface, defined by a velocity magnitude and displaying static pressure.


(a) Particle Comets


(b) Particle Ribbons

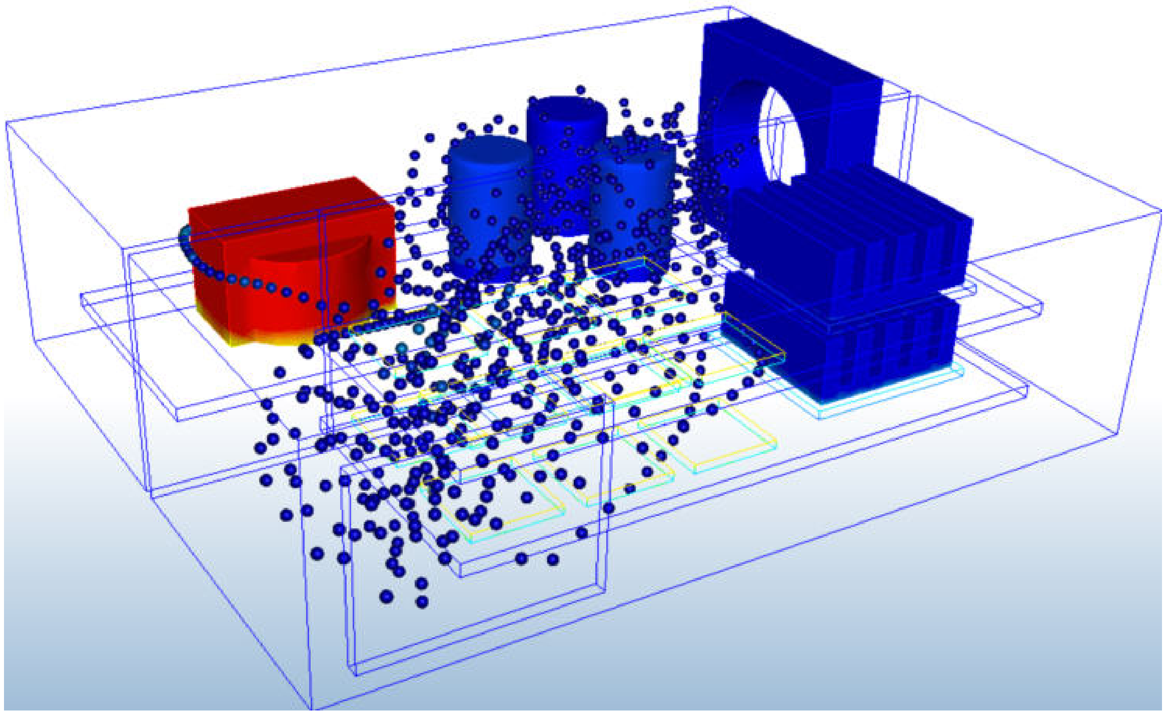

(c) Particle Spheres

Figure 2.3.6: Examples of Autodesk particle trails.

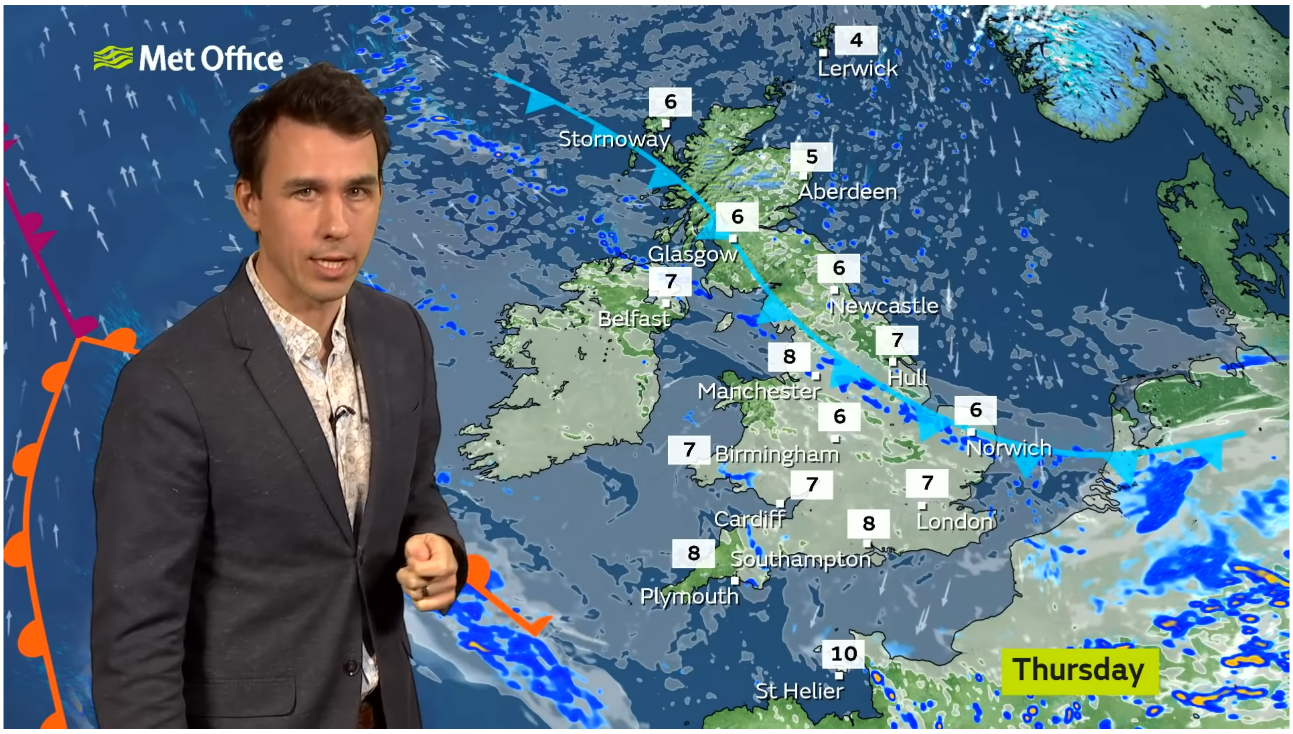


Figure 2.3.7: MET Office weather report[67]

### 2.3.4 Stagnation & Composition

The techniques described above have been well known for at least 20 years. The most prevalent algorithm for computing iso-surfaces was published in 1987[27]. Particle Traces and Results Planes were used in 1999[28], although this is not the origin of either technique. In some cases, there has been extensive research into optimizing visualization techniques[29], and in domain-specific areas there is recent academic research into creating new techniques[30], but research into new generic visualization techniques has stagnated. This stagnation was noted in [31] from 2004, which concluded the primary challenges facing visualization were "identifying and characterizing features" to visualize rather than developing new techniques. The MET Office's weather reports demonstrate this principle, and show that composing multiple techniques together can increase information density while remaining easy to understand.

Fig. 2.3.7 shows at least three visualization techniques used in the same image, each of which has been filtered to relevant points.

1. The large blue, orange, and mauve lines show weather 'fronts' moving in, the blue line particularly shows a cold front moving in from the north-east.
2. The arrows show the wind direction, and only appear in areas with high wind speeds. In the video they also move, making it easier to understand at a glance.
3. The gray and dark blue shape overlays show cloud cover and rain, respectively.

This proves the potential of combining multiple techniques, which was taken into account when building the program.

### 2.3.5 Realtime Particle Simulation Techniques

The particle simulation element of our visualization doesn't affect the simulation content, and does not have to be completely accurate as long as the flow is approximately correct for the viewer. That is, the particle movement should fulfil Fig. 2.3.8[32] for the following variables:

- $p =$ particle position.
- $t =$ time, and $t \in [t_1, t_n]$where$n =$ number of timesteps.
- $\vec{V}(p, t) =$ fluid velocity at point $p$ and time $t$.

$$\frac{dp}{dt} = \vec{V}(p, t)$$

Figure 2.3.8: Equation for particle movement in unsteady flow

The unsteady-flow variant is shown because the fluid and particles will be moving at the same time, so the fluid itself is unsteady.

A common numerical method for accurately integrating this is the second-order Runge-Kutta scheme with adaptive timesteps, described in [32]. This takes a constant step size $0 < c \leq 1$ which can be changed to control accuracy.

$$h = c\left\|\vec{V}(p_k, t)\right\|, \qquad p^* = p_k + h\vec{V}(p_k, t),$$
$$p_{k+1} = p_k + h\frac{(\vec{V}(p_k, t) + \vec{V}(p^*, t+h))}{2},$$
$$t = t + h, \qquad k = k + 1 \quad (2.3.3)$$

A key issue with implementing this method in the program is that $\vec{V}$ is only kept in memory for one value of $t$, so interpolating between $\vec{V}(p_k, t)$ and $\vec{V}(p^*, t+h)$ is impossible. This also requires an indeterminate amount of steps, and potentially a different amount of steps for each particle, which is not GPU friendly. Instead, a simpler version was chosen based on the steady-state variant. At the instant the particle is simulated, the fluid is modelled as steady, and the particle moves with the same timestep taken by the simulation. To avoid particles stepping over cells entirely, the timestep is subdivided into 4 iterations.

$$\forall i \in [1..4] \qquad p_i = p_{i-1} + \vec{V}(p_{i-1}, t)/4 \quad (2.3.4)$$

If $p$ does not align with a grid space, $\vec{V}(p)$ is chosen using trilinear interpolation between the closest grid cells as specified in [32].

In addition to the particle simulation, particles must also be spawned and deleted when necessary. GPU-based particle simulations are standard in video games, which need to run at high frame rates just like our visualization, so their techniques are a natural fit. Turánszki[68] proposed a three-phase model:

1. The kickoff phase, which determines the amount of new particles to create.
2. The emission phase, which adds the required amount of new particles to the set by pulling from a queue of inactive particles.
3. The simulation phase, which updates particle positions, adds dead particles to the inactive queue, and adds alive particles to a render queue.

This approach has been used in the final visualization.

### 2.3.6 Conclusions

The research outlined so far has established common visualization methods, the potential for composing them together, and a design for an efficient particle simulation. This was used to design the visualization (Chapter 4) and informed the implementation (Fig. 6.5.4).

# CHAPTER 3

# Ethical, Social, and Legal Issues

As predicted in the Specification and Progress Report, there have been no ethical or social issues with the development of this simulation and visualization. The simulation has been derived from code provided to the students for a coursework[59], which itself is directly derived from a book[8] available at the Warwick Library. The visualization is entirely original code, and the inspirations and research used to design it have all been cited.

To ensure the work can be trusted, and to maintain professional standards, the BCS Code of Conduct[69] has been followed. Professional standards were maintained during development, and research performed has been effectively referenced to a high standard.

# CHAPTER 4

# Project Requirements

Ahead of initial development, a set of requirements were created to further specify the project goals. These requirements evolved as more research was completed, for example, the visualization-related requirements were only determined after visualization research completed (see Chapter 7). This chapter shows the functional and non-functional requirements, prioritized as either **must**-have or **should**-have. Complex requirements have sub-requirements, which clarify certain features that must/should be present for the top-level requirement to be met. The hardware and software constraints for the program are also shown.

## 4.1 Functional Requirements

Functional Requirements define actions a program must/should be able to perform.

**F1** The system **must** store simulation state in a file or set of files.

**F2** The system **must** be able to load the initial state of a simulation from these file(s).

**F3** The system **must** be able to generate initial simulation state files.

**F4** The system **must** be able to simulate from an initial state for a set amount of time without visualizing.

- **F4.1** This mode **must** be able to store the final state to output file(s).

**F5** The system **must** be able to simulate from an initial state for an indeterminate amount of time while visualizing.

- **F5.1** This mode **must** allow the user to pause and resume the simulation.
- **F5.2** This mode **should** be able to save it's state to output file(s) when requested.
- **F5.3** This mode **should** allow the user to manipulate the simulation or visualization state while simulating.
- **F5.4** This mode **should** be able to run at a locked frame-rate.
- **F5.5** This mode **should** be able to run as fast as possible, without locking the framerate.
- **F5.6** This mode **must** be able to perform at least one of Reqs. F5.4 and F5.5.

**F6** Both methods of simulation **must** be capable of using the GPU for simulating.

**F7** The system **must** be able to compare how similar two simulation states are.

- **F7.1** This comparison **should** produce a binary SIMILAR/NOT SIMILAR verdict using heuristics.

**F8** The visualization **must** consist of multiple layers which can be individually controlled.

- **F8.1** The visualization **must** always display a background layer which shows the simulation obstacles in a different color to the fluid.
- **F8.2** The visualization **must** be able to display an optional scalar quantity (e.g. pressure) using a color scale, where the value is within a user-defined range.
- **F8.3** The visualization **must** be able to display an optional vector quantity (e.g. velocity) using a vector field, where the magnitude is within a user-defined range.
- **F8.4** The visualization **must** feature an optional particle simulation, where particles are continuously emitted and move with the velocity of the field.

**F9** Range-based quantities (Reqs. F8.2 and F8.3) **should** have an auto-range function to automatically calculate the range based on the values present.

**F10** All colors used in the visualization (e.g. particle colors, the scalar color scale) **should** be user-controlled.

**F11** The locations of particle emitters **should** be user-controllable.

## 4.2 Non-Functional Requirements

Non-functional Requirements do not define actions, but rather define properties of those actions or the program that must be met.

**NF1** The system **must** be capable of operating on large datasets (e.g. 4096x4096 grids) without failing.

**NF2** The system **must** be efficient and avoid wasting any resources allocated to it.

**NF3** The simulation **must** produce similar results to the original simulation when equivalent initial state is used.

**NF4** The simulation **should** run at least 2x as fast as the original simulation when equivalent initial state is used.

**NF5** The visualized simulation **must** run in real-time at framerates $\geq$ 30 FPS for some outputs.

**NF6** The visualization features **should not** have a significant impact on the framerate.

**NF7** The visualized simulation **should** intuitively represent the fluid flow such that it can be understood by someone unfamiliar with fluid simulation.

**NF8** The particle simulation (Req. F8.4) **should** demonstrate advanced behaviour to make the visualization more intuitive e.g. avoiding clumping.

**NF9** The system **must** be fully documented and maintainable.

**NF10** The system **should** have a simple guide to common operations for new users to refer to.

**NF11** The system **should** be fully compilable and executable from a DCS machine with minimal extra installations.

## 4.3 Hardware and Software Constraints

As this simulation uses a GPU, the developer must have one available for debugging and testing the program. As the CUDA API is used to implement the simulation (see Section 6.1.3), the program requires an NVIDIA GPU to run. Due to COVID-19 restrictions, the only hardware available to test the device was the researcher's GTX 1080, but it should function on all devices which support CUDA 10.

The high-speed rendering requirements of the program necessitated the use of Vulkan over OpenGL. Vulkan gives the developer finer control over scheduling and allows the hardware to take shortcuts that it may not be able to do under OpenGL. For more on this decision see Section 6.1.3.

CHAPTER 5

# Design

When building a large project it's important to develop a consistent, logical, and properly separated design; both to make initial development easy and to make it intuitive for any future developers to understand. This applies to all aspects of the program including the codebase (i.e. which classes exist and how they communicate), how the program implements complex processes (such as the simulation/visualization), and how the end user will eventually use the program.

This section first separates the codebase into layers and analyses them in order. All notable design decisions for each layer are noted, and the means of interaction between these layers are documented. The final section, for the Command-Line layer, also documents the design of the command-line interface and the file formats used to store simulation states.

## 5.1 Code Structure

The project structure, shown in Fig. 5.1.1, is split into four layers: Command-Line, Visualization, Simulation, and Memory. Each element broadly represents a C++ class which depends on the classes defined below it.

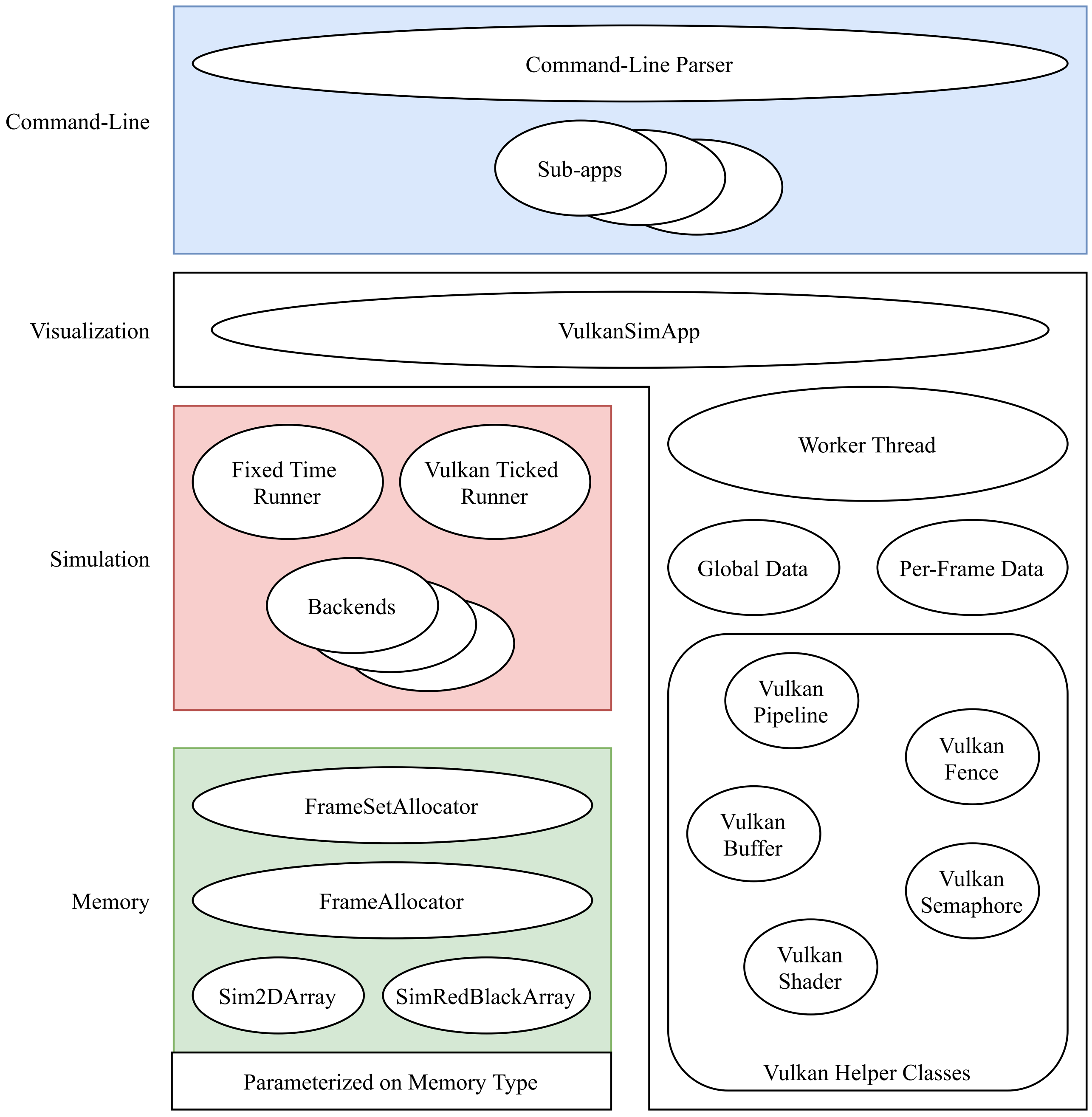


Figure 5.1.1: Overall Code Structure

The Command-Line layer has a Command-Line Parser, which converts the input command-line arguments to suitable representations, and a set of sub-apps implementing the subcommands shown in Section 5.4. The figure shows all classes relevant to the `run` subcommand, which shows a visualized simulation.

The Visualization layer contains a high-level `VulkanSimApp` class, which initiates all visualization-related code. Beneath that, the Worker Thread handles most Vulkan API calls, and depends on multiple sets of data built with Vulkan helper classes. This layer uses classes from the Simulation layer, which are reused for the headless simulation to avoid code duplication.

The Simulation layer consists of two main elements: Runners and Backends. The Runners use different strategies for invoking a Backend - the fixed-time Runner runs the simulation flat out until a specific time is reached, and the Vulkan ticked Runner runs the simulation for small timesteps while synchronising with the visualization. Each Backend implements the same functions, so Runner implementations can be Backend-agnostic. The fixed-time Runner supports all defined Backends (Section 5.2), but the Vulkan ticked Runner only supports the CUDA backend.

Finally, the Memory layer exposes APIs for the Backends to allocate simulation memory. Runners decide how many 'frames' to create (see Section 5.2.2), and use the `FrameSetAllocator` to create a set of `FrameAllocators`. The Backends use each `FrameAllocator` to allocate a set of buffers, which are then used to store simulation data, represented with `Sim2DArray` and `SimRedBlackArray` instances. The `SimRedBlackArray` splits a given grid size into two halves, storing red elements and black elements separately, and can be configured to store an additional full matrix (helpful for e.g. pressure, where both representations are useful).

All elements of the Memory layer are parameterized on the type of memory. This can be CPU memory, CUDA Unified Memory, or Vulkan on-device memory. The memory type affects not just the allocation method, but also the properties it has e.g. CPU memory cannot be accessed from the GPU. Using parameterized array classes allows these differences to be expressed while keeping a consistent interface.

## 5.2 Simulation & Memory Layer

To allow easy comparisons between CPU and GPU simulations the program contains multiple simulation backends. The headless and visualized simulations use a `--backend` command-line option to allow the user to choose the backend from this selection:

- Null, a backend which does no simulation for testing purposes.
- CPU Simple, equivalent to the pre-optimization CPU simulation.
- CPU Optimized, equivalent to the initial optimized CPU simulation, which produces identical results to CPU Simple.
- CPU Optimized Adapted, a version of CPU Optimized slightly modified to be closer to the GPU version.
- CUDA Backend V1, the only GPU-based backend.

The only modification present in the CPU Optimized Adapted backend is the removal of double-precision floating-point logic, which is not present on the GPU for speed concerns. It still isn't identical to the CUDA environment, as CUDA aggressively contracts floating-point operations to fused multiply-add while the CPU compiler does not. The other CPU optimizations are present on the GPU where possible: the residual check between each Poisson phase is still removed, and the red-black data is still separated into separate arrays.

### 5.2.1 CUDA Design

The CUDA backend implements each stage of the simulation (Fig. 2.1.2) as one or more CUDA Kernels. Each CUDA Kernel represents a computation for a single grid cell, which is executed by a GPU thread. The grid is split into small 'blocks' of threads, and the threads within each block are executed by a Streaming Multiprocessor (SM) in groups of 32 (a 'warp'). Grid and block dimensions can be specified in 1D, 2D, or 3D depending on the dimensionality of the problem. The program uses a 2D grid for most kernels because each computation reads adjacent data in 2D space. Reduction kernels treat the 2D grid as a flat 1D array, because there is no need to retrieve ‘adjacent’ data.

The CUDA backend must execute both with and without Vulkan, depending if a visualization is used, which affects the type of memory used in the simulation. When visualized, crucial data such as velocity and pressure are stored in shared Vulkan-CUDA memory, to allow direct usage from both APIs without copying the data. In all other cases CUDA Unified Memory is used, which can be paged between the CPU and GPU on-demand without manually mapping it across[47]. This allowed for granular debugging during development, as GPU kernels could be easily swapped out for known-correct CPU implementations without having to move memory manually.

### 5.2.2 N-Buffering

When developing the Visualization, it was noted that keeping a separate copy of the simulation output could allow the visualization to run in parallel with the simulation[1], without the simulation overwriting data currently being visualized. To allow this, N-buffering was introduced to the simulation backends and aspects of the visualization. Multiple ‘frames’ are stored, where each one contains all data used in a simulation tick. Each simulation tick is assigned to a specific frame chosen by the Runner, and only the data in this frame is written to. The last-written frame is then used as the input for the next simulation tick. During a simulation of frame #N, all other frames contain constant data and can be read out without any race conditions.

[1]This was unfortunately for other reasons discussed in Section 5.3.2

## 5.3 Visualization Layer

This section details the design employed to efficiently and effectively visualize simulation results in real-time.

### 5.3.1 Components

There are four major components of the visualization which work together to create the final output.

- CPU 0 - The Main Thread, which handles the CUDA simulation.
- CPU 1 - The Worker Thread, which records and enqueues Vulkan commands.
- The GPU, which executes both the CUDA and Vulkan commands.
- The Swapchain, which provides render targets that are displayed in the application window.

The GPU itself has three phases of execution, performed sequentially for each frame (Table 5.3.1). Each frame accesses a set of per-frame data, which is reused in a circular buffer. Each phase for the frame will use this data.

| Name | Abbreviation |
|---|---|
| Simulation | Sim |
| Visualization Compute (e.g. simulating particles) | VizComp |
| Visualization Graphics | VizGfx |

Table 5.3.1: GPU Execution Phases, with abbreviations

A key design goal with the visualization was to maximize GPU utilization. The CPU work is much less complicated than the GPU work, so the CPU should always be able to keep the GPU fed with new work. If the CPU failed to do this, the GPU would waste time idling when it could be doing useful work.

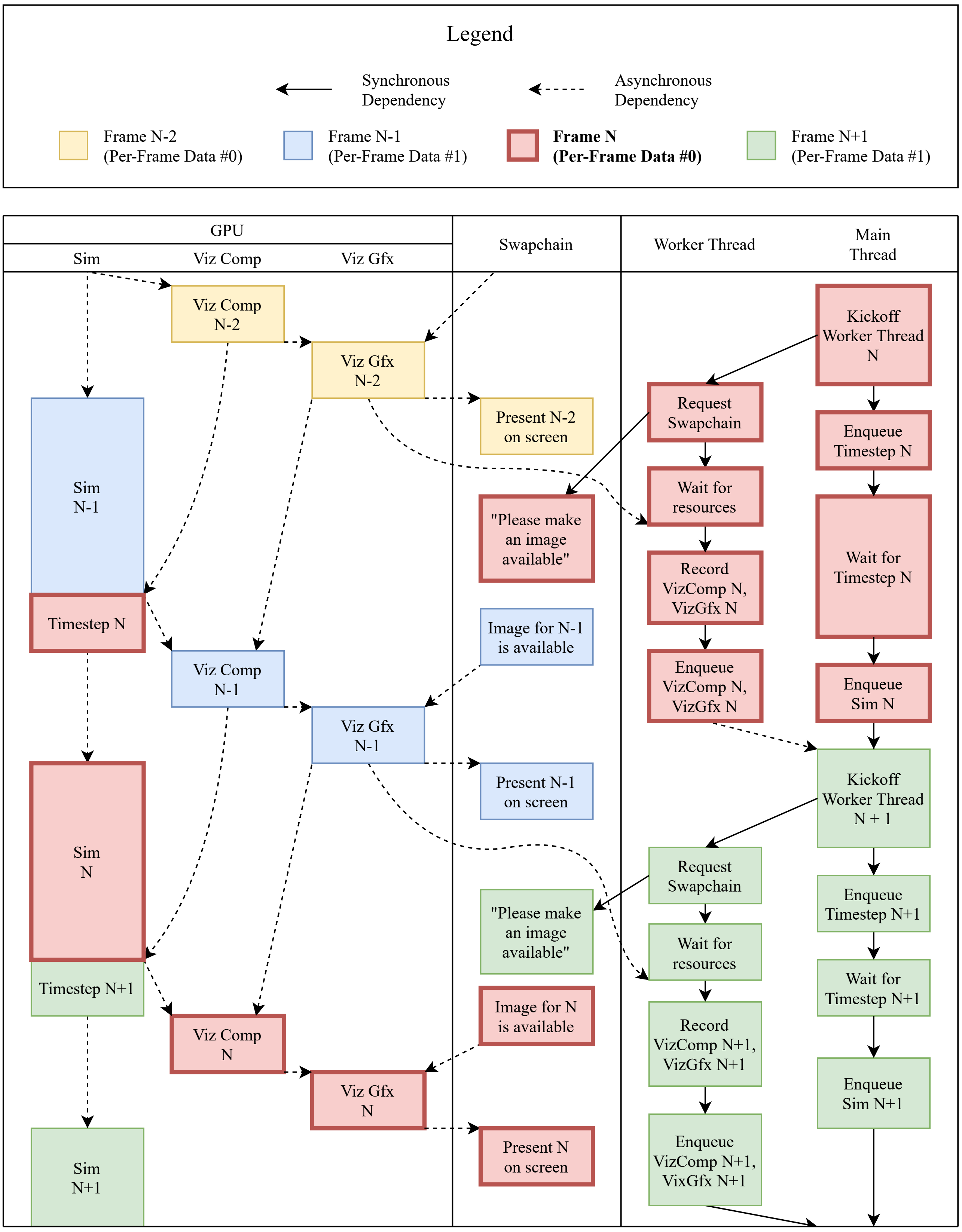


Figure 5.3.1: Timing Breakdown of four visualization frames, assuming two sets of per-frame data

### 5.3.2 Timing Breakdown

Fig. 5.3.1 shows a breakdown of multiple frames of simulation and visualization, focusing on **frame #N** which is highlighted in red. It includes both synchronous and asynchronous dependencies. A task with synchronous dependencies starts immediately once all dependencies finish. A task with asynchronous dependencies can only start once all dependencies are finished, but may not start until later.

The main thread begins by initiating the worker thread before handling the simulation. Before any CUDA work is enqueued, a semaphore is used to create a dependency on a previous compute job ('Viz Compute $N-2$'). This compute cycle and the incoming CUDA work will access the same per-frame data[2], so this dependency prevents the simulation from writing to the data while the compute job reads from it. For higher efficiency, this dependency could be inserted on 'Sim $N$' instead, but it would not affect the results in practice.

The `VulkanTickedRunner` enqueues CUDA work to determine the maximum timestep for the next tick ('Timestep $N$' in the diagram), waits to get the results back on the CPU, and enqueues the rest of the simulation based on the calculated timestep ('Sim $N$')[3]. If visualization requested a larger timestep than the calculated maximum, the process would be repeated until the total requested timestep had elapsed. Once 'Sim $N$' is finished, it signals a semaphore to allow 'Viz Comp $N$' to continue the frame.

The worker thread uses the `vkAcquireNextImageKHR` function to ask the swapchain[4] for an image. This returns the index of a swapchain image that will eventually become available, and a semaphore that will be signalled once this happens. Before 'Viz Gfx $N$' can render to this image, it has to wait for this semaphore to signal it is ready. The thread then waits for frame $N-2$ to finish using the Vulkan resources in Resource Set 0 before it uses them to record 'Viz Comp $N$' and 'Viz Gfx $N$'.

[2]Viz Gfx jobs don't access the raw simulation buffers, so it doesn't delay the simulation.
[3]'Sim' and 'Timestep' work are enqueued on the same CUDA stream, so they are implicitly ordered.
[4]The swapchain is a set of images provided by the OS that are shown in the windowing system

The Viz jobs use semaphores to guarantee ordering: 'Viz Gfx $N$' must start after 'Viz Comp $N$' finishes, which must start after 'Sim $N$' finishes. 'Viz Comp $N$' also has to wait for 'Viz Gfx $N-1$' to finish to avoid race conditions - all 'Viz Comp' and 'Viz Gfx' jobs share global memory instead of per-frame memory to avoid data copying. Once 'Viz Gfx $N$' finishes, it signals a semaphore to tell the swapchain/OS to present the newly rendered frame to the screen. The worker thread records and enqueues the above work for the GPU. Once the worker thread has finished it signals the main thread, and once the main thread finishes enqueueing 'Sim $N$' the process restarts.

It's worth noting that based on these dependencies some GPU work could theoretically run in parallel, such as 'Viz Comp $N$' and 'Timestep $N-1$'. Unfortunately, in practice this doesn't happen. Running parallel compute workloads is only supported on NVIDIA GPUs from the Ampere generation onwards[33], such as the RTX 3000 series, and is not supported on the researcher's GTX 1080. This also affects the work breakdown, preventing smaller pieces of work (such as computations for separate visualization layers) from running in parallel. In the future, this could be mitigated by using an Ampere-level GPU, or by running the simulation and visualization on separate GPUs.

#### Synchronization

Implementing the unlocked framerate (Req. F5.5) without running the simulation faster than real-time required the time taken for each frame to be measured. This is approximated by measuring the time between 'Kickoff Worker Thread' tasks on the CPU. Using this, the program predicts how long a simulation frame *would* take, and combines that with the requested simulation rate[5] to predict if the next frame should be a simulation frame.

This restriction can be removed, to run with an unlocked simulation rate, in which case the average time per simulation frame is chosen as the timestep for the

[5] e.g. 120 simulation ticks per second

next simulation tick. This is capped with sensible minimum/maximum limits to avoid instability from outliers.

### 5.3.3 Visualization Work Breakdown

As specified in the Requirements (Chapter 4) the selected visualization layers are the Background, Quantity-by-Scalar, Quantity-by-Vector, and Particle Simulation. The Background and Quantity-by-Scalar layers are visualized at the same time for simplicity. Accounting for this, the breakdown of required work for each layer is shown in Fig. 5.3.2.

The Compute sections are implemented using Vulkan Compute Shaders[56], which are nearly equivalent to CUDA Kernels and are invoked similarly. The Graphics sections are implemented using Vertex and Fragment Shaders, where the Vertex Shader determines the onscreen positions of the vertices that make up a model, and the Fragment Shader determines the colour of the onscreen pixels between those vertices.

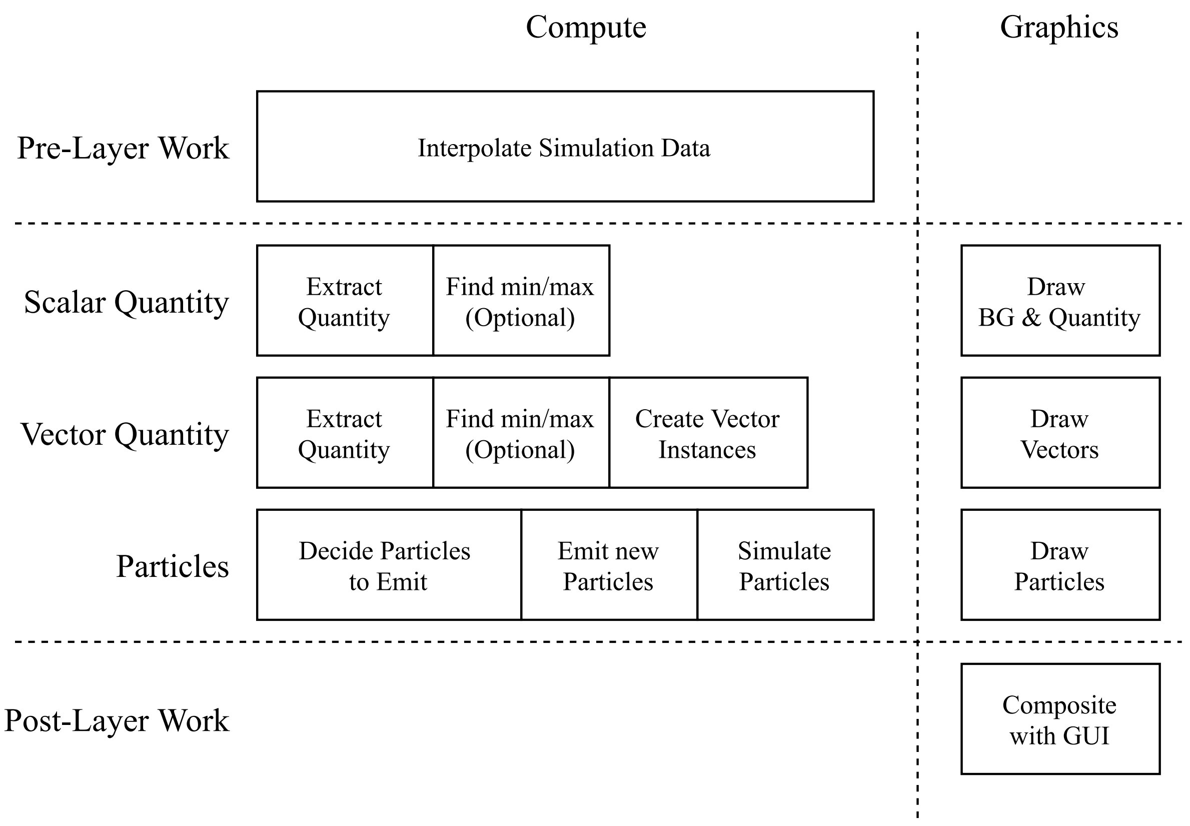


Figure 5.3.2: Visualization Work Breakdown

### Data Interpolation

The simulation stores data on a staggered grid (Fig. 2.1.1), but this is inconvenient for the visualization. The first step of the visualization is to move the exposed simulation data into a 2x resolution texture, applying interpolation where necessary, allowing the GPU to sample the exposed data at arbitrary points using the texture filtering hardware. This applies trilinear interpolation, as required for the particle simulation.

### Auto-ranging

Both Quantity-by-Scalar and Quantity-by-Vector have an optional auto-range mode, where the minimum and maximum values for the quantity are calculated and used instead of the user-defined range. This requires a GPU reduction, which is implemented in Vulkan just like it is in CUDA, using the second kernel of [24]. To simplify the rendering code, in both cases the selected quantity is extracted to two buffers using a specialized compute shader. The first buffer includes the quantity with a 'fluidmask' which shows if the selected pixel is a fluid or obstacle, and is used for the rendering along with the reduction result. The second buffer is used for the reduction, and contains a min/max property for each element.

### Indirect Instanced Rendering

For Quantity-by-Vector and the Particle Simulation, the final outputs are rendered using Instanced Rendering. The same model is rendered $N$ times, and the Vertex Shader uses an 'instance index' $0 \leq i < N$ to look up the instance's position/orientation. This method is much faster than rendering each instance separately, as it requires fewer draw calls on the CPU.

When recording the command buffer on the CPU, the number of instances for both layers is not yet known. For Quantity-by-Vector, the number of vectors is dependent on the simulation output. The previous visualization phase may kill some particles, so the particles cannot be predicted either. To mitigate this, Indirect in-

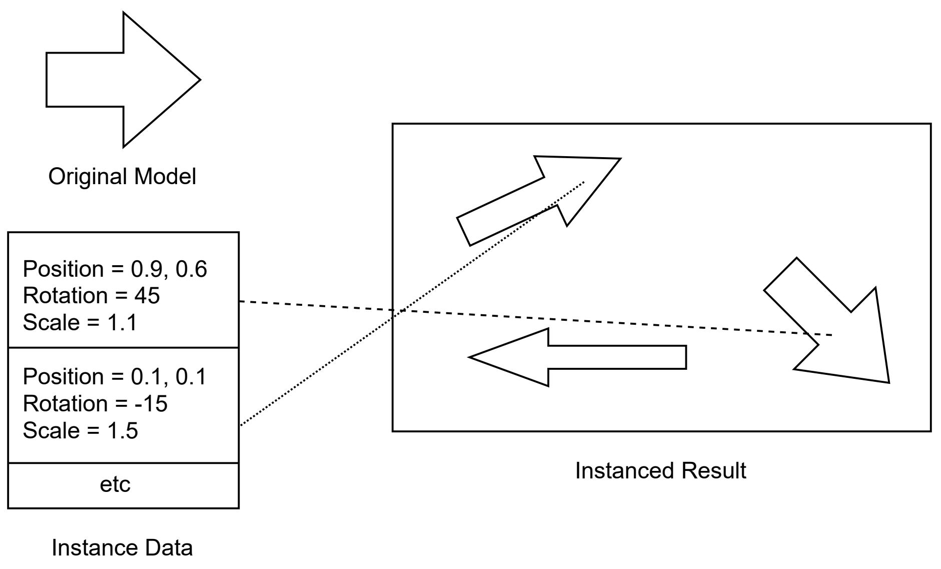


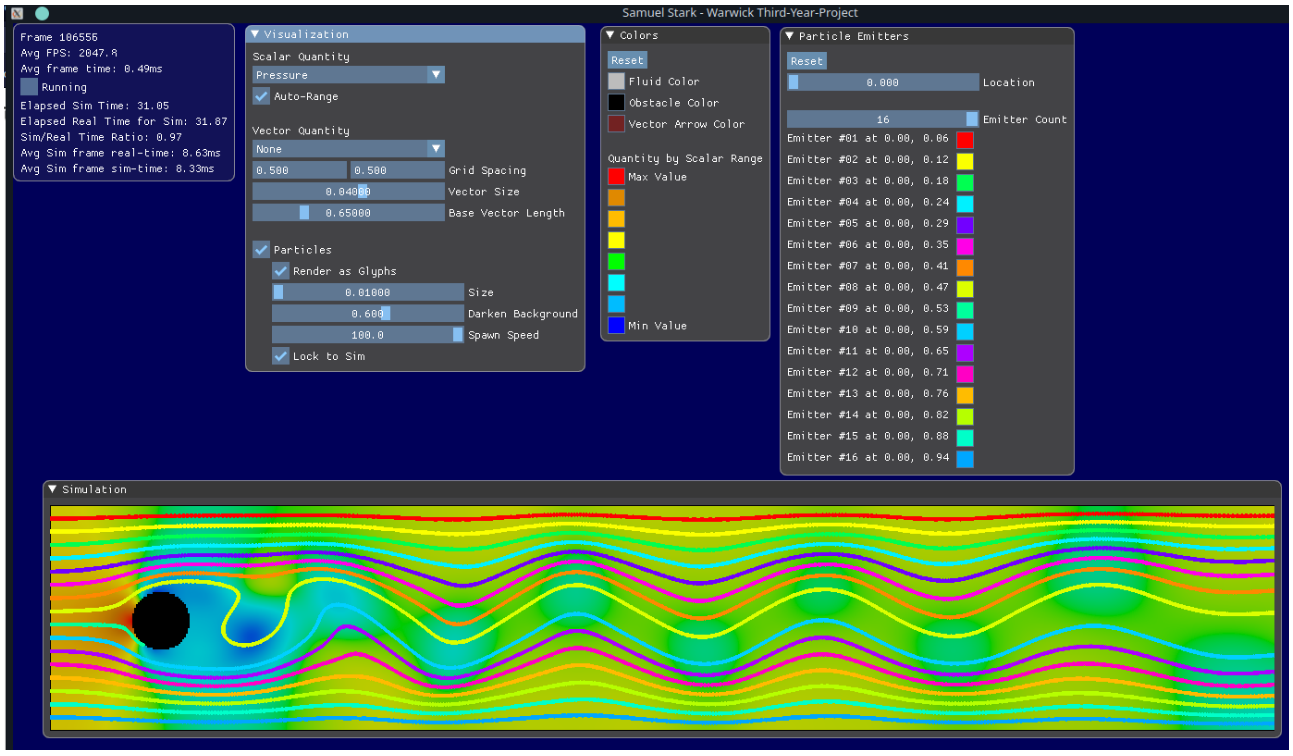
Figure 5.3.3: Instanced Rendering Demonstration

vocations are used for the vector/particle rendering and the particle simulation. Instead of specifying the instance count at record time, a reference to a GPU buffer is used. This GPU buffer contains the required parameters for the instanced rendering/compute dispatch, and can be atomically written by the GPU in a separate compute shader.

Creating new vectors/particle instances is done safely on the GPU using growable lists and atomic variables. Each growable list consists of an array of values with a maximum length, and a 'current size' variable. When a value is added, the 'current size' variable is incremented atomically, and the pre-increment value is then used to index the array and write the new instance parameter. When a value is removed, the 'current size' variable is decremented atomically, and the post-increment value can be used to see the deleted index. Within a compute shader invocation, a list can only be growable or shrinkable but cannot be both. Despite this limitation, the lists are suitable to implement the technique from [68].

### Final Composite

As a final step, the visualization output is rendered with the other GUI elements. This step is controlled by the Dear ImGUI library (see Section 6.1.3). An example of the visualization GUI is shown in Fig. 5.3.4.



Samuel Stark - Warwick Third-Year-Project

Frame 106556
Avg FPS: 2047.8
Avg frame time: 0.49ms
Running
Elapsed Sim Time: 31.05
Elapsed Real Time for Sim: 31.87
Sim/Real Time Ratio: 0.97
Avg Sim frame real-time: 8.63ms
Avg Sim frame sim-time: 8.33ms

▼ Visualization
Scalar Quantity
Pressure
✓ Auto-Range
Vector Quantity
None
0.500 0.500 Grid Spacing
0.04000 Vector Size
0.65000 Base Vector Length
✓ Particles
✓ Render as Glyphs
0.01000 Size
0.600 Darken Background
100.0 Spawn Speed
✓ Lock to Sim

▼ Colors
Reset
Fluid Color
Obstacle Color
Vector Arrow Color
Quantity by Scalar Range
Max Value
Min Value

▼ Particle Emitters
Reset
0.000 Location
16 Emitter Count
Emitter #01 at 0.00, 0.06
Emitter #02 at 0.00, 0.12
Emitter #03 at 0.00, 0.18
Emitter #04 at 0.00, 0.24
Emitter #05 at 0.00, 0.29
Emitter #06 at 0.00, 0.35
Emitter #07 at 0.00, 0.41
Emitter #08 at 0.00, 0.47
Emitter #09 at 0.00, 0.53
Emitter #10 at 0.00, 0.59
Emitter #11 at 0.00, 0.65
Emitter #12 at 0.00, 0.71
Emitter #13 at 0.00, 0.76
Emitter #14 at 0.00, 0.82
Emitter #15 at 0.00, 0.88
Emitter #16 at 0.00, 0.94

▼ Simulation

Figure 5.3.4: Example of the Visualization GUI

## 5.4 Command-Line Layer & Program Usage

The compiled binary uses a command-line interface to configure and run one of many subcommands available. These subcommands are:

- `makeinput`, which generates simulation input files, fulfilling Req. F3.
- `fixedtime`, which runs a headless simulation for a fixed time, fulfilling Req. F4.
- `compare`, which compares two simulation states for equality (see Section 5.4.3), fulfilling Req. F7.
- `renderppm`, which visualizes a static simulation state using the techniques from Section 2.3.2.
- `run`, which starts a real-time visualized simulation, fulfilling Req. F5.

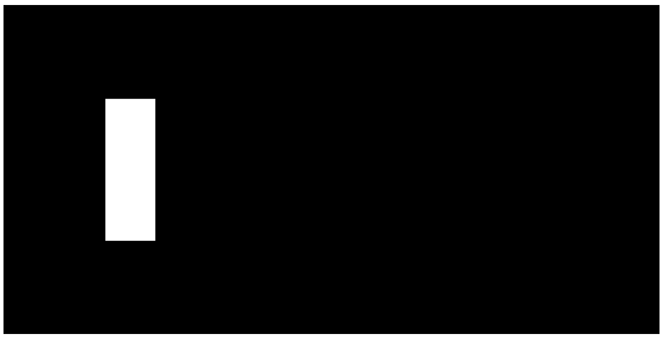
Splitting the program into subcommands was inspired by Git[70], and avoids creating separate binaries for each operation. Each subcommand can be configured with command-line options conforming to POSIX standard[71]. Examples of using the program are in Fig. 5.4.1.

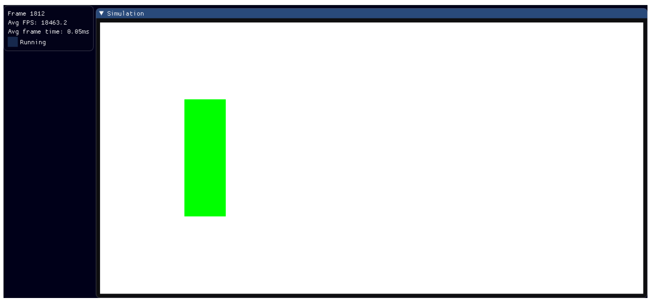
```
# Create an input file based on simple_layout with a size of 1x2 metres
./sim_cuda makeinput ./simple_layout.png 1 2 ./initial.bin

# Run it in headless mode for 10 seconds
./sim_cuda fixedtime --backend=cuda ./fluid.json ./initial.bin 10  -o ./output_after_10.bin

# Compare it to the expected output
./sim_cuda compare ./output_after_10.bin ./expected_after_10.bin

# Render it out to an image
./sim_cuda renderppm ./output_after_10.bin zeta ./output_after_10.ppm

# Try visualizing it in real-time
./sim_cuda run --backend=cuda ./fluid.json ./initial.bin
```

Figure 5.4.1: Example usage of the simulation program

### 5.4.1 Generating Inputs

The `makeinput` subcommand allows input simulation states to be generated from image files. Each pixel of the input image represents a cell of the grid, not including padding cells, where non-black pixels denote obstacle cells and all other cells are fluid. The example in Fig. 5.4.2 shows an example file denoting a rectangular obstacle, and the visualization of the generated state.


(a) Base Image



```
Frame 1812
Avg FPS: 18463.2
Avg frame time: 0.05ms
Running
```

(b) Simulation

Figure 5.4.2: Example conversion of an image to a simulation state

Velocities and pressure in every cell can be interpolated horizontally - 1 m/s east at the left edge, 0 m/s at the right. This alleviates simulation instability near obstacle edges, an advantage over having a constant initial velocity across the field. One such instability would be a situation where fluid is occluded from the input direction by an obstacle, but moves east anyway with no reason to do so.

The exact initial value of pressure is inconsequential as the simulation only cares about the difference between cells. The pressure can be set to 0 at all points, representing a constant pressure across the simulation grid. This is inconsistent with the nonzero velocities mentioned above, but applying variable pressure made the system more unstable.

### 5.4.2 File Formats

To fulfil Req. F1 two file formats have been defined to store simulation data and parameters.

#### Fluid Parameters

Parameters that are characteristic of a particular fluid or simulation type are stored in a "Fluid Parameters" file. This includes the Reynolds number, the timestep safety factor, and the maximum iteration count for the Poisson solver. They are stored in a JSON format to be human-readable, are reusable for different simulation states, and can be easily edited by the end user. An example is shown in Fig. 5.4.3.

```json
{
    "Re": 150.0,
    "initial_velocity_x": 1.0,
    "initial_velocity_y": 0.0,
    "timestep_divisor": 60,
    "max_timestep_divisor": 480,
    "timestep_safety": 0.5,
    "gamma": 0.9,
    "poisson_max_iterations": 100,
    "poisson_error_threshold": 0.001,
    "poisson_omega": 1.7
}
```

Figure 5.4.3: Example Fluid Parameters file

#### Simulation State

Data unique to an individual state such as simulation resolution, physical size, and velocity fields are stored in a binary format reused from the original simulation. As the data is much more sensitive to individual modifications[6], it makes more sense to store this data in a binary format where it cannot be easily modified by a user. Additionally, the binary format is much smaller than any text-based format, which helps as the volume of data stored is much larger than that stored in the fluid parameters.

[6]For example, changing a single value in the velocity field can introduce discontinuities.

The header consists of a pair of unsigned 32-bit integers specifying the resolution of the simulation, and a pair of 32-bit floating-point numbers specifying the physical dimensions of the simulation. From there, four sets of data for each column are stored, including the boundary padding squares:

1. Horizontal Velocity $u$ (`float32`)
2. Vertical Velocity $v$ (`float32`)
3. Pressure $p$ (`float32`)
4. Cell Flags, defining which adjacent squares are boundaries (`uint8`)

This structure is somewhat unintuitive and error-prone, an example being the Cell Flags which may end up being inconsistent between adjacent cells, but it has been kept for the sake of compatibility with the original simulation.

### 5.4.3 Comparison Heuristics

In the `compare` subcommand heuristics are used to judge if one simulation is accurate and precise with respect to the other. This does not quite fulfil Req. F7.1, as there are two results and two heuristics used instead of just one, but it is useful for comparisons regardless so was not changed.

This assumes one of the supplied states is a known-valid simulation state, and the other is not. The velocity and pressure values $u, v, p$ of the two simulation states are compared separately. The simulation states must be of the same size and use the same obstacle squares.

The comparison is performed by calculating the mean and standard deviation of the square error between the datasets. These are then compared to tolerance values to produce two binary outputs: ACCURATE if the mean is below tolerance, and PRECISE if the standard deviation is below tolerance. Examples are shown in Fig. 5.4.4.

The tolerance for the mean was derived from an expected error magnitude of $\pm 10^{-7}$, which was squared to produce $10^{-14}$. It is assumed that the standard deviation should always be smaller than the mean, so the tolerance for standard deviation is also $10^{-14}$.

```
Velocity X:
    Sq. Error Mean:     0     ACCURATE
    Sq. Error Std. Dev: 0     PRECISE
Velocity Y:
    Sq. Error Mean:     0     ACCURATE
    Sq. Error Std. Dev: 0     PRECISE
Pressure:
    Sq. Error Mean:     0     ACCURATE
    Sq. Error Std. Dev: 0     PRECISE
```

(a) Comparison of Equal States

```
Velocity X:
    Sq. Error Mean:     0.0233842     INACCURATE
    Sq. Error Std. Dev: 0.0996487     IMPRECISE
Velocity Y:
    Sq. Error Mean:     0.00566354    INACCURATE
    Sq. Error Std. Dev: 0.0139529     IMPRECISE
Pressure:
    Sq. Error Mean:     0.0214799     INACCURATE
    Sq. Error Std. Dev: 0.0511252     IMPRECISE
```

(b) Comparison of Unequal States

Figure 5.4.4: Examples of outputs from the comparison tool

CHAPTER 6

# Implementation

Building the program was a huge technical challenge on many levels, resulting in 8.5k lines of code spread over 146 files in three different programming languages. Solving the more complicated problems required some interesting tricks which may have interacted with lesser-known language features, memory models, and in one case the particulars of the C++17 specification. The correctness of the resulting program was then ensured through the use of other features, including C++ macros and files that crossed language boundaries[1]. This chapter documents these tricks and the background needed to understand them.

The first section contains an overview of relevant C++ features, and some other concepts employed while developing the program. The next section focuses on Code Safety, detecting any faults during compilation and then ensuring any other faults do not then manifest into problematic errors. As in the Design section, each layer of the codebase is then examined and all interesting problems solved during development are documented.

[1]See Section 6.5.3.

## 6.1 Preliminary Work & Background

The primary languages used in the program are C++17 and CUDA. This section will explain key elements of C++17 used in the program, the build system, and the external libraries used.

### 6.1.1 C++ Primer

Virtual classes use virtual functions to allow subclasses to override behaviour in the parent. The seminal example is creating a parent class `Animal` that can `talk()`, and a subclass `Dog` that overrides `talk()` to bark. When a virtual function is called on an object, instead of statically determining which function to call at compile-time, the *vtable* of the object is read out at run-time with the correct function pointer[34]. In Java and Python all functions are considered virtual, but in C++ virtual behaviour can be selectively enabled. As each virtual function call requires multiple indirections (object → vtable → function), the performance is slightly worse than using normal functions (see Fig. 6.1.1). Virtual functions are avoided where possible in the codebase.

One of C++'s greatest innovations over C is the template system. Classes and functions can be 'templated' on types or values, and then 'instantiated' when these parameters are known. When such a class or function is instantiated a complete copy is created with the new parameter values, which is compiled and optimized separately from any other instantiations. Some classes can also be "specialized" to implement custom behaviour for specific parameter values. This is useful for encoding extra information in a type for safety, e.g. `VulkanShader<Vertex>` cannot be passed to a function expecting `VulkanShader<Compute>` because they're independent types. It's also useful for static function dispatch, as instead of taking a virtual class with a `talk()` function you can instead template a function on the type of animal it uses, and call the function directly. This technique is used in the Simulation to efficiently use Backends.

```cpp
class Animal {
public:
    virtual void talk() = 0;
    virtual ~Animal() = default;
};

class Dog : public Animal {
public:
    void talk() override {
        printf("bark\n");
    }
    void talk_static() {
        printf("bark\n");
    }
    ~Dog() override = default;
};

int main() {
    // Virtual call
    Animal* animal = new Dog();
    animal->talk(); // Prints 'bark'
    delete animal;

    Dog dog = {};
    dog.talk_static();

    return 0;
}
```

(a) C++ implementation

```asm
# Calling the function virtually
↪  (animal->talk())
mov     rax, QWORD PTR [rbp-24]
mov     rax, QWORD PTR [rax]
mov     rdx, QWORD PTR [rax]
mov     rax, QWORD PTR [rbp-24]
mov     rdi, rax
call    rdx

# Calling the function statically
↪  (dog.talk_static())
lea     rax, [rbp-32]
mov     rdi, rax
call    Dog::talk_static()
```

(b) x86 Assembly for calling the functions

Figure 6.1.1: Inefficiencies of virtual inheritance
(x86 assembly code generated from https://godbolt.org/z/PfEq3TPdn)

```cpp
class Dog {
public:
    void talk() {
        printf("bark\n");
    }
};

class Cat {
public:
    void talk() {
        printf("meow\n");
    }
};

template<class TAnimal>
void make_animal_talk(TAnimal* animal) {
    animal->talk();
}

int main() {
    Dog dog{};
    // Instantiates make_animal_talk<Dog>,
    ↪  which calls Dog::talk statically
    make_animal_talk<Dog>(&dog);

    Cat cat{};
    // Instantiates make_animal_talk<Cat>,
    ↪  which calls Cat::talk statically
    make_animal_talk<Cat>(&cat);

    return 0;
}
```

```asm
# void make_animal_talk<Dog>(Dog*):
# ...
mov     rax, QWORD PTR [rbp-8]
mov     rdi, rax
call    Dog::talk()
# ...

# void make_animal_talk<Cat>(Cat*):
# ...
mov     rax, QWORD PTR [rbp-8]
mov     rdi, rax
call    Cat::talk()
# ...
```

Figure 6.1.2: Using templates for polymorphism
(x86 assembly code generated from https://godbolt.org/z/hfM465EYa)

### "Typeclasses"

In other languages, like Haskell, a typeclass defines some behaviour a class should fit. From [35]: "If a type is a part of a typeclass, that means that it supports and implements the behaviour the typeclass describes". C++17 does not have a convenient way of denoting this but it is incredibly helpful when building generic code with templates, as it allows the generic code to make assumptions about what behaviour types will support. The rest of this chapter will define typeclasses where convenient to describe behaviour shared by certain classes.

### 6.1.2 Build System

The build system is implemented in CMake as specified in Section 7.3. This section highlights a few changes that were made to an otherwise standard setup to accommodate the project.

#### CUDA-less Binaries

The project can be built to produce both CUDA and CUDA-less binaries, in case it needs to be run on computers without CUDA. The list of regular C++ source files and CUDA source files are maintained separately. A CUDA-less binary (`sim_nocuda`) will only build the C++ files while a CUDA binary (`sim_cuda`) will build both. When building with CUDA support the preprocessor macro `CUDA_ENABLED` is defined in all source files, including the C++ files. This allows support for CUDA backends in C++ code (i.e. as selectable options on the command-line) to be conditionally enabled without maintaining two copies of the relevant source files. In Fig. 6.1.3 (which has been amended for brevity), the switch statement only contains a case for CUDA if the directive is set, triggering a fatal error otherwise. The enumeration defining the existing Backends also uses this technique to completely remove the concept of a CUDA backend from non-CUDA builds.

```
switch(backendType) {
    case Null:
        return SimFixedTimeRunner<NullSimulation, Host2DAllocator>();
    case CpuSimple:
        return SimFixedTimeRunner<CpuSimpleSimBackend, Host2DAllocator>();
    case CpuOptimized:
        return SimFixedTimeRunner<CpuOptimizedSimBackend, Host2DAllocator>();
    case CpuAdapted:
        return SimFixedTimeRunner<CpuOptimizedAdaptedSimBackend, Host2DAllocator>();
#if CUDA_ENABLED
    case CUDA:
        return SimFixedTimeRunner<CudaBackendV1<true>, CudaUnified2DAllocator>();
#endif
    default:
        FATAL_ERROR("Enum val %d doesn't have an ISimFixedTimeRunner!\n", backendType);
```

Figure 6.1.3: Conditionally supporting CUDA based on a preprocessor directive

#### Shader Build Infrastructure

The shaders used for visualization are written in GLSL, with appropriate extensions to be compatible with Vulkan. They are separated by file type, with Vertex shaders in `.vert` files, Fragment shaders in `.frag` files, and Compute shaders in `.comp` files. As Vulkan does not natively support GLSL, they must be compiled to SPIR-V before they can be used. CMake does not support GLSL as a first-class language, so a custom build command was used to compile them with `glslc`[72] when they change. This allows them to be treated just like any other source file from the programmer's perspective. SPIR-V files are placed in a `shaders` directory next to the binaries, where they can be easily accessed and passed to Vulkan.

### 6.1.3 Library Selection

| | OpenGL | Vulkan |
|---|---|---|
| OpenCL | Y | N |
| CUDA | Y | Y |
| OpenGL | Y | N |
| Vulkan | N | Y |

Figure 6.1.4: Graphics and Compute Backend Interoperability Matrix

CUDA and Vulkan have been chosen as backends, but other backends were also considered. As the simulation would have to run on DCS systems (Req. NF11) and thus run on Linux, the only possible GPU rendering backends were OpenGL and Vulkan. However, there were still multiple choices of compute backend:

- OpenCL[54] is an "Open Standard for Parallel Programming of Heterogeneous Systems"[57].
- CUDA[48] is a proprietary library for running parallel code on NVIDIA GPUs.
- OpenGL has Compute Shaders[55] which can execute computations outside of the graphics pipeline.
- Vulkan also has Compute capability[36], similar in function to OpenGL.

To decide on the compute backend to use, an interoperability matrix was drawn (Fig. 6.1.4) to show which libraries could share data without copying it between buffers. As the researcher was already experienced with Vulkan, and the more granular control it provides would be beneficial to performance, Vulkan was selected as the rendering backend. This prevented OpenCL and OpenGL from being used as compute backends, as they are not compatible with Vulkan. CUDA and Vulkan have comparable ability, but CUDA was chosen as the compute backend. The Vulkan compute shaders are still a very graphics-oriented view of computation, and CUDA would give the researcher experience with other kinds of libraries. A Vulkan compute backend is used for the visualization portion of the code.

In other cases, there were clear choices: the SDL2[73] window and input library and the Dear ImGUI[74] UI library were chosen due to personal experience. The `stb_image.h` header was found to be a simple method of importing image colour data as byte arrays, used for the input generator (Req. F3).

There are a great many options for Command-Line parsing libraries, even more so because C++ is used instead of C. A recent survey of the possibilities[75] was whittled down to five options.

`getopt`[76], `argp`[77], and `gopt`[78] are C libraries that use arrays of structures to define the required arguments. Of them, only `argp` can automatically generate a `--help` argument, which is a very valuable feature. `cxxopts`[79] was considered as a C++ alternative but used very odd syntax for defining arguments. Ultimately CLI11[80] was chosen as a modern C++11 library that had native support for subcommands, which were used heavily for separating program components (see Section 5.4).

## 6.2 Code Safety

No program is faultless, and when faults manifest during execution it's important to ensure they have a minimal effect on their surroundings. This program includes many means of detecting errors both at compile-time and run-time to ensure its dependability.

The G++ flag `-Wall` enables many warnings that are emitted at compile-time if potential errors are detected. Unlike some other languages (i.e. Verilog) these warnings are generally reliable and it is feasible to build a C++ program that compiles without any warnings. To ensure this the `-Werror` flag is added to upgrade these warnings to compilation errors, ensuring a program which compiles will be warning-free[2].

Runtime errors are detected with a set of C++ macros that check if required conditions are met. As there is no liveness requirement for the program, failure triggers an immediate program exit to avoid errors propagating through the system. The `DASSERT` family of macros are included only in Debug builds, and the `FATAL_ERROR` family of macros test both in Release and Debug. Both families print a message including the file and line of code that triggered the error, and any other relevant debug information. These families are also integrated with the `CHECKED_CUDA` and `CHECKED_VULKAN` macro families, which surround CUDA/Vulkan function calls and check the returned error codes. An example is shown in Fig. 6.2.1.

The final tool for error detection is the Vulkan Validation Layer. This is a standard Vulkan extension which checks each Vulkan call to ensure the Vulkan specification [56] isn't violated. These checks are very in-depth, and are a must-have when debugging visualization errors. The base Vulkan functions don't do this error checking for efficiency's sake, and the program disables these layers in Release mode for the same reason.

[2]Some warnings, such as those from `-Wextra` and those relating to unused variables and parameters, are usually benign so weren't upgraded.

```
if (value == unexpected) {
    FATAL_ERROR(
        "Unexpected Value %d\n",
        value
    );
}
// equivalent to
FATAL_ERROR_IF(
    value == unexpected,
    ...
);
// or
FATAL_ERROR_UNLESS(
    value != unexpected,
    ...
);

// Same, but only fails in Debug builds
DASSERT(value != unexpected);
```

(a) Example of assertion macros

```
cudaError_t error = cudaDeviceSynchronize();
FATAL_ERROR_IF(error != cudaSuccess);
// Equivalent to
CHECKED_CUDA(cudaDeviceSynchronize());

auto result = vkDeviceWaitIdle();
FATAL_ERROR_IF(result != vk::Result::eSuccess);
// Equivalent to
CHECKED_VULKAN(vkDeviceWaitIdle());
```

(b) Example of API failure safety macros

Figure 6.2.1: Examples of error safety via macros

### 6.2.1 Smart Resource Classes

All memory allocations, CUDA objects, and Vulkan objects follow the same allocate/release pattern. They are not destroyed automatically when they go out of scope, but must be released manually. This is an error-prone process, as a programmer may forget which resources need to be released or try to release a resource twice.

Smart resource classes alleviate this by tying the resource lifetime to a C++ object, using the object *destructor* to release the resource. The prime example of this is `std::unique_ptr<T>`, which holds a `T*` object and `free()`-s it when the object leaves scope (Fig. 6.2.2). However this does not easily map to other deletion methods, and the pointer adds an unwanted extra level of indirection. `vulkan.hpp`, included in the Vulkan SDK, includes similar classes for each type of Vulkan resource, but still requires a lot of boilerplate to initially create objects[3]. In some cases, it's also more convenient to store multiple resources together, such as a buffer and the

[3]Vulkan objects require a full CreateInfo struct to create, rather than taking function arguments.

```
// Manual handling
{
    auto* semaphore = new Semaphore();
    // do things with semaphore...
    delete semaphore;
}

// Automatic handling
{
    std::unique_ptr<Semaphore> semaphore = std::make_unique<Semaphore>();
    // do things with semaphore...
    // automatically deleted
}
```

Figure 6.2.2: Example of memory management with C++ standard classes

device memory it uses, which cannot be directly achieved with either method. To solve these problems custom smart resource classes are implemented for individual resources and aggregates, with more convenient constructors (see Appendix A).

These smart classes are affected by C++ copy/move semantics. C++ allows objects to be copied with a copy-constructor, or moved with a move-constructor. The resource objects should not be copyable as it would become unclear which copy would be responsible for destroying the resource. The copy-constructor can be deleted to prevent this, but the move-constructor is useful for transferring ownership e.g. from a resource factory to the person using the resource.

When an object is moved, the original version should forget the data it's holding and give it to the new object. C++ can generate this constructor automatically, but this “default move-constructor” will just copy the data across if the data is *Trivially-Copyable*[81]. All pointers and CUDA handles are *TriviallyCopyable*, so they don't get forgotten automatically. Manually writing each move-constructor to address this would be error-prone, so instead the `ForgetOnMove<T>` class is used to wrap these values and automatically forget them when the move-constructor is invoked. The final result is shown in Fig. 6.2.3: a clean, easy, and safe method of implementing new smart resource classes.

```cpp
// Doesn't use ForgetOnMove<>
class ComplexWrapper {
    void* memoryPointer;

    // Constructor
    ComplexWrapper(size_t memorySize) {
        // Allocate memory
    }
    // Destructor
    ~ComplexWrapper() {
        if (memoryPointer != nullptr) {
            // Free memory
        }
    }

    // Copy constructor - deleted
    ComplexWrapper(const ComplexWrapper&) = delete;
    // Move constructor - complicated
    ComplexWrapper(ComplexWrapper&& movedFrom) {
        this->memoryPointer = movedFrom.memoryPointer;
        movedFrom.memoryPointer = nullptr;
    }
};

// Uses ForgetOnMove<>, is simpler
class SimpleWrapper {
    ForgetOnMove<void*> memoryPointer;

    // Constructor as before
    SimpleWrapper(size_t memorySize) {
        // Allocate memory
    }
    // Destructor as before, but checks if the memoryPointer is present
    ~SimpleWrapper() {
        if (memoryPointer.has_value()) {
            // Free memory
        }
    }

    // Copy constructor - deleted
    SimpleWrapper(const SimpleWrapper&) = delete;
    // Move constructor - defaulted!
    // We don't have to write this in full
    SimpleWrapper(SimpleWrapper&&) = default;
};
```

Figure 6.2.3: Automatic forgetting with `ForgetOnMove<T>` vs. manual handling

## 6.3 Memory Layer

As mentioned in the Design section, all elements of the Memory system are parameterized on the memory type. This was accomplished by creating an enum MType and a set of classes templated on it (Fig. 6.3.1). These templates were then specialized for each type of memory, implementing the logic for each type separately. Separate implementations were required due to the unique constraints and allocation methods of each memory type.

```
enum MType {
    CPU,
    Cuda,
    VulkanCuda
};

enum RedBlackStorage {
    RedBlackOnly, // Just store the red and black matrices
    WithJoined // Store the red, black, and combined matrices
};

typeclass DataArray {
    // Has a static value MemType telling you what memory type it takes
    static MType MemType;
    // Has a function for calculating the total bytes used for a matrix
    static size_t sizeBytesOf(Size<uint32_t> size);
}

class Sim2DArray<T, MType> fits DataArray;
class SimRedBlackArray<T, MType, RedBlackStorage> fits DataArray;

typeclass FrameAllocator<MType> {
    // Function for allocating a 2D array
    Sim2DArray<T, MType> allocate2D(Size);

    // Function for allocating a red/black array
    SimRedBlackArray<T, MType, RBStorage> allocateRedBlack(Size);
}

typeclass FrameSetAllocator<MType, TFrame> {
    std::vector<TFrame> frames;
}
```

Figure 6.3.1: Memory Layer Typeclasses

### 6.3.1 Array Handles

The `Sim2DArray<...>` and `SimRedBlackArray<...>` classes represent handles to 2D arrays of values of arbitrary types. Both implement a common *DataArray* typeclass (Fig. 6.3.1). They do not own the data they point to, so if a `Sim2DArray` is destroyed the referenced memory isn't freed. Freeing memory is handled by the `FrameAllocator` instead.

`SimRedBlackArray` serves as an aggregate of `Sim2DArrays`, without defining any special behaviour. It does not specialize on the memory type, but provides two storage variants `RedBlackOnly` and `WithJoined` (Fig. 6.3.1).

`Sim2DArray` implements a set of functions for accessing data from the CPU, CUDA, and Vulkan. These functions are only present if that access type is supported, so trying to use an unsupported function results in a compile-time error. Other generic operations are also supported such as zeroing out memory, copying memory in from different sources, and copying memory out to the CPU. Each of these is implemented differently based on the memory type, and may not be implemented if the operation is impossible. For example, attempting to 'prefetch', i.e. move CUDA Unified Memory to the GPU before usage, is only supported for CUDA Unified Memory and not the other types.

### 6.3.2 `FrameAllocator`

`FrameAllocator<MType>` is an allocator associated with a single memory frame. The behaviour is specialized for each memory type, but each specialization fits a typeclass (Fig. 6.3.1) for allocating 2D and red-black arrays. The `FrameAllocator` owns these allocations, and when it is destroyed the allocations are freed. The CPU and CUDA Managed memory variants allocate data directly using their respective allocation functions when requested, store the raw pointers in a list, and frees all of them on destruction.

The Vulkan variant allocates a fixed amount of Vulkan device memory, which is exported using the CUDA-Vulkan interop API to a CUDA-compatible pointer.

All new allocations are then sub-allocated from this memory. The fixed amount is calculated by the `FrameSetAllocator`, which assumes only the bare minimum ($u, v, p, fluidmask$) requirements are allocated in Vulkan. The associated CUDA pointer is *not* compatible with Unified Memory, so cannot be paged to the CPU.

### 6.3.3 `FrameSetAllocator`

`FrameSetAllocator<MType, TFrame>` creates a set of `TFrame` objects which are each allocated using separate `FrameAllocator<MType>` objects. The simple CPU and CUDA variants simply construct $N$ `TFrames` using $N$ `FrameAllocators`. The Vulkan variant is more advanced as it has to check the `TFrame` exposes the correct data, calculate the amount of Vulkan data to allocate per frame, and expose this data by implementing a virtual `VulkanFrameSetAllocator` interface. The Vulkan variant also passes in a CUDA `FrameAllocator` with the Vulkan one to allow the other buffers to be allocated.

### 6.3.4 Usage in Other Layers

The Simulation layer instantiates `FrameSetAllocators` in the simulation Runners. The Backend typeclass (Fig. 6.4.1) requires each backend to implement a Frame class, and to take a list of Frame instances as an argument to their constructor.

The visualization uses the `VulkanFrameSetAllocator` interface to grab references to simulation memory, which it uses while rendering.

## 6.4 Simulation Layer

As shown in the Design section, the Simulation layer is split into generic Runners and multiple simulation Backends. Building a system that efficiently allowed Runners to be Backend-agnostic while remaining performant was nontrivial.

### 6.4.1 Runners

Runners had three major design constraints:

1. Runners should be generic with respect to the backends they implement.
2. The same Runner should have the same interface for each Backend it implements.
3. Virtual functions should be avoided where possible.

The immediate thought would be to implement a single Runner class, taking a virtual Backend class and using virtual functions to start each tick. This would meet (1) by using a virtual Backend interface, and (2) by using the same class for all Backends, but would violate (3) by forcing every Backend to use virtual function calls. Instead, a slightly more complex approach is taken.

Each Runner defines a virtual interface, such as `IFixedTimeRunner` for the fixed time runner. A templated implementation class `SimFixedTimeRunner<B>` is then instantiated for each compatible backend `B`. Each of these classes implements the virtual interface `IFixedTimeRunner`, but they call the simulation functions directly as they are templated on the backend type. This setup meets (1) by using a single templated implementation, (2) by implementing a virtual interface, and (3) by only using the virtual functions where absolutely necessary i.e. on the Runner itself. This allows `IFixedTimeRunner` implementations to define a single virtual function `runForTime(t)` instead of calling a virtual function every simulation tick.

```
typeclass BackendFrame {
    // Must have a constructor that takes an allocator
    BackendFrame(FrameAllocator<MemoryType> allocator);
}
typeclass Backend {
    // Must define a Frame class
    class Frame fits BackendFrame;
    // Must have a constructor
    Backend(std::vector<Frames>, FluidProperties, SimSnapshot);

    float findMaxTimestep();
    void tick(float timestep, int targetFrame);

    LegacySimDump dumpStateAsLegacy();
    SimSnapshot get_snapshot();
}
```

Figure 6.4.1: Backend typeclass

### 6.4.2 Backends

To allow the generic implementation described above, the Backend typeclass ensures all backends follow a consistent interface (Fig. 6.4.1). Five classes implement this typeclass, matching those described in Section 5.2:

- `NullSimulation`
- `CPUSimpleSimBackend`
- `CPUOptimizedSimBackend`
- `CPUOptimizedAdaptedSimBackend`
- `CudaBackendV1`

#### CPU Backends

Most of the simulation code for CPU backends has been directly copied from the original simulation[59][60]. Some templates and template specializations have been added for the Adapted backend, but for the most part, the simulation is unchanged. The backend classes simply wrap up this code to be compatible with the typeclass.

```
template<typename T>
using in_matrix =
    const T* const __restrict__;

template<typename T>
using out_matrix =
    T* const __restrict__;
```

(a) Matrix templates

```
__global__ void computationKernel(
    CommonParams config,
    in_matrix<float> inputs1,
    in_matrix<int> inputs2,

    out_matrix<float> output
);
```

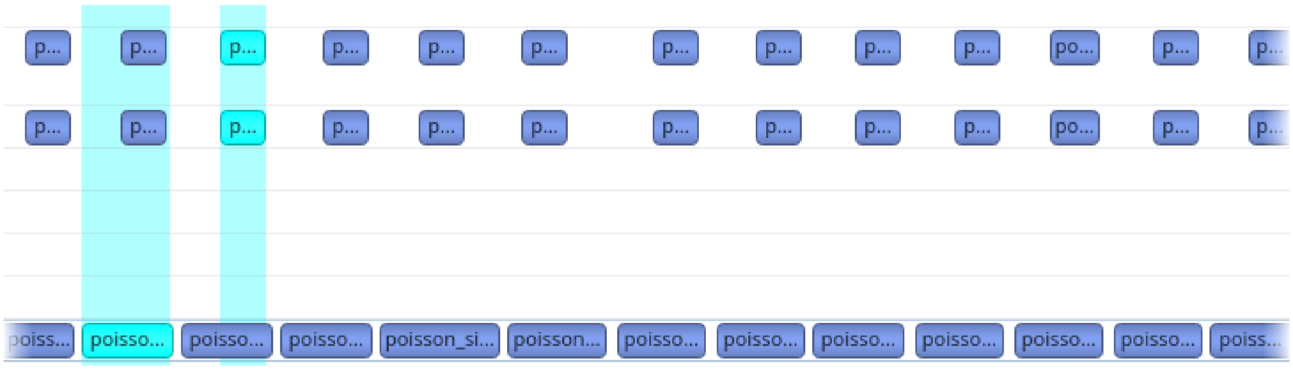
(b) A kernel using the matrix templates

Figure 6.4.2: Example of CUDA matrix templates

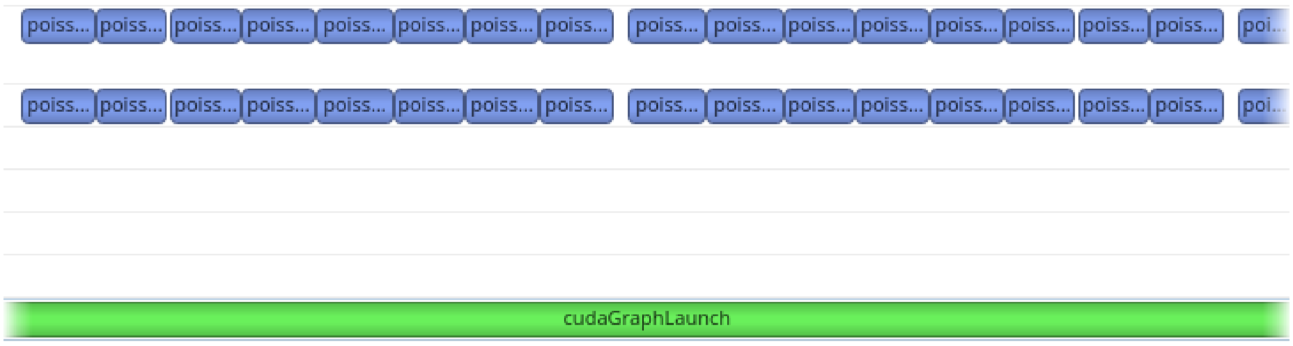
**CUDA Backend**

The CUDA backend is where the bulk of new simulation work has been done. As stated in the Design section, all simulation code has been ported to CUDA Kernels. Each of these kernels takes a `CommonParams` struct as the first argument, containing run-time constants such as the simulation grid size. To ensure `const __restrict__` pointers are used wherever possible, all other kernel arguments must be either an `in_matrix<T>` or an `out_matrix<T>`. These templates alias to simple pointers which properly use `const` and `__restrict__` (Fig. 6.4.2). Using these enforces that each kernel has clear, distinct inputs and outputs, and that the inputs are read from fast global memory.

The most intensive step in every implementation is the Poisson iterations, which are individually trivial but intensive at scale. During development the profiler showed large gaps between the individual kernel runs, equivalent to almost 50% of the runtime. Each kernel was finishing quicker than the CPU could enqueue a new one, so to solve this a CUDA Graph was employed. CUDA Graphs consist of a prerecorded set of kernel invocations with constant arguments. A CUDA Graph was recorded that consisted of 100 Poisson kernels, and this was launched instead of the individual kernels. This resulted in a 2x speedup in the profiler (see Fig. 6.4.3), but almost no speedup in practice. The CPU overhead for each enqueue may be larger in the profiler, which would explain why this behaviour is not present in the final simulation.



(a) Before CUDA Graphs



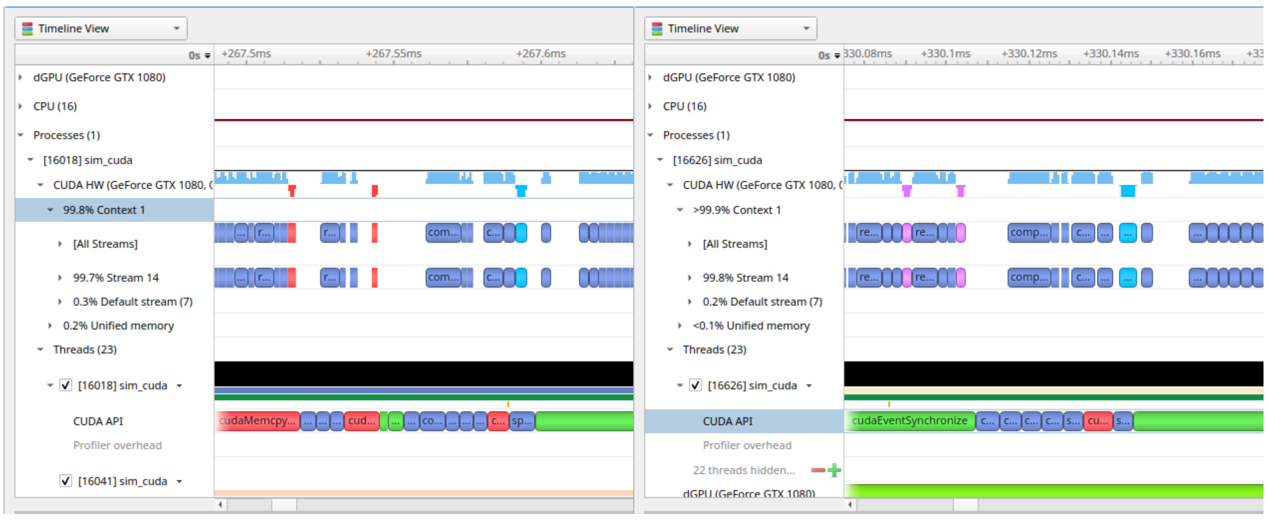
(b) After CUDA Graphs

Figure 6.4.3: Profiler traces of the Poisson kernels before and after CUDA graphs

The timestep calculation is implemented with two reductions, based on the second kernel from [24][4]. A constant factor N is chosen, and the values in the array are reduced by a factor of N multiple times until only one is left. Two data buffers are used to ping-pong the reductions - each iteration flips the input and output, so data is reduced from A to B to A and so on.

Because there are two reductions, it is most efficient to perform the first one asynchronously and enqueue both before waiting for them to finish. By default copying reduction results back to the CPU is synchronous, which prevents this. Allocating pinned memory, which cannot be paged to disk or moved around by the OS, allows the copy to be done asynchronously.

[4]This isn't the fastest kernel, but reductions aren't frequent enough for it to matter.



(a) Synchronous Copy (b) Asynchronous Copy

Figure 6.4.4: Using asynchronous copies for greater efficiency

### 6.4.3 Usage in Other Layers

The visualization layer instantiates a `VulkanTickedRunner` with `CudaBackendV1` to run a visualized simulation.

The command-line layer instantiates a `FixedTimeRunner` with any one of the backends to run a headless simulation.

## 6.5 Visualization Layer

### 6.5.1 Multithreading

The worker thread is implemented with a `SystemWorker` class combined with a generic threading system. An `IWorkerThread` virtual interface is defined, and then implemented by the `IWorkerThread_Impl<Worker>` template for a specific `Worker` class, similar to the Simulation Runners pattern.

To kick off the worker thread, a `WorkerThreadController` writes to a mutex-protected set of input data. The ‘work index’ of this data is incremented to signal it is new, and a condition variable is signalled to alert the worker thread and begin processing. Work cannot be enqueued until the thread produces an output, which is sent to the main thread in the same way as before - a mutex is taken to update the output data with the new index, and the condition variable is signalled in case the main thread is waiting for the worker to finish.

### 6.5.2 GPU Work Breakdown

Fig. 6.5.1 expands on the coarse GPU work breakdown from Section 5.3.3. Each rectangle represents a piece of memory, and each arrow represents a transformation from input to output via a compute shader, an image layout transfer, or a graphics pipeline. Most memory is global rather than per-frame, as the system does not run any visualization stages in parallel. Some per-frame buffers (highlighted in bold) are used to allow race-free accesses at record-time. These buffers allow user interaction, such as moving the particle emitters and setting the quantity ranges.

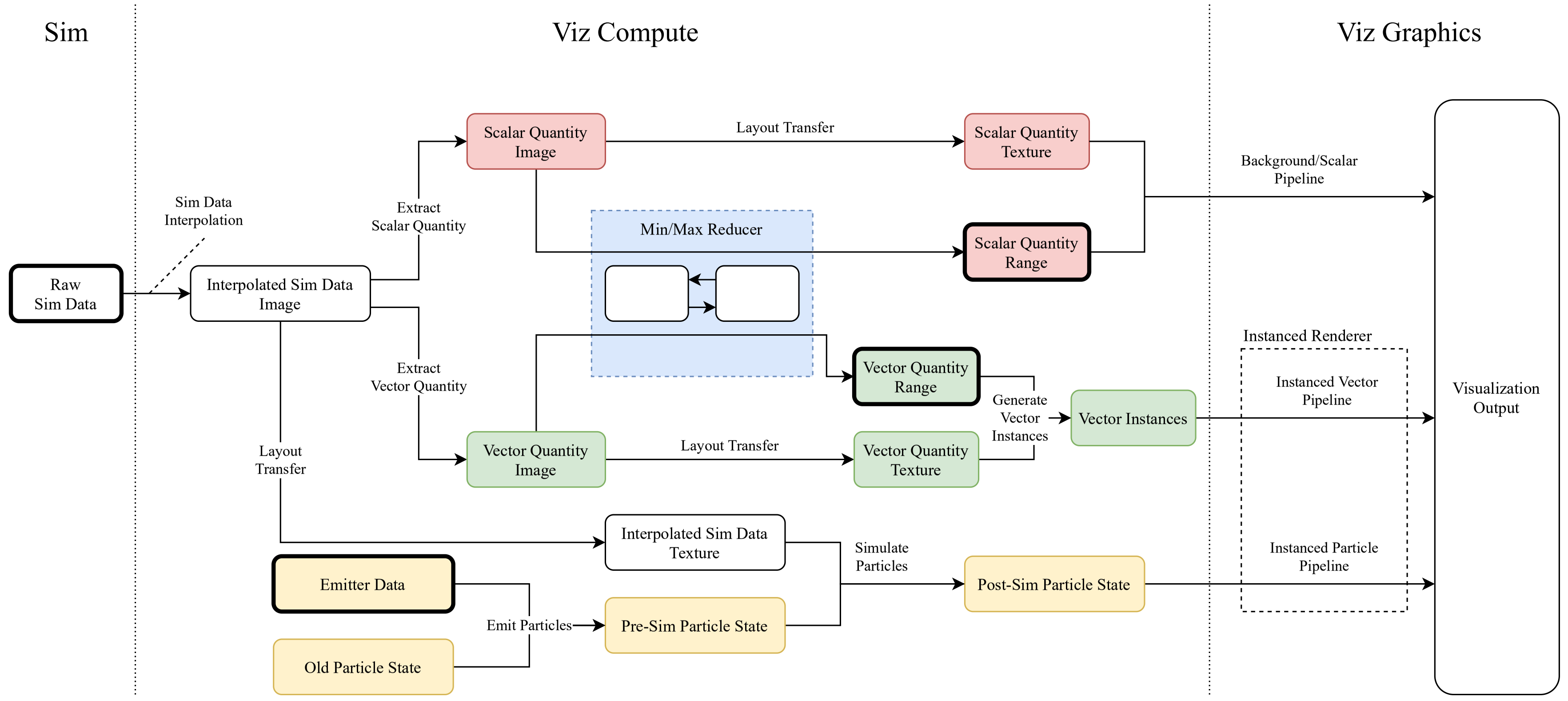


Figure 6.5.1: Data Transformation Diagram showing the data flow for the Visualization

```
uniform readonly image2D resultImage;
 // = (u, v, p, isfluid);

// Specify the exact pixel location
ivec2 pxIdx = ivec2(200, 450);
vec4 data = imageLoad(simDataImage, pxIdx);
```

(a) Directly

```
uniform sampler2D simDataSampler;
 // = (u, v, p, isfluid);

// 50% across, 20% up the image
vec2 sampleAt = (0.5, 0.2);
vec2 velocity = texture(simDataSampler,
↪  sampleAt).xy;
```

(b) With a Sampler

Figure 6.5.2: Reading from an image directly vs. using a sampler

Image layout transfers allow the GPU to optimize access times for an image by changing the format it's stored in. Images are transferred to a "read-only optimial" layout (listed as 'Texture' rather than 'Image' in Fig. 6.5.1) for efficient sampling at arbitrary points, and kept in the "General" layout when accessed at 2D data arrays (see Fig. 6.5.2 as a comparison).

Memory barriers (not shown in Fig. 6.5.1) are inserted between every compute shader to ensure any required data written from a previous shader is visible to the next shader[56]. These memory barriers are quite granular, as shown in Fig. 6.5.3.

```
// Make ShaderWrites from the ComputeShader stage available + visible to
//      IndirectCommandReads in the DrawIndirect stage
fullMemoryBarrier(computeCmdBuffer,
    vk::PipelineStageFlagBits::eComputeShader, vk::PipelineStageFlagBits::eDrawIndirect,
    vk::AccessFlagBits::eShaderWrite, vk::AccessFlagBits::eIndirectCommandRead);
// Make TransferWrites from the Transfer stage available + visible to the
//      ShaderReads in the ComputeShader phase.
fullMemoryBarrier(computeCmdBuffer,
    vk::PipelineStageFlagBits::eTransfer, vk::PipelineStageFlagBits::eComputeShader,
    vk::AccessFlagBits::eTransferWrite, vk::AccessFlagBits::eShaderRead);
```

Figure 6.5.3: Example showing the granularity of Memory Barriers

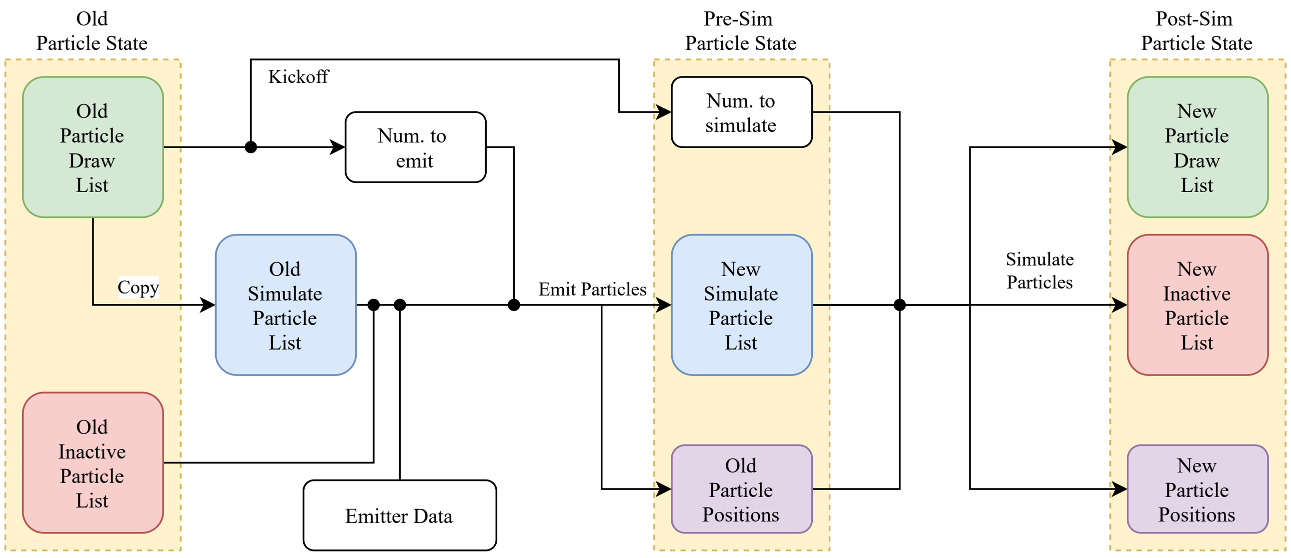


Figure 6.5.4: Breakdown of particle-related GPU work

The particle system implementation in Fig. 6.5.1 is a simplified view for compactness, Fig. 6.5.4 shows a full breakdown of this subsystem. This maintains three growable/shrinkable lists, plus a buffer containing particle positions.

1. The Draw list, a list of particle indices to draw on screen
2. The Inactive list, a list of inactive particle indices
3. The Simulate list, a list of particle indices which take part in Simulation.

The previous Draw list is the authority on which particles currently exist, and is used for the Kickoff shader to determine how many particles will be emitted/simulated[5]. It is also copied into the Simulate particle list, which is grown by the Emit Particles shader. The particles are then moved by the Simulate Particles shader as shown in Section 2.3.5. These particles are added to the inactive list if out-of-bounds, and the new Draw list otherwise.

### 6.5.3 Safe CPU/GPU Communication

Unlike CUDA, the Vulkan API does not provide any means of type-safety when communicating between the CPU and GPU. If the GPU expects data in a specific structure, it is the CPU's job to create data that fits this structure. A naive solution might be to keep a C++ structure definition and a GLSL structure definition, and assume that one matches the other. This is error-prone as the structures are not automatically kept in sync - if one changes, the other will not, and communication

[5]This isn't known at record time, because the last frame may still be simulating the particles

will break down. This project's approach is to create a GLSL file defining all interoperable structures (`global_structures.glsl`), and then include it into a C++ header with some extra code to define GLSL types correctly. Both sides will now use the same structure definitions, which are all defined in exactly one place. All GLSL code uses the `std430` memory layout rules, which closely matches the C++ memory layout, so the structures can be passed directly from the CPU to the GPU safely.

### 6.5.4 Usage in Other Layers

The `VulkanSimApp` class is instantiated by the command-line layer to run the visualization.

## 6.6 Command-Line Layer

The command-line layer is implemented with a set of "sub-app" classes, each implemented by a separate virtual class satisfying an interface `ISubApp`. Virtual inheritance was chosen here because it is convenient and not in a performance-critical area. Each `ISubApp` instance is used to create a CLI11 subcommand with some input arguments, then CLI11 parses the command-line arguments and runs a callback on the selected sub-app. These sub-apps then invoke other layers of the system to complete their execution.

# CHAPTER 7

# Project Management

To ensure a smooth development process, all research and implementation was planned ahead of time. This chapter details these plans including a complete schedule, the development methodology used, the tools used, and the potential risks and associated contingencies.

## 7.1 Software Development Methodology

Plan-driven solutions depend on a rigid specification being completed before development[82], which did not fit with the more abstract goals of the visualization portion. Additionally, some of the main advantages of plan-driven approaches only apply when introducing new team members and handling large teams. Neither scenario applies here, as only one person is undertaking active development. For these reasons, an Agile approach was taken with a development cycle completing every two weeks. The goals for each development cycle were documented using Trello.

It was planned that the supervisor would be contacted every week with the current status of the project and the progress made in the current cycle. These contacts would either take place over e-mail if there were no pressing questions to ask, and otherwise take place on Microsoft Teams. Unfortunately, this did not happen for the first few weeks, as other work was vying for attention and preventing project work from taking place. This was resolved in Week 5, and from then on there was frequent email correspondence.

## 7.2 Project Timeline

The project was split into multiple tasks to schedule it effectively. These tasks are scheduled on both a Gantt Chart in Fig. 7.2.1, and as a table in Table 7.2.1. The timeline has been well followed, and this schedule has been left unchanged over the course of the project.

No programming was scheduled over the Christmas break to allow time to be spent on other assignments. The development of the visualization was scheduled concurrently with optimizing the simulation, in case some strides in visualization required extra optimizations to run in real-time. While not strictly required, some optimizations were developed in this time to push performance further. A Code Freeze was set for Week 22, to focus the researcher entirely on the presentation.

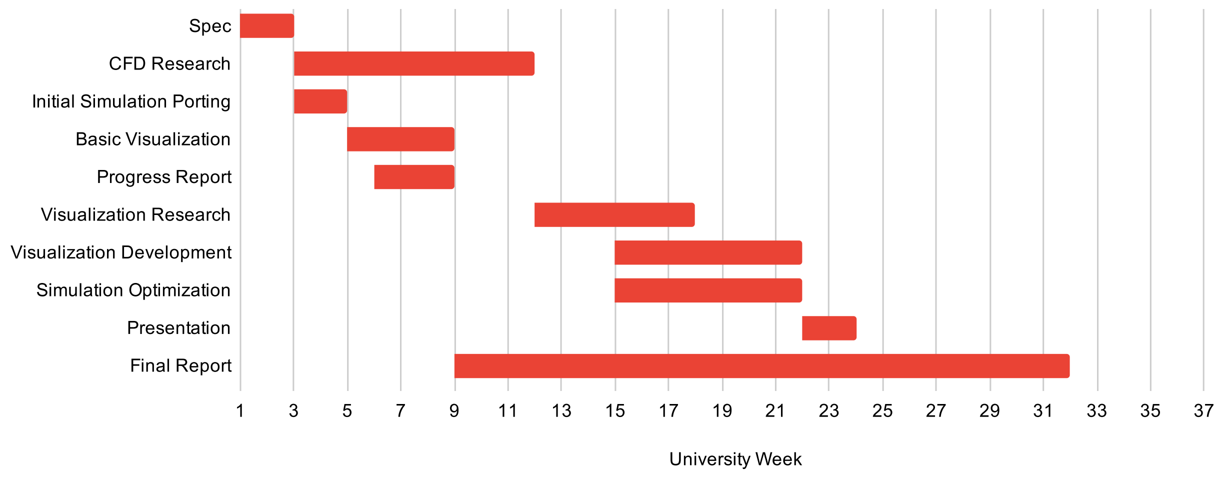


Figure 7.2.1: Project Schedule as a Gantt Chart

## 7.3 Tools

gcc 8 was used to compile the program. This version had stable support for the C++14 and C++17 standards, allowing modern techniques to be used in the program. CMake was used to handle building the program source files. Versions 3.8 and up support CUDA as a first-class language, which simplified the compilation process.

Git was used for source control, synchronized to a private GitHub repository to avoid data loss. The researcher used the CLion IDE to develop the program, which

| Task | Start Week | End Week |
|---|---|---|
| Spec | 1 | 3 |
| CFD Research | 3 | 12 |
| Initial Simulation Porting | 3 | 5 |
| Basic Visualization | 5 | 9 |
| Progress Report | 6 | 9 |
| Visualization Research | 12 | 18 |
| Visualization Development | 15 | 22 |
| Simulation Optimization | 15 | 22 |
| Presentation | 22 | 24 |
| Final Report | 9 | 32 |

Table 7.2.1: Project Schedule Tasks

simplified building the program and interacting with source control.

LaTeX was used to create the various reports and non-program deliverables required by the project, which were hosted on Overleaf so they could be compiled on Windows and Linux without installing a LaTeX environment.

Trello was used to track bugs and upcoming features in each development cycle. Google Drive was used to host other documents, e.g. scanned notes, that were created during development.

## 7.4 Risk Management

When progressing through the project, there were risks that could impede progress and even prevent the project from succeeding. Being aware of these risks allowed them to be predicted ahead of time, avoided, or in the worst case mitigated once they arrived. Risk can be calculated with the following equation, where Severity and Likelihood are graded between 1 and 5.

$$Risk = Severity * Likelihood$$

Of these risks, Illness and Other Pressures were encountered during development. Both were mitigated quickly and did not cause a large delay.

### 7.4.1 Misscheduling

It may have been possible that the features outlined in Chapter 4 were too great to be implemented in the allotted time. In that case, the quality of work could have to be reduced to meet deadlines, or the schedule would need to be changed. This is especially relevant to the Visualization portion of the project, which was not fully planned until the research was completed.

**Risk** = 2 * 2 = 4

**Avoidance:**

Previous projects were used as a reference to predict how long implementing features will take, and inform the schedule. As new Visualization features were discussed, the impact on scheduling they each have were considered.

**Contingency:**

The scope of the project could have been reduced to allow the report to be completed in time. A "code freeze" was implemented close to the presentation deadline to ensure enough time is spent polishing the presentation and report.

### 7.4.2 Other Pressures

While the project schedule may have been well estimated based on the work required for the project, the amount of work required for other modules was larger than expected. This manifested in Term 1, where the researcher took more modules than usual. Additionally, the removal of in-person lectures due to COVID-19 led to a lack of overall structure, which made organizing the other work more difficult. This did not impact the schedule.

**Risk** = 2 * 1 = 2

**Avoidance:**

This could have been avoided by better balancing the modules between Term 1 and Term 2, but on the flip-side having fewer modules in Term 2 allowed for more project work to be completed.

**Contingency:**

As before, the scope of the project could have been reduced to allow the report to be completed in time. If module work took more time than expected by week 20, the code freeze could have been pulled forwards to week 20 or 21 to spend more time on the presentation.

### 7.4.3 Loss of Hardware Access

As noted in Section 4.3, a GPU is required for the project to be tested and developed. The main development environment was the researcher's personal computer, which has a suitable GPU. However, if this computer were to break down or be stolen, there was no readily available alternate environment. Under normal circumstances, the Department of Computer Science labs would be used instead, as they also have suitable GPUs, but the virus situation prevented this.

**Risk** = 5 * 1 = 5

**Avoidance:**

Not possible.

**Contingency:**

Student insurance could have been used to purchase a new GPU/computer if it is stolen. Failing this, the DCS clusters could be used, but these would likely have high contention from other students who need to use GPUs remotely.

### 7.4.4 Illness

It is always prudent to consider the possibility that the stakeholders may fall ill and be unable to work on the project for some time. This was exacerbated by the situation with COVID-19, making potential illnesses more dangerous than usual.

This risk manifested during Week 7 and delayed work on the project by three days. However the bulk of the current work had been completed by that point, so this module was not affected.

**Risk** = 4 * 2 = 8

**Avoidance:**

Not possible.

**Contingency:**

The schedule would need to be changed to account for the lack of time spent working. Some requirements could be reduced or removed entirely.

CHAPTER 8

# Testing & Success Measurement

In order to measure the degree of success a project achieves, testing must be performed to verify the behaviour of the program is correct. This covers testing the functionality of individual units of the program (unit testing), testing how those units interact with each other (integration testing), and validating the behaviour of the overall system against the functional and non-functional requirements[83]. This section also defines the means of Success Measurement for some non-functional requirements, which are then measured and evaluated in subsequent sections.

## 8.1 Unit Tests

The first layer of testing splits the program into 'units', that are independent of each other, which are individually tested before combining them with other units in the system. In some systems, it is practical to automate these tests, but this was not pursued for this system as the behaviour is generally too complex to be automatically verified.

Helpfully the program is already split into subcommands at the command-line level (Section 5.4), which can all be tested individually. Because the file format is the same as the original simulation, the original input file can be used to test comparisons (`compare`), simple visualization (`renderppm`), and both simulations (`fixedtime` and `run`). These commands all have equivalents in the original program, which provides a basis for validating correctness. The `makeinput` subcommand, which creates a new input file based on an image, can be tested by passing the resulting input file to other known-functional subcommands and checking their behaviour. This provides a coarse view of system functionality, but a finer level of detail can be obtained by testing individual code components.

Unit-testing this particular codebase is difficult because many components are dependent on other components - for example, the visualization components use data gathered from the simulation output, which is impractical to extract for the sake of testing individual components. It is easier to just test the complete visualized simulation while assuming the simulation itself is correct. In other cases, unit behaviour may be impractical to directly model or verify: the automated resource management classes are difficult to test individually as the creation/destruction of the resources they manage cannot be directly checked. However, there are some areas where the codebase can be effectively unitized, the most prominent of which is the simulation itself.

The simulation is split into stages, which are effectively independent code units. Each unit depends on the output of the previous unit, so they cannot be tested independently, but if the rest of the simulation units are known to be correct then an individual unit can be tested. This technique was used during development to ensure the CUDA simulation was consistent with the CPU version.

Overall, while unit tests are not always suitable for elements of the codebase, they are helpful at a coarse level. The final set of unit tests are shown in Table 8.3.1.

## 8.2 Integration Testing

Once the program units have been individually tested, the Integration Tests in Table 8.3.2 test that the units can interface with each other correctly. Again the subcommands are treated as units, and testing is performed by passing the output from one subcommand as the input to another. In this program only the `makeinput` and `fixedtime` subcommands produce output, so their output is exhaustively tested against the other commands. At the codebase level some previous unit tests can be counted as integration tests: the headless simulation functions as an integration test for the Memory and Simulation layers, and the visualized simulation tests the integration between the Simulation and Visualization layers.

The C++ type system ensures that low-level connections between CPU code use the correct types, making integration testing at this level redundant. Moving data between the CPU and GPU is more complicated, but the elements put in place in

Section 6.5.3 with the Vulkan validation layers (Section 6.2) ensure that any integration errors are caught when simply running the program in Debug mode.

## 8.3 System Testing

This final layer of testing determines if the program upholds the functional and non-functional requirements set out in Chapter 4. Some functional requirements such as Req. F5.1 require in-depth checks of the visualization not suitable for other layers, so are tested here. The System Tests specified in Table 8.3.3 cover all such tests, and combined with the previous layers prove that the system meets the functional requirements.

Many non-functional requirements can be tested directly: Req. NF2 is tested with external programs `valgrind` and `cuda-memcheck`, Reqs. NF9 and NF10 are tested by simply inspecting the program and source files, Req. NF11 is tested by attempting to compile + run the program on DCS systems. A few require a large amount of data collection or at least careful attention to detail. These tests are specified in Table 8.3.4. All gathered data is specified in the next subsection, and then the results are shown in Chapter 9.

### 8.3.1 Success Measurement

Some non-functional requirements require more in-depth testing. These tests are planned here, and the results are shown in Chapter 9. The results are then evaluated in Chapter 10 along with the rest of the test outcomes.

Tests T27 and T28 test the memory usage of the system. The most important kind of error they can detect are memory leaks, where memory is allocated without being released, leading to the program taking up memory it does not need anymore. While it is important to avoid memory leaks in all cases, the most important variations are continuous memory leaks, where memory is continuously leaked over and over, and large leaks of sizes larger than 100MB. Single small leaks are less concerning, as they should not impact the rest of the system greatly. The programs used to find these leaks may themselves be bad at recognizing allocation/freeing[84], and lead to false positives, so care must be taken when evaluating their results.

Tests T30 and T31 evaluate the speed and accuracy of the CUDA-based simulation backend vs. the adapted CPU backend. The accuracy is measured by comparing the output of equivalent simulations on the original simulation input state. Other states were tested, but any newly generated states with obstacles proved to be unstable and produce Not-a-Number outputs on both backends. The simulation speed is tested on the original simulation input state, then behaviour at scale is tested on custom generated states with no obstacles. Obstacle configuration does not affect time-per-tick, so time-per-tick will be a representative value and equal for any state of the same size. This would not be suitable for accuracy tests, as having no obstacles greatly reduces the complexity and would likely produce disproportionately high accuracy values.

As there is no visual component to the simulation tests, the program is run from a terminal without running the X windowing system. Combined these tests should give a complete picture of how the CUDA simulation's speed and accuracy will scale, and be enough to evaluate the requirements in context.

Test T33 evaluates the speed of individual visualization features vs. the simulation. As the time taken to run a simulation tick can be variable based on the input, these visualization speeds are compared to the time allotted to the target 60FPS, i.e. 16.6 ms. These visualization times are measured using the frame-time counter in the GUI (Fig. 5.3.4), which measures the average time taken to present the last 32 frames. First, the time taken to render a frame with no visualization features is taken. Each feature is then individually enabled, brought to the worst-case scenario (using auto-range where applicable, and rendering the maximum amount of instances where applicable), then the average frame-time is taken. To ensure the results are not affected by external sources, the program is run in unlocked framerate mode with no other programs running on the system. This is also the case for the GPU utilization test (Test T29).

| ID | Description | Expected | Output | Result |
|---|---|---|---|---|
| T1 | `compare`: identical states | No difference | No difference | ✓ |
| T2 | `compare`: Original input to original target output | Some difference | Some difference | ✓ |
| T3 | `renderppm`: render state vorticity | Equal to original program | Equal to original program | ✓ |
| T4 | `makeinput`: generate an input file from a PNG | Valid simulation state | Valid simulation state | ✓ |
| T5 | `fixedtime`: simulate from an input state for 25 seconds. | Valid simulation state | Valid simulation state | ✓ |
| T6 | `run`: visualize a simulation from an input state for 25 seconds. | Valid simulation state | Valid simulation state | ✓ |

Table 8.3.1: Unit Tests

| ID | Integrated Modules | | | Output | Expected | Result |
|---|---|---|---|---|---|---|
| T7 | `makeinput` | → | `renderppm` | Valid render image with the same obstacle squares as the initial image. | As expected | ✓ |
| T8 | `makeinput` | → | `compare` | `compare` runs successfully | As expected | ✓ |
| T9 | `makeinput` | → | `fixedtime` | Valid simulation output with the same obstacle squares as the initial image. | As expected | ✓ |
| T10 | `makeinput` | → | `run` | Visualization of a simulation with the same obstacle squares as the initial image. | As expected | ✓ |
| T11 | `fixedtime` | → | `renderppm` | Valid render image with the same obstacle squares as the initial state. | As expected | ✓ |
| T12 | `fixedtime` | → | `compare` | `compare` runs successfully | As expected | ✓ |
| T13 | `fixedtime` | → | `fixedtime` | Valid simulation output with the same obstacle squares as the initial state. | As expected | ✓ |
| T14 | `fixedtime` | → | `run` | Visualization of a simulation with the same obstacle squares as the initial state. | As expected | ✓ |

Table 8.3.2: Integration Tests

| ID | Description | Expected | Output | Result |
|---|---|---|---|---|
| T15 | `fixedtime`: GPU Simulation | Simulation backend can be set to CUDA | As expected | ✓ |
| T16 | `run`: GPU Simulation | Simulation backend can be set to CUDA | As expected | ✓ |
| T17 | `run`: Test pausing/resuming the simulation | Simulation can pause/resume while the visualization is running | As expected | ✓ |
| T18 | `run`: Test saving the simulation state | Simulation state can be saved while visualizing | Couldn't save state while running | ✗ |
| T19 | `run`: Test moving the particle emitters | Particle emitters can be moved while the simulation is running | As expected | ✓ |
| T20 | `run`: Can run with a fixed framerate | Framerate can be fixed at some value | Framerate was fixed at 120FPS and did not change | ✓ |
| T21 | `run`: Can run with an unlocked framerate | Framerate can be unlocked | Framerate was not locked and varied between 750-800FPS | ✓ |
| T22 | `run`: All Viz layers work as expected. | All layer combinations can be used, all layers function as described in Chapter 4. | As expected | ✓ |
| T23 | `run`: Test auto-range functionality | Auto-ranged Scalar and Vector quantities display all values in the sim boundary. | As expected | ✓ |
| T24 | `run`: Test changing colors | All colors used in the simulation should be modifiable | As expected | ✓ |
| T25 | `run`: Check Vulkan validation | No Vulkan validation errors in Debug mode | As expected | ✓ |

Table 8.3.3: System Tests (Functional)

| ID | Description | Expected | Output | Result |
|---|---|---|---|---|
| T26 | `run`: Large Simulation | Simulation can run on a 4096x4096 input. | As expected | ✓ |
| T27 | `run`: No CPU memory leaks | Visualization run under `valgrind` should have no program-controlled memory leaks. | See Sections 9.1.4 and 9.2.3 | ✓ |
| T28 | `run`: No CUDA memory leaks | Simulation run under `cuda-memcheck` should have no program-controlled memory leaks. | See Sections 9.1.4 and 9.2.3 | ✓ |
| T29 | `run`: High GPU Utilization | Nsight Systems profiler output should show maximum achievable GPU utilization[a] | See Sections 9.1.3 and 9.2.2 | ✓ |
| T30 | `fixedtime`: Simulation is faster than original simulation | CUDA simulation should run 2x faster than the original simulation on the original input. | See Section 9.1.1 | ✓ |
| T31 | `fixedtime`: Simulation produces similar results to original simulation | CUDA simulation solver residual should be within 5% of adapted CPU backend. | See Section 9.1.2 | ✓ |
| T32 | `run`: Can run at high framerate | Visualization can run at >30FPS in some case | Simulating the original input at N=100 runs at 800FPS. | ✓ |
| T33 | `run`: Visualization features are faster than simulation | See Section 8.3.1 | See Section 9.2.1 | ✓ |
| T34 | Try to compile on DCS systems | Should be able to compile and run the simulation on elements of the DCS system. | Successfully compiled non-CUDA sim on DCS. | ✓ |

Table 8.3.4: System Tests (Non-Functional)

[a] 100% may not be possible due to other programs using the GPU

# CHAPTER 9

# Results

Some non-functional requirements are based on comparisons to the original program or on tests that require more detailed analysis. This section shows the data used to evaluate these more complex requirements. The speed of the simulation is compared to the original, and the performance scaling with simulation grid size is measured and justified. Simulation accuracy vs. the original is evaluated. GPU Utilization of both the simulation and visualization is measured, and shown to be as high as possible given the design. Multiple programs are used to detect memory leaks, and show the program is leak-free. These results are then used in the Evaluation (Chapter 10).

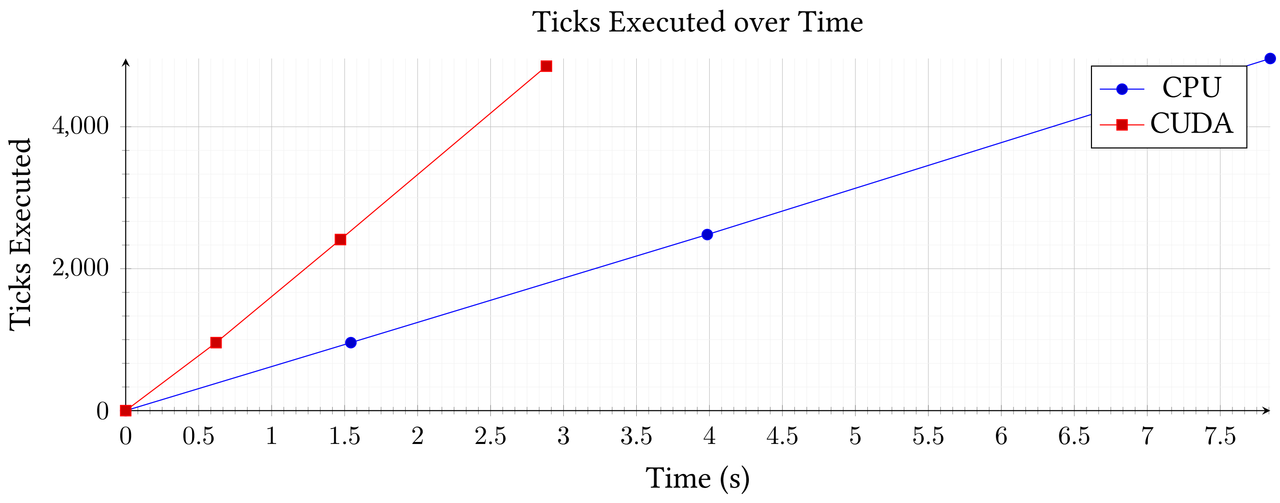


Figure 9.1.1: Ticks executed over time

## 9.1 Simulation

This section investigates four properties of the headless simulation: speed, accuracy, GPU utilization, and memory leaks. Some of these properties affect the visualization, as it runs a simulation internally. This is discussed in Section 9.2.

### 9.1.1 Speed

Each tick of a simulation performs the same amount of work, so it was hypothesized that the number of ticks per second would be a suitable metric for measuring simulation speed. To verify this, the original simulation input was simulated with the CUDA and CPU backends for 10, 25, and 50 simulation-seconds with $N_{Poisson} = 100$. As shown in Fig. 9.1.1, elapsed real-time is directly proportional to the number of ticks executed with R-values of $R^2_{CPU} = 0.9999$ and $R^2_{CUDA} = 0.9996$. This proves that ticks-per-second can be used to accurately compare CUDA/CPU performance.

The ticks-per-second for CPU and CUDA are shown in Fig. 9.1.2, for $N_{Poisson}$ values of $100, 200, 300, 1000$. As expected the ticks-per-second decreases as more iterations are added, and CUDA is consistently faster than the CPU. When normalized relative to the CPU speed (Fig. 9.1.3), CUDA is shown to consistently have ~2.6x the ticks-per-second of the CPU, i.e. CUDA is consistently 2.6x faster than the CPU. The Poisson stage is likely less sped up than the others, as the speedup for $N_{Poisson} = 1000$ (where the Poisson stage dominates) is less than that for lower $N_{Poisson}$ values. When running at $N_{Poisson} = 300$ the speed is close to the CPU speed for $N_{Poisson} = 100$, so the original goal of "using the speedup to increase simulation accuracy" can be met.

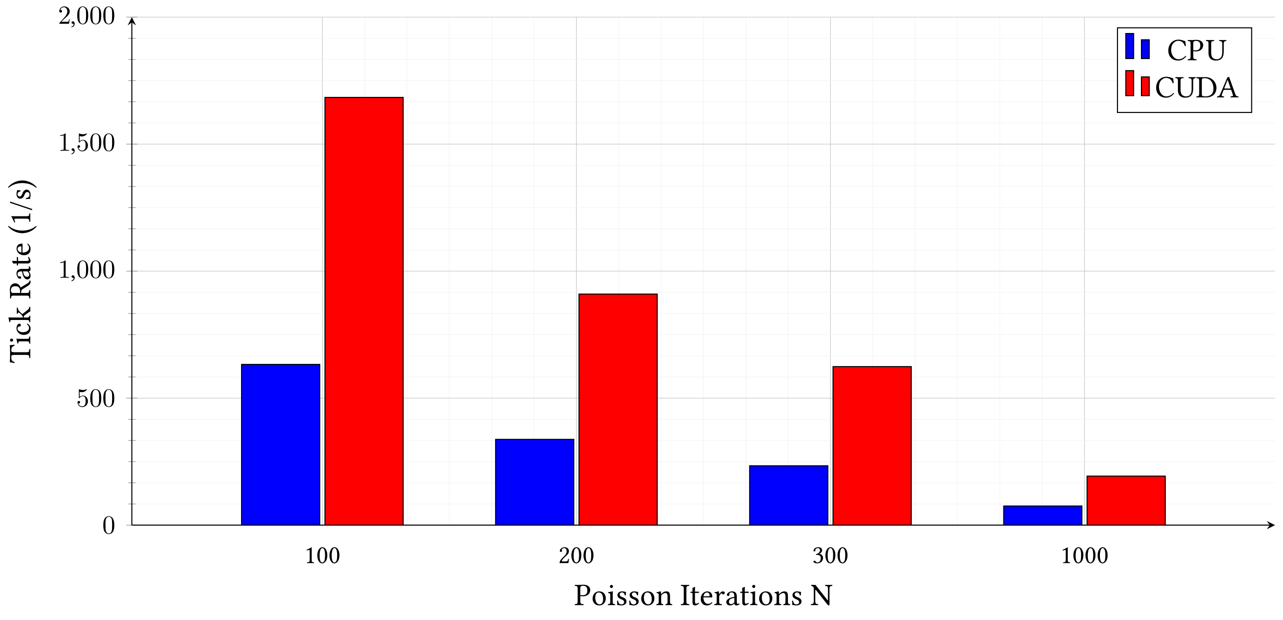


Figure 9.1.2: Simulation Tick Speed vs. Poisson Iterations

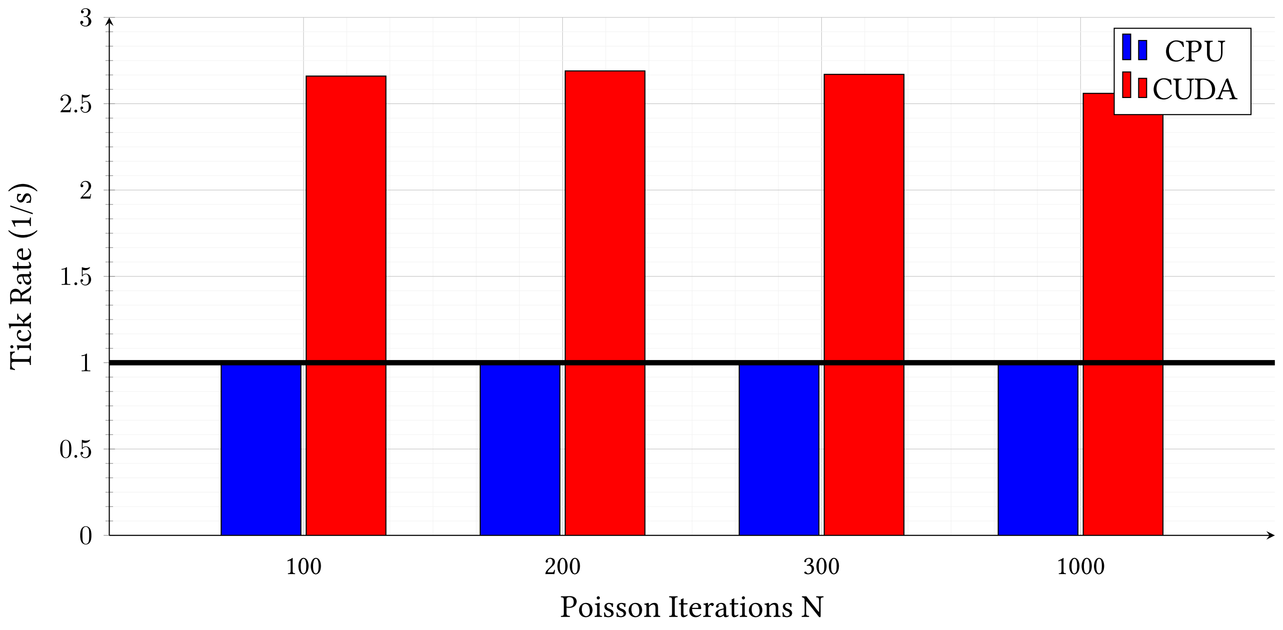


Figure 9.1.3: Simulation Tick Speed relative to CPU

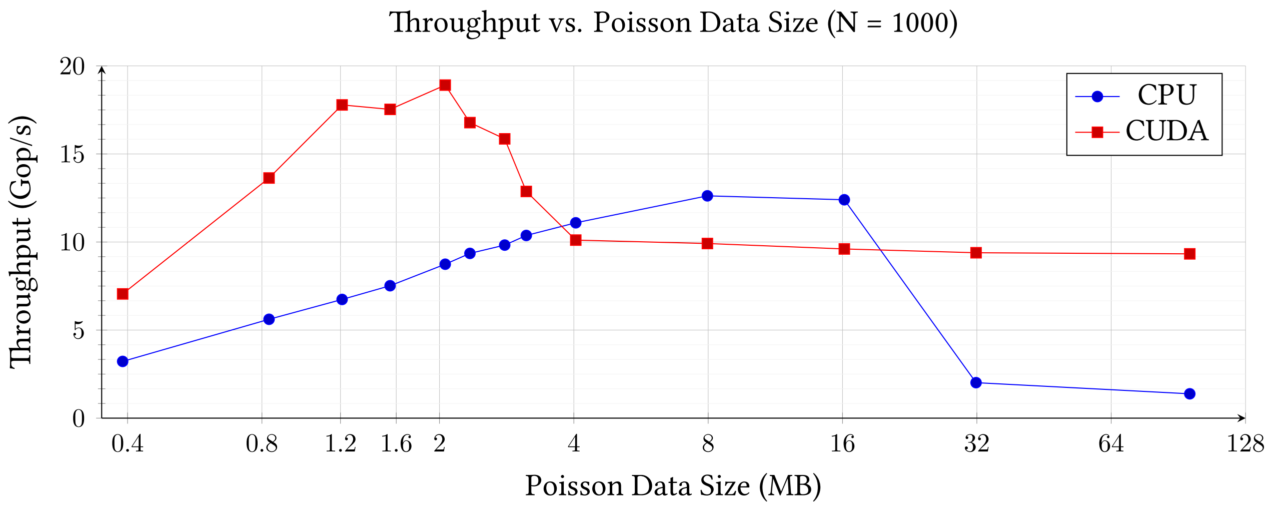


Figure 9.1.4: Simulation Throughput for CPU and CUDA vs Poisson cache size

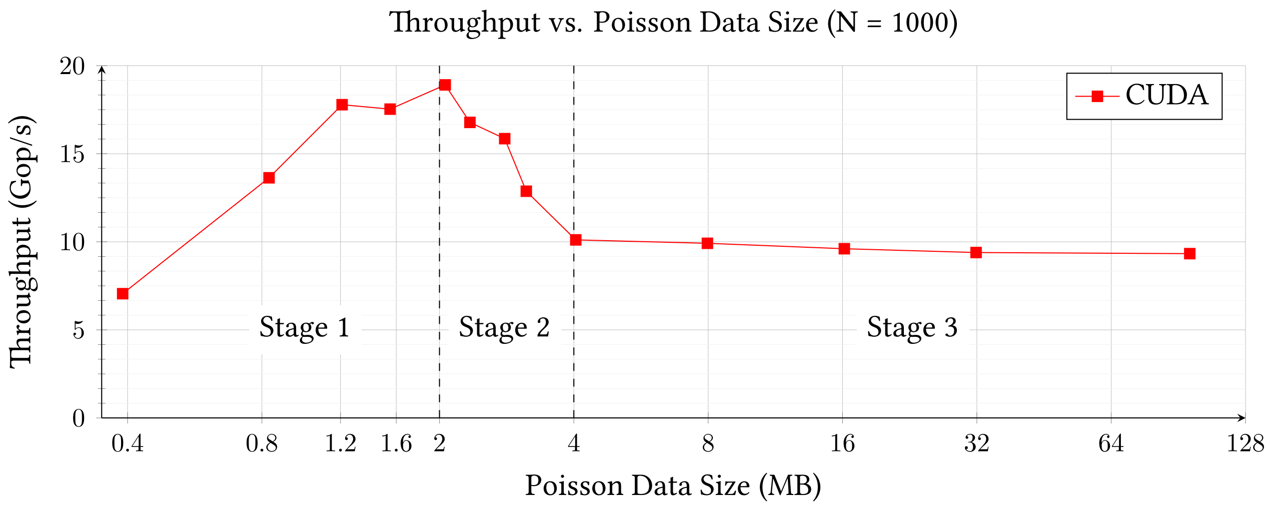


Figure 9.1.5: Simulation Throughput for CUDA, split into stages

**Scaling**

To determine how the implementations scale with grid resolution, a simple simulation with no obstacle squares was performed for resolutions between 260 x 130 and 4096 x 2048. As the amount of work per simulation tick varies with grid size, each tick is multiplied by the number of grid cells and number of Poisson iterations to determine the number of operations (ops) executed, and giga-operations-per-second (Gop/s) is used to measure performance. The increasing size is measured in the total amount of memory (in MB) required for a full Poisson red/black iteration, equal to three full-size matrices: the pressure matrix, the $\beta$ matrix, and the $rhs$ matrix. Each simulation simulated 25 s of simulation time with $N_{Poisson} = 1000$. Fig. 9.1.4 shows Gop/s for CUDA and CPU against the memory required in MB.

| Grid Size | Physical Size (m) | Poisson Data Size (MB) | Total Cells | CUDA Threads per Colour Stage |
|---|---|---|---|---|
| 260x130 | 20.0 x 10.0 | 0.39 | 33,800 | 16,900 |
| 380x190 | 29.2 x 14.6 | 0.83 | 72,200 | 36,100 |
| 460x230 | 35.4 x 17.7 | 1.21 | 105,800 | 52,900 |
| 520x260 | 40.0 x 20.0 | 1.55 | 135,200 | 67,600 |
| 600x300 | 46.2 x 23.1 | 2.06 | 180,000 | 90,000 |
| 640x320 | 49.2 x 24.6 | 2.34 | 204,800 | 102,400 |
| 700x350 | 53.8 x 26.9 | 2.80 | 245,000 | 122,500 |
| 740x370 | 56.9 x 28.5 | 3.13 | 273,800 | 136,900 |
| 840x420 | 64.6 x 32.3 | 4.04 | 352,800 | 176,400 |
| 1180x590 | 90.8 x 45.4 | 7.97 | 696,200 | 348,100 |
| 1680x840 | 129.2 x 64.6 | 16.15 | 1,411,200 | 705,600 |
| 2360x1180 | 181.5 x 90.8 | 31.87 | 2,784,800 | 1,392,400 |
| 4096x2048 | 315.0 x 157.5 | 96.00 | 8,388,608 | 4,194,304 |

Table 9.1.1: Throughput Measurement Points

Initially, the biggest surprise in Fig. 9.1.4 was the CPU performing better than CUDA between 4-16 MB. Other interesting points were the linear increase in CUDA throughput between 0.4-1.2 MB, and the performance plateau from 4 MB onwards (CUDA) and 32 MB onwards (CPU). These are due to multiple factors, which are best explained by separating the CUDA graph into stages (Fig. 9.1.5).

The first stage shows Gop/s increasing almost linearly before plateauing at 1.2-1.5 MB. This is due to the GPU's parallelization not being fully utilized. This data was measured on a GTX 1080, which has 2560 CUDA cores separated into 20 Streaming Multiprocessors (SMs) of 128 cores each[37]. Each SM can execute 2048 threads at once[49], thus the GPU can execute at most 40,960 threads in parallel. The first data point uses a 260 x 130 grid with only 33,800 cells, thus 16,900 threads per Poisson colour stage[1], so the GPU isn't saturated and isn't producing as many outputs as possible per second. This is also the case for the second data point, with 36,100 threads per colour stage. Beyond this point, the GPU is saturated and the throughput plateaus at 2.6x the CPU value.

The throughput decreases after 2 MB, which is the size of the GTX 1080 L2 cache[37]. As the size increases beyond this point the chances of a memory read being present in the cache decreases, so requests to main memory are made more frequently. Main memory is much slower than L2 cache so the throughput decreases until the data is 2x the cache size (4 MB), at which point all accesses are to

[1]The red and black stages only write to the 1/2 of the grid corresponding to their colour.

main memory and the throughput plateaus at 10 Gop/s. This behaviour can also be seen with the CPU, an AMD Ryzen 7 1800X with 16 MB of L3 cache, which peaks at 16 MB then drops at 32 MB onwards. Initially when the GPU performance drops it falls below the CPU performance, but once the CPU performance falls the GPU ends up being 6x faster.

Caching is vital to performance, and this program is no exception. Adapting the implementation to better utilize cache is crucial to improving simulation speed at scale, and would be excellent to investigate this as future work.

### 9.1.2 Accuracy

Initially, accuracy was measured by comparing the Mean Square Error (MSE) between CPU and CUDA outputs after equivalent simulations (see Table 9.1.2). Ideally, the CUDA and CPU results would be similar, and the MSE would be low (perhaps around $10^{-14}$, as expected in Section 5.4.3.) Instead, the results are quite different - the velocity is quite similar after 10 s, but that delta increases to $10^{-6}$ as the simulation progresses. Pressure is even worse, starting at $10^{-8}$ and going as far as a mean square error of $10^0 = 1$. In both cases, the divergence between CPU and CUDA increases as more iterations are performed (see Figs. 9.1.6a and 9.1.6b). However, this isn't the whole story. Measuring the difference between the CPU and CUDA doesn't show which one is more accurate.

| | Velocity | | | | Pressure | | | |
|---|---|---|---|---|---|---|---|---|
| Time (s) | 100 | 200 | 300 | 1000 | 100 | 200 | 300 | 1000 |
| 10 | −14.12 | −14.01 | −13.46 | −13.02 | −8.14 | −7.71 | −7.01 | −5.18 |
| 25 | −5.93 | −5.93 | −5.93 | −5.93 | −5.36 | −4.14 | −3.33 | −0.91 |
| 50 | −6.41 | −6.34 | −6.33 | −6.29 | −5.02 | −4.21 | −3.11 | −0.13 |

Table 9.1.2: Log of Mean Square Error between CPU and CUDA results (Numbers closer to 0 are worse)

The true measure of accuracy for a differential equation solver is the precision of

| Poisson Iterations | Residual (CUDA) | Residual (CPU) | Delta |
|---|---|---|---|
| 100 | $6.34 \times 10^{-3}$ | $6.32 \times 10^{-3}$ | $1.88 \times 10^{-5}$ (+0%) |
| 200 | $1.49 \times 10^{-3}$ | $1.48 \times 10^{-3}$ | $1.73 \times 10^{-5}$ (+1%) |
| 300 | $6.49 \times 10^{-4}$ | $6.37 \times 10^{-4}$ | $1.16 \times 10^{-5}$ (+2%) |
| 1000 | $5.54 \times 10^{-5}$ | $5.28 \times 10^{-5}$ | $2.54 \times 10^{-6}$ (+5%) |

Table 9.1.3: Residual values after 50 s of simulation on original simulation input

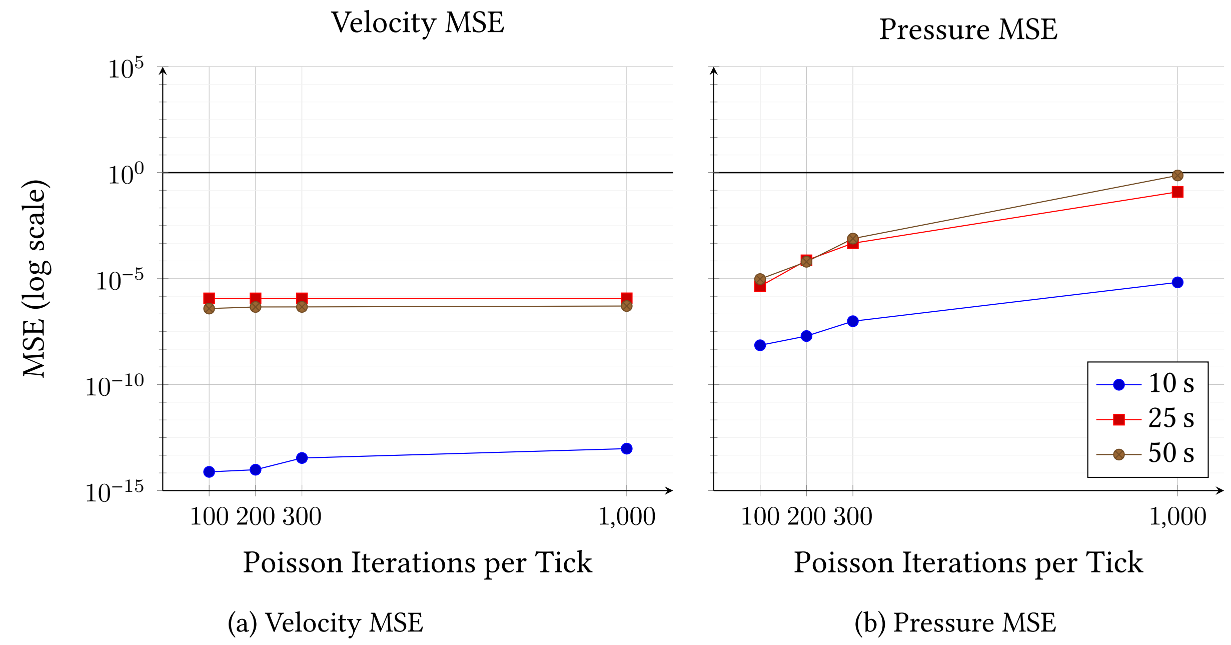


Figure 9.1.6: Initial MSE Results

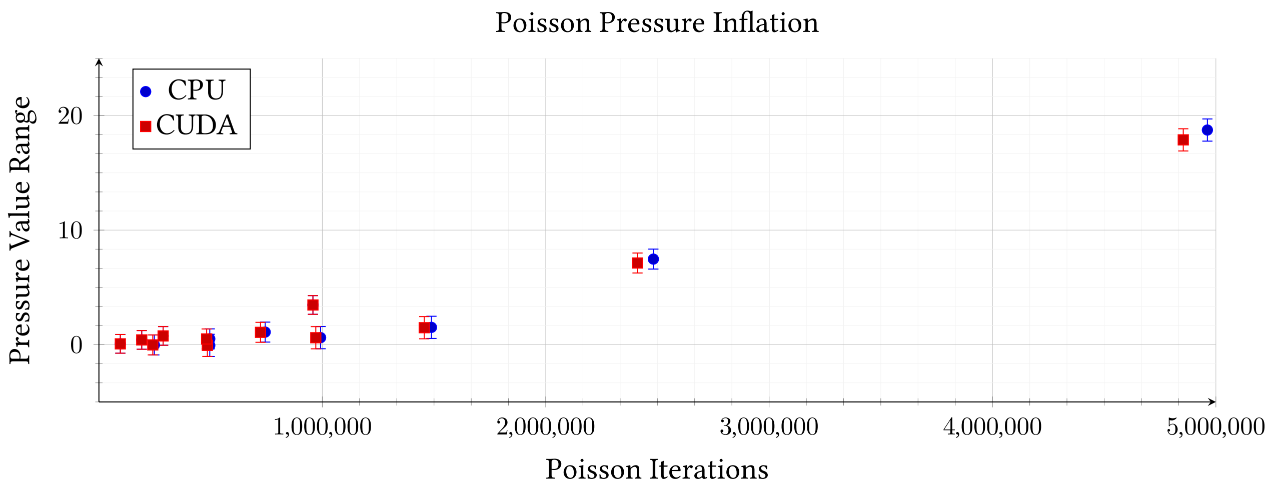


Figure 9.1.7: Inflation of pressure values with Poisson iterations

the solution - how close the values come to fulfilling the constraints of the equation. The original simulation calculated this residual value as part of the Poisson loop, but this was removed for optimization purposes. Table 9.1.3 presents the residual values as calculated after the simulations completed, showing that the difference between CPU and CUDA in terms of the solver accuracy is very small. CUDA is slightly less precise, but is at worst within just 5% of the CPU residual. This is likely forgivable for the sake of real-time visualization, but could still be improved in the future.

The large increase in Pressure MSE is explained by Fig. 9.1.7. The actual pressure values increase as more Poisson iterations are performed, for both CPU and CUDA. CUDA's values inflate slightly more slowly, giving each grid cell a high square error, resulting in a large MSE. When the MSE is calculated *after* subtracting the mean

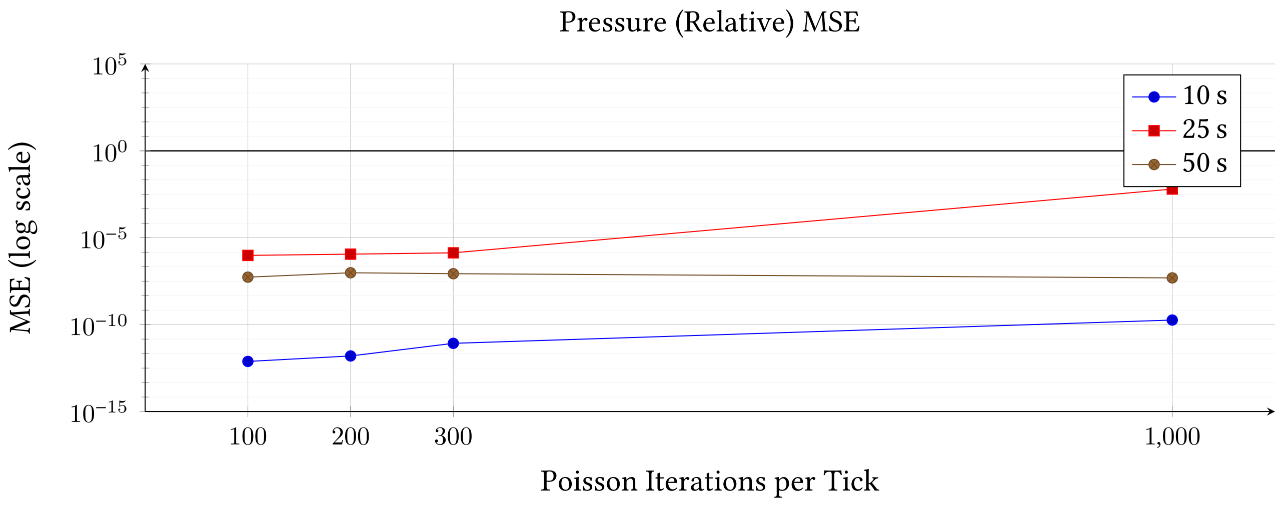


Figure 9.1.8: MSE for Pressure when adjusted to use relative values

from both datasets, i.e. comparing the relative values, the results are closer to the velocity MSE (Fig. 9.1.8). The pressure is only used internally as a relative value so in theory this inflation is fine, but in practice if it increases too much it could reach the limits of IEEE-754 floating point[38] and cause accuracy/precision loss in the solver. The original book mentioned that “nonphysical pressure values” may be the result of noncontinuous starting velocities[8], and mentioned a method for resolving this (see Section 2.2.2). This may prevent inflation, but it doesn’t explain the deviation between the CPU and CUDA.

The CUDA and CPU compilers are configured to handle floating-point numbers slightly differently. `nvcc` compiles with the `--fmad` option turned on by default[50] which “enables the contraction of floating-point multiplies and adds/subtracts into floating-point multiply-add operations”. This optimization performs a multiply and an addition at the same time, without a rounding step in between, resulting in a slightly more accurate result[51]. It is disabled for the CPU compiler, to keep the results from the other CPU backends consistent with the original simulation. While the change is subtle it’s present in all CUDA kernels, and when applied thousands of times may have a significant effect, leading to the deviation. In the future, this could be verified by compiling the CPU simulation with this option enabled.

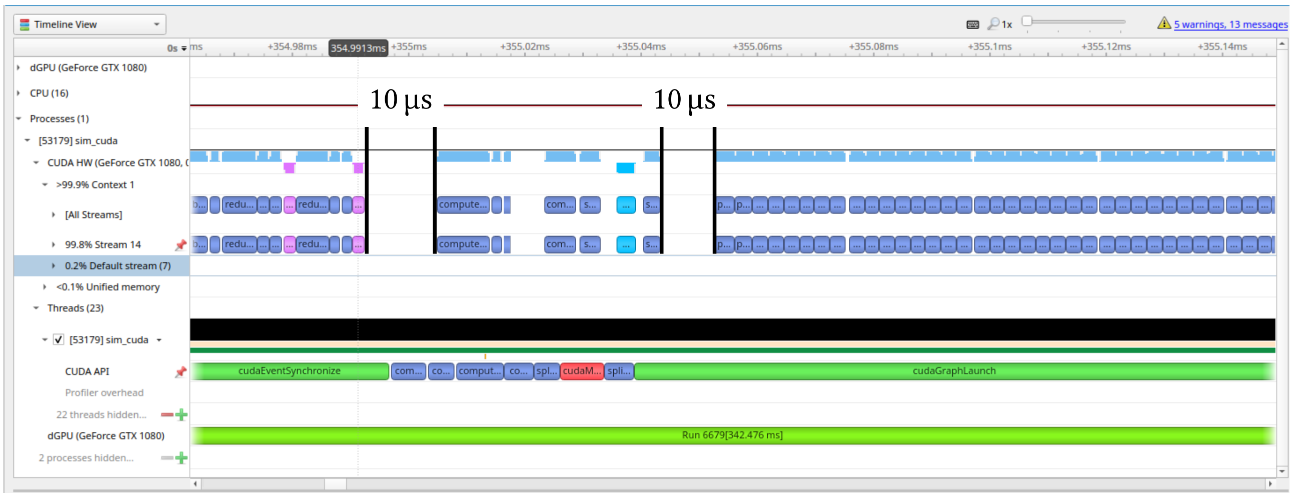


Figure 9.1.9: Simulation Profile, highlighting the GPU bubbles

### 9.1.3 GPU Utilization

During the Poisson iterations the GPU utilization is 100% thanks to the CUDA Graph optimization. At the tick boundaries, a bubble is unavoidable (see Fig. 9.1.9) where the CPU has to wait for the GPU reductions to finish before calculating the timestep for the next tick and invoking the next kernels. This bubble is approximately 10 µs long, combined with another 10 µs bubble between the Tentative Velocity stage finishing and the Poisson stage beginning. Each tick at 100 iterations for this simulation took 700 µs, so this gap accounts for 2.8% of the runtime. As the simulation gets larger, or the iteration count increases, this gap should remain constant and become even less significant. Computing the timestep entirely on the GPU, instead of sending data back and forth from the CPU, could allow the kernels to be enqueued earlier and avoid the GPU bubble.

### 9.1.4 Memory Leaks

The program is designed to allocate all memory up-front, instead of allocating during a simulation. This makes memory leaks unlikely, but not impossible. `valgrind` was used to test if the program leaked any memory using the CPU backend. Support for CUDA-based memory in `valgrind` seemed to be lacking, so it was run with suppressions enabled that hid CUDA-related false positives. It did not find any leaks in the project code, but did find a potential memory leak in the OpenMP implementation[2]. `cuda-memcheck` was then used to find any leaked CUDA memory, and found nothing.

[2]This may also be a false positive similar to those from CUDA.

| Feature | Worst-Case |
|---|---|
| Quantity-by-Scalar | Auto-range enabled |
| Quantity-by-Vector | Auto-range enabled, grid spacing set to display maximum number of vectors = 10,000 |
| Particle System | Particle simulation enabled, emitting maximum amount of particles per frame = 16, simulating and rendering maximum amount of particles = 100,000 |

Table 9.2.1: Testing Scenarios for Visualization Feature Speed.

| | Base Frame | with Sim | Scalar Quantity | Vector Field | Particles |
|---|---|---|---|---|---|
| Mean Time (ms) | 0.30 | 1.18 | 0.39 | 0.46 | 0.42 |
| △ from base (ms) | - | +0.88 | +0.09 | +0.16 | +0.12 |

Table 9.2.2: Visualization feature execution times

## 9.2 Visualization

This section investigates three properties of the real-time visualization: speed, GPU utilization, and memory leaks. While these properties do overlap with the simulation, this section focuses only on the properties of the visualization itself.

### 9.2.1 Speed

The speed of each feature was measured by enabling only that feature, moving to the worst-case for that feature such as enabling auto-range and maximizing on-screen instances (Table 9.2.1), and then reading off the time-to-render from the GUI. The rendered simulation state was the 660 x 120 original simulation input, which is rendered internally at 2x resolution in both directions i.e. 1320 x 240 then composited with the GUI onto a 1600 x 900 window.

The individual features are all faster than the simulation, even when combined, and are significantly shorter than the 16.6 ms required for a 60FPS visualization. Even at scale, the visualization features should not have a significant impact on the visualization speed.

### 9.2.2 GPU Utilization

Just like the Simulation, the GPU utilization is 100% where possible. In theory, the visualization work would hide the extra latency from waiting for the reduction to finish, but in practice the visualization work itself waits for around 70 µs after the semaphore is raised before starting (Fig. 9.2.1). The next CUDA tick, which has been

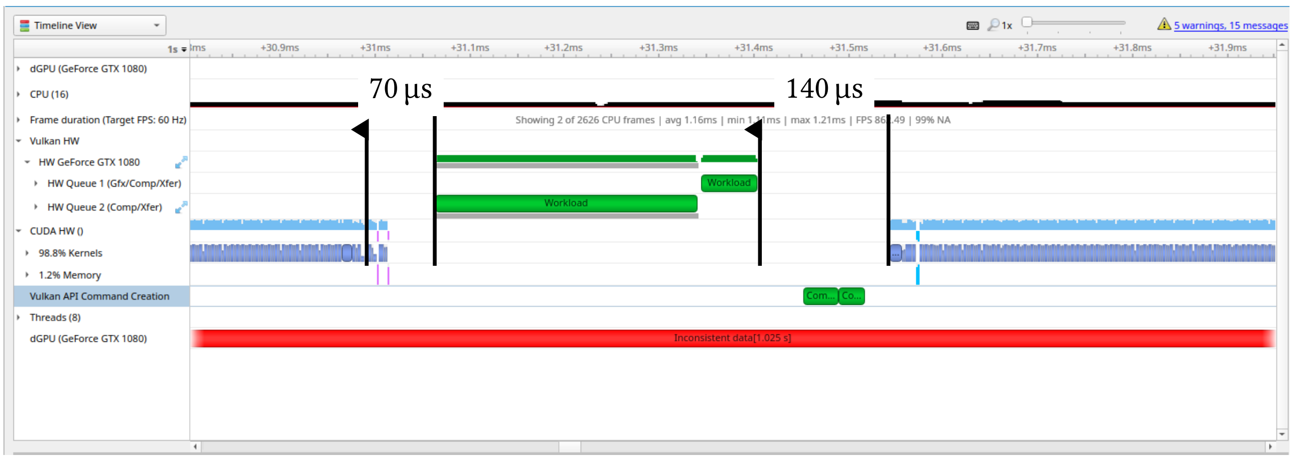


Figure 9.2.1: Visualization Profile, highlighting the GPU bubbles

enqueued well in advance by the CPU, then takes 140 µs to start after the previous visualization rendering has finished. In both cases there was no other GPU work executing for this program, nor any other element of this program that could have delayed it, thus it must be due to outside factors e.g. the OS compositing system using the GPU to render the desktop.

### 9.2.3 Memory Leaks

Testing the visualization for memory leaks proved more difficult than for the simulation. Running `valgrind` with suppressions enabled found no errors in this program, but found many potential errors (or more likely false positives) in the SDL2 windowing library and the underlying X windowing system. `cuda-memcheck` produced a host of "Invalid read" errors, which seemed to originate from CUDA/Vulkan shared memory. The documentation stated it could not handle DirectX interoperability[52], but said nothing about Vulkan. It suggested using the `compute-sanitizer` tool in other situations, which explicitly does not support Vulkan interoperability[53], and that tool produced exactly the same errors. Given that the errors were identical, it is very likely that both programs do not support Vulkan and should not be trusted.

As a last resort, the Memory Usage statistic from the Nvidia NSight Systems profiler was consulted. This showed that the CUDA memory usage stayed constant, but had no option to show memory allocated in Vulkan. Vulkan memory should all be explicitly allocated at the start with smart resource manager classes, so it is incredibly unlikely that any Vulkan memory is leaked, but currently there are no tools able to verify this.

# CHAPTER 10

# Evaluation

This chapter evaluates the project's final outcomes against the requirements laid out in Chapter 4, providing an objective measurement of success. This is then supplemented with a justification for the few missed requirements, all of which were non-essential. The project management is then appraised.

## 10.1 Requirements Evaluation

The Requirements in Chapter 4 evolved as the simulation and visualization areas were researched throughout the project. The program was designed with these requirements in mind, and by checking if the program meets these requirements the overall success of the project can be determined. Each requirement has been tested by one or more of the tests defined in Chapter 8. Table 10.1.1 and Table 10.1.2 show the tests for each requirement and the combined status of these tests. The full description of each requirement can be found in Chapter 4 and the descriptions of the tests can be found in Chapter 8.

All 24 of the **must**-have requirements have been met, as have 8 out of the 11 **should**-have requirements. This is an overwhelming success, showing the core functionality of the program is as expected. The 3 failed **should**-have requirements, shown in Table 10.1.3, are relatively minor. Furthermore, each omission can be justified.

| ID | Priority | Tests | Status |
|---|---|---|---|
| F1 | **must** | Implicit in program behaviour | ✓ |
| F2 | **must** | T9, T10, T13, T14 | ✓ |
| F3 | **must** | T4 | ✓ |
| F4 | **must** | T5 | ✓ |
| F4.1 | **must** | T5 | ✓ |
| F5 | **must** | T6 | ✓ |
| F5.1 | **must** | T17 | ✓ |
| F5.2 | **should** | T18 | ✗ |
| F5.3 | **should** | T19 | ✓ |
| F5.4 | **should** | T20 | ✓ |
| F5.5 | **should** | T21 | ✓ |
| F5.6 | **must** | T20, T21 | ✓ |
| F6 | **must** | T15, T16 | ✓ |
| F7 | **must** | T1, T2 | ✓ |
| F7.1 | **should** | T1, T2 | ✗ |
| F8 | **must** | T22 | ✓ |
| F8.1 | **must** | T22 | ✓ |
| F8.2 | **must** | T22 | ✓ |
| F8.3 | **must** | T22 | ✓ |
| F8.4 | **must** | T22 | ✓ |
| F9 | **should** | T23 | ✓ |
| F10 | **should** | T24 | ✓ |
| F11 | **should** | T19 | ✓ |

Table 10.1.1: Evaluation of Functional Requirements

| ID | Priority | Tests | Status |
|---|---|---|---|
| NF1 | **must** | T26 | ✓ |
| NF2 | **must** | T25, T27, T28, T29 | ✓ |
| NF3 | **must** | T31 | ✓ |
| NF4 | **should** | T30 | ✓ |
| NF5 | **must** | T32 | ✓ |
| NF6 | **should** | T33 | ✓ |
| NF7 | **should** | By Inspection | ✓ |
| NF8 | **should** | T22 | ✗ |
| NF9 | **must** | By Inspection | ✓ |
| NF10 | **should** | By Inspection | ✓ |
| NF11 | **should** | T34 | ✓ |

Table 10.1.2: Evaluation of Non-Functional Requirements

| ID | Priority | Tests | Status |
|---|---|---|---|
| F5.2 | **should** | T18 | ✗ |
| F7.1 | **should** | T1, T2 | ✗ |
| NF8 | **should** | T22 | ✗ |

Table 10.1.3: Failed Requirements

Satisfying Req. F5.2, which required saving the simulation state during a visualization, would have delayed the work on the visualization layers. As the visualization is a significant portion of the project, and this feature would not have had a relevant use-case during development, it was cut.

Req. F7.1 would require the `compare` tool to output a single SIMILAR/NOT SIMILAR value rather than the more detailed metrics it now uses. Using a single numeric value to convey this information would be near-impossible and result in a significant loss of nuance, even more so if only binary options are available, so would be pointless if shown alongside the more detailed metrics.

Req. NF8 was planned in the Progress Report, specifically citing BOID-like behaviour[39] as a potential means for reducing clumping. This would have greatly increased the complexity of the particle system. Particles would need to identify other nearby particles, which would likely require the positions to be sorted for efficient access, and while this has been implemented on the GPU before[40][85] it wasn't possible to implement it before the code freeze.

## 10.2 Project Management

Project Management has been incredibly successful, allowing an extremely complex combined simulation and visualization to be efficiently completed on schedule. The schedule itself allotted enough time for both research and implementation, and gave enough leeway that the manifested risks did not prevent success. Using third-party libraries for GUI management and command-line parsing allowed developer time to be spent on important problems, instead of hooking together customized implementations. The Code Freeze implemented in Week 22 ensured enough time was devoted to developing the presentation, which was crucial to success. As shown by the many successful requirements it did not make the program fall short in any way.

CHAPTER 11

# Conclusion

This project aimed to create a GPU fluid simulation and real-time in-situ visualization program, which required substantial research on fluid simulations, optimizing large parallel computation on GPUs, and various visualization methods. The implementation required knowledge of C++, Vulkan, and CUDA; an in-depth understanding of the complex underlying details of each, including their handling of memory; and an effective high-level design. On top of this, the project was well scheduled, allowing all core features to be implemented while allowing enough time for the associated reports and presentation to be developed. Rigorous testing was undertaken to ensure the program met the requirements, and the program passed with flying colours. The behaviour of the simulation at scale was measured, producing interesting findings and paving the way for future work in the area. Overall, the project has been a success.

## 11.1 Summary

The full in-situ visualization is a very large program, with over 8.5 thousand lines of custom code, which is impressive in its own right. It also brings novelty as an accurate simulation/visualization that runs in real-time, rather than rendering a visualization to disk for later viewing or rendering a static simulation state. It uses the high-performance Vulkan rendering API, which other toolkits have been slow to adopt[1]. Along with the up-and-coming Datoviz library[86], it is a step forwards to bring Vulkan-level performance to the wider visualization community.

Porting the simulation to CUDA is not a new work, but it was still a significant undertaking for the researcher and required extra thought to adapt to a tightly-coupled visualization. Gathering results at different scales demonstrated the upper limit of GPU throughput, and empirically showed the importance of cache-friendliness in GPU algorithms. It is certainly a good starting point for future work in this space.

## 11.2 Reflection

Completing the project successfully relied on using good development practices. During Term 2 extensive notes were taken while solving bugs and designing the rest of the program, ensuring all notable choices could be documented in this report and the presentation. Using Git branches to develop multiple features separately prevented confusion when working with the code, and maintaining a 'master' branch ensured that an up-to-date bug-free version of the program was always available. This project also tied in the researcher's prior knowledge from other areas, such as memory models, caches, and functional programming.

While the project as a whole was successful, some small elements could have been better executed. Third-party libraries were used in places, most notably for the GUI, but were not used in the low-level memory management or other Vulkan code. For Vulkan specifically, using the Vulkan Memory Allocator library[87] would have allowed for easier memory allocation. Other Vulkan wrappers and tools, such as the codebase developed by Sascha Willems for their Vulkan samples[88], may have made certain actions less cumbersome.

A common pattern encountered when implementing new code was to design for a larger system than necessary. For example, when implementing the worker thread, a generic worker thread setup was created in case another worker thread was needed later. Building a single worker thread would have been simpler and quicker. In general, a lack of initial investigation on the coding side led to slight overcomplication. However, this was very minor, as most of the code developed is still used in the program. Some elements, like the simulation runners, even benefited from being designed as generic code first!

[1]VTK has a Vulkan branch at `https://gitlab.kitware.com/ken-martin/vtk/-/tree/vulkan/Rendering/Vulkan`, which hasn't been updated since August 2020.

## 11.3 Future Work

On the simulation side, clear areas for improvement are the cache usage/performance at scale, and the pressure inflation problem. Both would require further investigation, but have some easy starting points. Other parallel GPU algorithms take advantage of shared memory and locality to improve performance, which the algorithm could be adapted to support. Investigating the non-physical pressure values fix from [8] (see Section 2.2.2) could be the key to removing pressure inflation. Re-implementing the Poisson residual check may reduce the number of required Poisson iterations per tick. Different Poisson solvers could also be added to the simulation, which may be more cache-friendly.

Visualization also has many potential improvements. A more advanced particle simulation as mentioned in Section 10.1 could be implemented, which would require more research into game industry particle simulations. For better accessibility, the colours initially used in the visualization could also be adapted to be more colourblind friendly.

Truly parallel simulation/visualization has not been achieved, mostly due to the limitations of the researcher's hardware (Section 5.3.2). As established in Section 9.2.1 the visualization is already very fast, but for larger & more complicated visualizations this may become a significant portion of runtime. Using multiple GPUs, perhaps on separate systems, to implement a loosely-coupled version of this visualization could allow for a truly parallel visualization and investigation of the benefits vs. tight coupling.

All in all, this project is in a very open research area and has great potential to expand. As massively parallel systems become more powerful and accessible, running programs like this on larger datasets will become more and more feasible, especially in industrial applications. It will be very interesting to see what comes next.

# CHAPTER 12

# Bibliography

[1] Antony Jameson, Luigi Martinelli and J. Vassberg. 'Using Computational Fluid Dynamics For Aerodynamics- A Critical Assessment'. In: *Proceedings of the 23RD International Congress of Aeronautical Sciences*. September 2002.

[2] Andrew L. Sullivan. 'Wildland surface fire spread modelling, 1990 - 2007. 1: Physical and quasi-physical models'. In: *International Journal of Wildland Fire* 18.4 (2009), p. 349. ISSN: 1049-8001. DOI: 10.1071/wf06143.

[3] 'Fluid Dynamics on the Big Screen'. In: *ANSYS Advantage* II.2 (2008), pp. 52–53. URL: https://www.ansys.com/-/media/ansys/corporate/resourcelibrary/article/aa-v2-i2-fluid-dynamics-on-big-screen.pdf.

[4] James Kress. 'In Situ Visualization Techniques for High Performance Computing'. 2017. URL: https://www.cs.uoregon.edu/Reports/AREA-201703-Kress.pdf (visited on 06/03/2021).

[5] Medvecký-Heretik Jakub. 'Real-time Water Simulation in Game Environment'. PhD thesis. Masaryk University, Faculty of Informatics, 2018.

[6] Jos Stam. 'Stable Fluids'. In: *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '99. USA: ACM Press/Addison-Wesley Publishing Co., 1999, pp. 121–128. ISBN: 0201485605. DOI: 10.1145/311535.311548.

[7] Peter Sikachev. 'Real-Time Fluid Simulation in Shadow of the Tomb Raider'. 2018.

[8] Michael Griebel, Thomas Dornseifer and Tilman Neunhoeffer. *Numerical simulation in fluid dynamics: a practical introduction*. SIAM, 1998. DOI: 10.1137/1.9780898719703.fm.

[9] R.B. Bird, W.E. Stewart and E.N. Lightfoot. *Transport Phenomena*. Transport Phenomena v. 1. Wiley, 2006. ISBN: 9780470115398.

[10] G. Falkovich. *Fluid Mechanics*. Cambridge University Press, 2018. ISBN: 9781107129566.

[11] Francis H. Harlow and J. Eddie Welch. 'Numerical Calculation of Time-Dependent Viscous Incompressible Flow of Fluid with Free Surface'. In: *Physics of Fluids* 8.12 (1965), p. 2182. ISSN: 00319171. DOI: 10.1063/1.1761178.

[12] M Perić, R Kessler and G Scheuerer. 'Comparison of Finite-Volume Numerical Methods with Staggered and Colocated Grids'. In: *Comput. Fluids* 16.4 (September 1988), pp. 389–403. ISSN: 0045-7930. DOI: 10.1016/0045-7930(88)90024-2.

[13] B. D. Nichols and C.W. Hirt. 'Methods for Calculating Multi-Dimensional, Transient, Free Surface Flows Past Bodies'. In: *First International Conference on Numerical Ship Hydrodynamics* (20th–22nd October 1975). Ed. by Joanna W. Schot and Nils Salvesen. David W. Taylor Naval Ship Research and Development Center, 1975, pp. 253–278.

[14] Murilo F. Tome and Sean McKee. 'GENSMAC: A Computational Marker and Cell Method for Free Surface Flows in General Domains'. In: *Journal of Computational Physics* 110.1 (1994), pp. 171–186. ISSN: 0021-9991. DOI: 10.1006/jcph.1994.1013.

[15] L. Adams and J. Ortega. 'A multi-color SOR method for parallel computation'. In: *11th International Conference on Parallel Processing - ICPP*. 1982, pp. 53–56.

[16] David M. Young. *Iterative Solution of Large Linear Systems*. Academic Press, 1971. ISBN: 9780127730509.

[17] 'L2 norm'. In: *Encyclopedia of Biometrics*. Ed. by Stan Z. Li and Anil Jain. Boston, MA: Springer US, 2009, pp. 883–883. ISBN: 978-0-387-73003-5. DOI: 10.1007/978-0-387-73003-5_1070.

[18] Masayuki Kuba, Constantine D. Polychronopoulos and Kyle Gallivan. 'The Synergetic Effect of Compiler, Architecture, and Manual Optimizations on the Performance of CFD on Multiprocessors'. In: *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*. Supercomputing '95. San Diego, California, USA: Association for Computing Machinery, 1995, 72–es. ISBN: 0897918169. DOI: 10.1145/224170.224426.

[19] Zhe Fan et al. 'GPU Cluster for High Performance Computing'. In: *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*. SC '04. USA: IEEE Computer Society, 2004, p. 47. ISBN: 0769521533. DOI: 10.1109/SC.2004.26.

[20] Tianyun Ni. 'Direct Compute - Bring GPU Compute to the Mainstream'. 2009.

[21] Kyle E Niemeyer and Chih-Jen Sung. 'Recent Progress and Challenges in Exploiting Graphics Processors in Computational Fluid Dynamics'. In: *J. Supercomput.* 67.2 (February 2014), pp. 528–564. ISSN: 0920-8542. DOI: 10.1007/s11227-013-1015-7.

[22] Jean-Michel Muller et al. 'The Fused Multiply-Add Instruction'. In: *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010, pp. 151–179. DOI: 10.1007/978-0-8176-4705-6_5.

[23] Rokiatou Diarra. 'Towards Automatic Restrictification of CUDA Kernel Arguments'. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE 2018. New York, NY, USA: Association for Computing Machinery, 2018, pp. 928–931. ISBN: 9781450359375. DOI: 10.1145/3238147.3241533.

[24] Mark Harris. *Optimizing Parallel Reduction in CUDA*. Tech. rep. URL: https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf.

[25] O Kreylos et al. 'Interactive Visualization and Steering of CFD Simulations'. In: *Proceedings of the Symposium on Data Visualisation 2002*. VISSYM '02. Goslar, DEU: Eurographics Association, 2002, pp. 25–34. ISBN: 158113536X. DOI: 10.5555/509740.509745.

[26] William Schroeder and Bill Lorenson. *Visualization Toolkit: An Object-Oriented Approach to 3-D Graphics*. 1st. USA: Prentice Hall PTR, 1996. ISBN: 0131998374.

[27] William E. Lorensen and Harvey E. Cline. 'Marching Cubes: A High Resolution 3D Surface Construction Algorithm'. In: *SIGGRAPH Comput. Graph.* 21.4 (August 1987), pp. 163–169. ISSN: 0097-8930. DOI: 10.1145/37402.37422.

[28] M. Schulz et al. 'Interactive visualization of fluid dynamics simulations in locally refined cartesian grids'. In: *Proceedings Visualization '99 (Cat. No.99CB37067)*. 1999, pp. 413–553. DOI: 10.1109/VISUAL.1999.809918.

[29] Shyh-Kuang Ueng, C. Sikorski and Kwan-Liu Ma. 'Efficient streamline, streamribbon, and streamtube constructions on unstructured grids'. In: *IEEE Transactions on Visualization and Computer Graphics* 2.2 (1996), pp. 100–110. DOI: 10.1109/2945.506222.

[30] Lei Chen et al. 'A new seismic data visualization method'. In: *2016 22nd International Conference on Automation and Computing (ICAC)*. 2016, pp. 467–472. DOI: 10.1109/IConAC.2016.7604964.

[31] K. Gaither. 'Visualization's role in analyzing computational fluid dynamics data'. In: *IEEE Computer Graphics and Applications* 24.3 (2004), pp. 13–15. DOI: 10.1109/MCG.2004.1297005.

[32] D.A. Lane. 'Visualization of time-dependent flow fields'. In: *Proceedings Visualization '93*. 1993, pp. 32–38. DOI: 10.1109/VISUAL.1993.398848.

[33] *NVIDIA AMPERE GA102 GPU ARCHITECTURE*. Tech. rep. 2020. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.pdf (visited on 22/02/2021).

[34] Louis Dionne. 'Runtime Polymorphism: Back to the Basics'. CppCon 2017. November 2017. URL: https://www.youtube.com/watch?v=gVGtNFg4ay0.

[35] M. Lipovača. *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press Series. No Starch Press, 2011. ISBN: 9781593272838.

[36] The Khronos Group Inc. *Vulkan 1.1 Reference Guide*. Tech. rep. URL: https://www.khronos.org/vulkan.

[37] *NVIDIA GeForce GTX 1080*. Tech. rep. 2016. URL: https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf (visited on 23/04/2021).

[38] 'IEEE Standard for Floating-Point Arithmetic'. In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), pp. 1–84. DOI: 10.1109/IEEESTD.2019.8766229.

[39] Craig W. Reynolds. 'Flocks, herds, and schools: A distributed behavioral model'. In: *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1987*. New York, New York, USA: Association for Computing Machinery, Inc, August 1987, pp. 25–34. ISBN: 0897912276. DOI: 10.1145/37401.37406.

[40] Sebastian Lindqvist. 'Performance Evaluation of Boids on the GPU and CPU'. PhD thesis. Blekinge Institute of Technology, Faculty of Computing, 2018. URL: https://urn.kb.se/resolve?urn=urn:nbn:se:bth-15970.

[41] NVIDIA. *NVIDIA CUDA Programming Guide*. 2007.

[42] Mark Harris. *How to Access Global Memory Efficiently in CUDA C/C++ Kernels | NVIDIA Developer Blog*. NVIDIA. January 2013. URL: https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/.

[43] NVIDIA. *Global Memory - CUDA C++ Programming Guide*. Version v11.3.0. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#global-memory-3-0.

[44] *PTX ISA :: CUDA Toolkit Documentation*. Version v11.3.0. NVIDIA. 2021. URL: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[45] *SIMD Intrinsics | CUDA Math API :: CUDA Toolkit Documentation*. Version v11.3.0. NVIDIA. 2021. URL: https://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH__INTRINSIC__SIMD.html.

[46] Alan Gray. *Getting Started with CUDA Graphs | NVIDIA Developer Blog*. NVIDIA. September 2019. URL: https://developer.nvidia.com/blog/cuda-graphs/.

[47] Mark Harris. *Unified Memory for CUDA Beginners | NVIDIA Developer Blog*. NVIDIA. June 2017. URL: https://developer.nvidia.com/blog/unified-memory-cuda-beginners/.

[48] NVIDIA. *CUDA Zone | NVIDIA Developer*. 2020. URL: https://developer.nvidia.com/cuda-zone.

[49] *CUDA Occupancy Calculator*. NVIDIA. URL: https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html (visited on 01/05/2021).

[50] *fmad | NVCC :: CUDA Toolkit Documentation*. Version v11.3.0. NVIDIA. 2021. URL: https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#options-for-steering-gpu-code-generation-fmad.

[51] *Floating Point and IEEE 754 :: CUDA Toolkit Documentation*. Version v11.3.0. NVIDIA. 2021. URL: https://docs.nvidia.com/cuda/floating-point/index.html#fused-multiply-add-fma.

[52] *Known Issues | CUDA-MEMCHECK :: CUDA Toolkit Documentation*. Version v11.3.0. NVIDIA. 2021. URL: https://docs.nvidia.com/cuda/cuda-memcheck/index.html#known-issues.

[53] *Compute Sanitizer - Release Notes*. Version v2021.1.0. NVIDIA. March 2021. URL: https://docs.nvidia.com/cuda/sanitizer-docs/pdf/ReleaseNotes.pdf.

[54] The Khronos Group Inc. *The Khronos Group Releases OpenCL 1.0 Specification*. 2008. URL: https://www.khronos.org/news/press/the_khronos_group_releases_opencl_1.0_specification.

[55] The Khronos Group Inc et al. *ARB_compute_shader*. The Khronos Group Inc. URL: https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_compute_shader.txt.

[56] *Vulkan® 1.1.176 - A Specification*. The Khronos Group. 2021. URL: https://www.khronos.org/registry/vulkan/specs/1.1/html/vkspec.html.

[57] The Khronos Group. *OpenCL Overview - The Khronos Group Inc*. URL: https://www.khronos.org/opencl/.

[58] *Computing and the Manhattan Project*. Atomic Heritage Foundation. 2014. URL: https://www.atomicheritage.org/history/computing-and-manhattan-project.

[59] Adam Chester and Graham Martin. *CS257 Advanced Computer Architecture Coursework*. 2020.

[60] S. Stark. *CS257 Report - Reducing the Execution Time of a Fluid Simulation Program*. 2020.

[61] OpenMP. *Home - OpenMP*. URL: https://www.openmp.org/.

[62] *Introduction to Intel® Advanced Vector Extensions.* Intel Corporation. 2011. URL: https://software.intel.com/content/www/us/en/develop/articles/introduction-to-intel-advanced-vector-extensions.html.

[63] *Autodesk Acquires ALGOR.* MCADCafe, December 2008. URL: https://www.mcadcafe.com/nbc/articles/view_article.php?section=Magazine&articleid=637593.

[64] *Definition of Streamlines.* NASA. 2015. URL: https://www.grc.nasa.gov/WWW/k-12/airplane/stream.html.

[65] Autodesk. *Autodesk CFD 2019.* 2019. URL: https://help.autodesk.com/view/SCDSE/2019/ENU/.

[66] Autodesk. *Exercise 7 | CFD.* 2019. URL: https://knowledge.autodesk.com/support/cfd/learn-explore/caas/CloudHelp/cloudhelp/2014/ENU/SimCFD/files/GUID-ADAF2C66-9992-43FA-B5FB-5CB13F967DAD-htm.html.

[67] *10 Day Trend – Big changes ahead 03/03/21.* Met Office - Weather. March 2021. URL: https://www.youtube.com/watch?v=y_1--MkiNjQ.

[68] János Turánszki. *GPU-based particle simulation.* wickedengine.net. November 2017. URL: https://wickedengine.net/2017/11/07/gpu-based-particle-simulation/.

[69] *CODE OF CONDUCT FOR BCS MEMBERS.* British Computing Society. 2015. URL: https://www.bcs.org/upload/pdf/conduct.pdf.

[70] *Git.* 2020. URL: https://git-scm.com/.

[71] *The Open Group Base Specifications.* 7. IEEE and The Open Group. 2018. URL: https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap12.html.

[72] Google LLC. *shaderc.* URL: https://github.com/google/shaderc/tree/main/glslc.

[73] *Simple DirectMedia Layer - Homepage.* URL: https://www.libsdl.org/.

[74] Omar Cornut. *Dear ImGui.* URL: https://github.com/ocornut/imgui.

[75] attractivechaos. *A survey of argument parsing libraries in C/C++.* August 2018. URL: https://attractivechaos.wordpress.com/2018/08/31/a-survey-of-argument-parsing-libraries-in-c-c/.

[76] Free Software Foundation. *getopt(3): Parse options - Linux man page.* URL: https://linux.die.net/man/3/getopt.

[77] GNU Project. *Argp (The GNU C Library).* URL: https://www.gnu.org/software/libc/manual/html_node/Argp.html.

[78] Tom Vajzovic. *Gopt - Free command line option and argument parsing C library*. URL: https://www.purposeful.co.uk/software/gopt/.

[79] jarro2783. *cxxopts: Lightweight C++ command line option parser*. URL: https://github.com/jarro2783/cxxopts.

[80] CLIUtils. *CLI11*. URL: https://github.com/CLIUtils/CLI11.

[81] *Move constructors*. cppreference.com. URL: https://en.cppreference.com/w/cpp/language/move_constructor.

[82] James Archbold. *CS261 Software Engineering*. 2020.

[83] *Levels of Testing*. ReQTest. URL: https://reqtest.com/testing-blog/different-levels-of-testing/ (visited on 01/05/2021).

[84] Tomasz Gebarowski. *Glib, GObject and memory leaks*. August 2008. URL: https://tgebarowski.github.io/2008/08/21/glib-gobject-and-memory-leaks/.

[85] Raphael Monnerat. *Unity-GPU-Boids*. URL: https://github.com/Shinao/Unity-GPU-Boids (visited on 01/05/2021).

[86] Cyrille Rossant. *Datoviz: GPU interactive scientific data visualization with Vulkan*. URL: https://github.com/datoviz/datoviz (visited on 01/05/2021).

[87] *Vulkan Memory Allocator*. GPUOpen. URL: https://gpuopen.com/vulkan-memory-allocator/ (visited on 01/05/2021).

[88] Sascha Willems. *Vulkan C++ examples and demos*. URL: https://github.com/SaschaWillems/Vulkan (visited on 01/05/2021).

# APPENDIX A

# Smart Resource Classes

This appendix lists the Smart Resource Classes included in the codebase, showing a brief overview of the kinds of resources used by the program. Each of these classes primarily manages the lifetime of one or more resources, which could be some form of memory or Vulkan/CUDA objects.

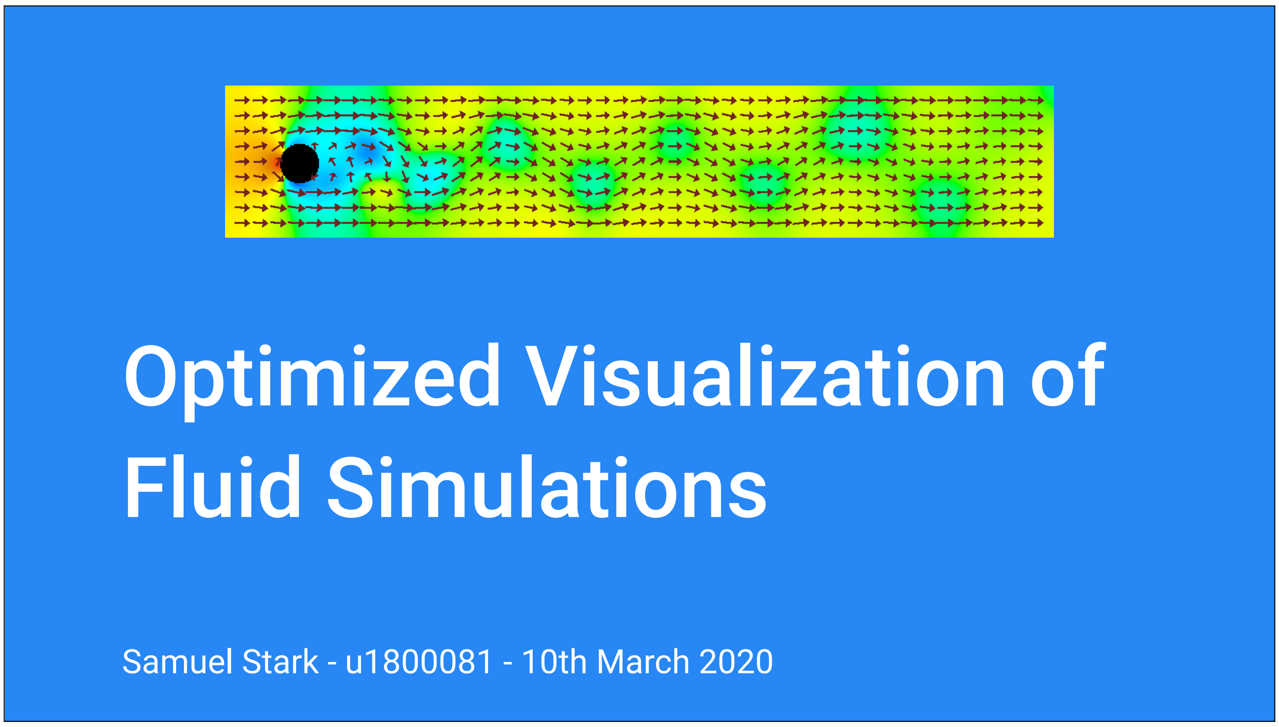
- `Sim2DArray`
- `SimRedBlackArray`
- `VulkanCudaBufferMemory`
- `FrameAllocator`
- `FrameSetAllocator`
- `VulkanSimAppData`
- `VulkanSimAppPipelineSet`
- `VulkanBackedBuffer`
- `VulkanBackedFramebuffer`
- `VulkanBackedGPUBuffer_WithStaging`
- `VulkanBackedGPUImage`
- `VulkanDescriptorSetLayout`

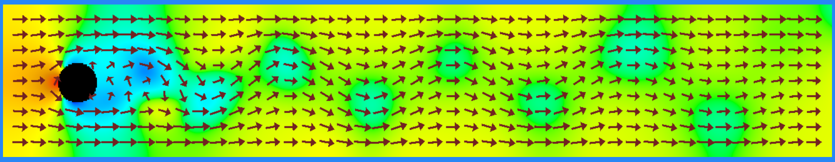
- `VulkanDeviceMemory`
- `VulkanFence`
- `VulkanFramebuffer`
- `VulkanImageSampler`
- `VulkanPipeline`
- `VulkanPipelineSpecMap`
- `VulkanRenderPass`
- `VulkanSemaphore`
- `VulkanShader`
- `VulkanSwapchain`
- `CudaGraphCapture`
- `CudaVulkanSemaphore`

# APPENDIX B

# Previous Project Reports

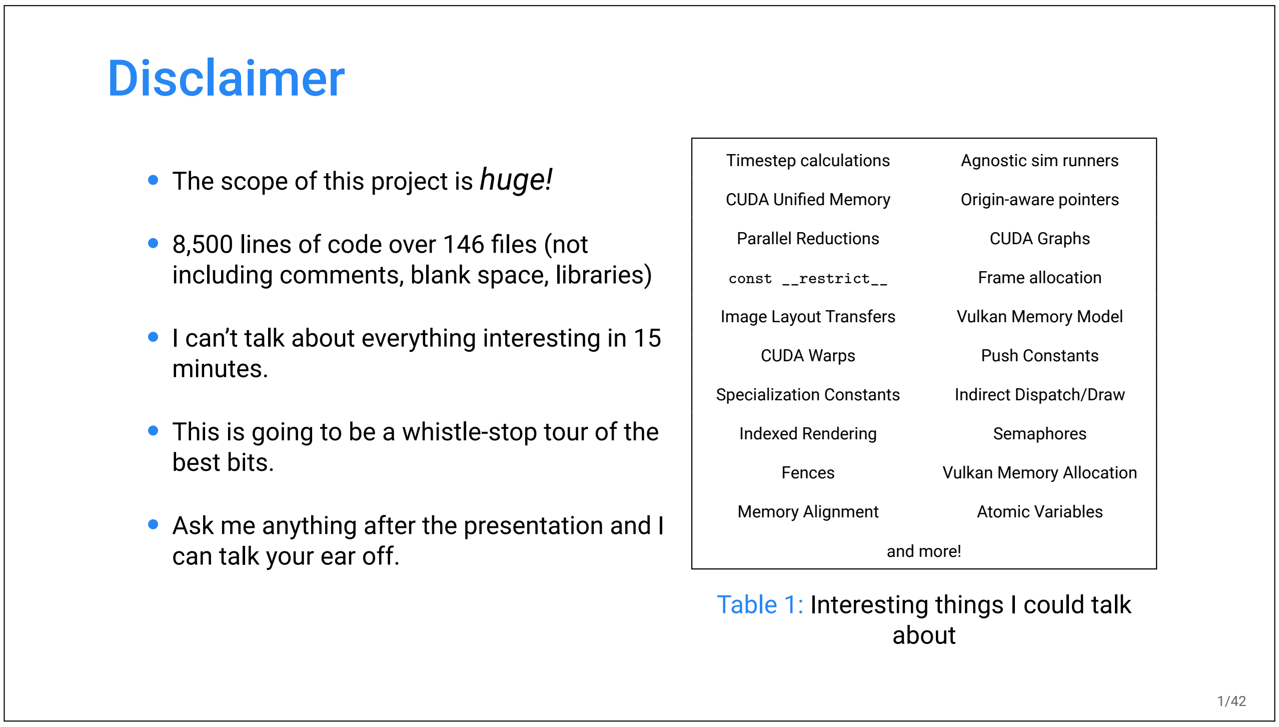
## B.1 Presentation





# Optimized Visualization of Fluid Simulations

Samuel Stark - u1800081 - 10th March 2020



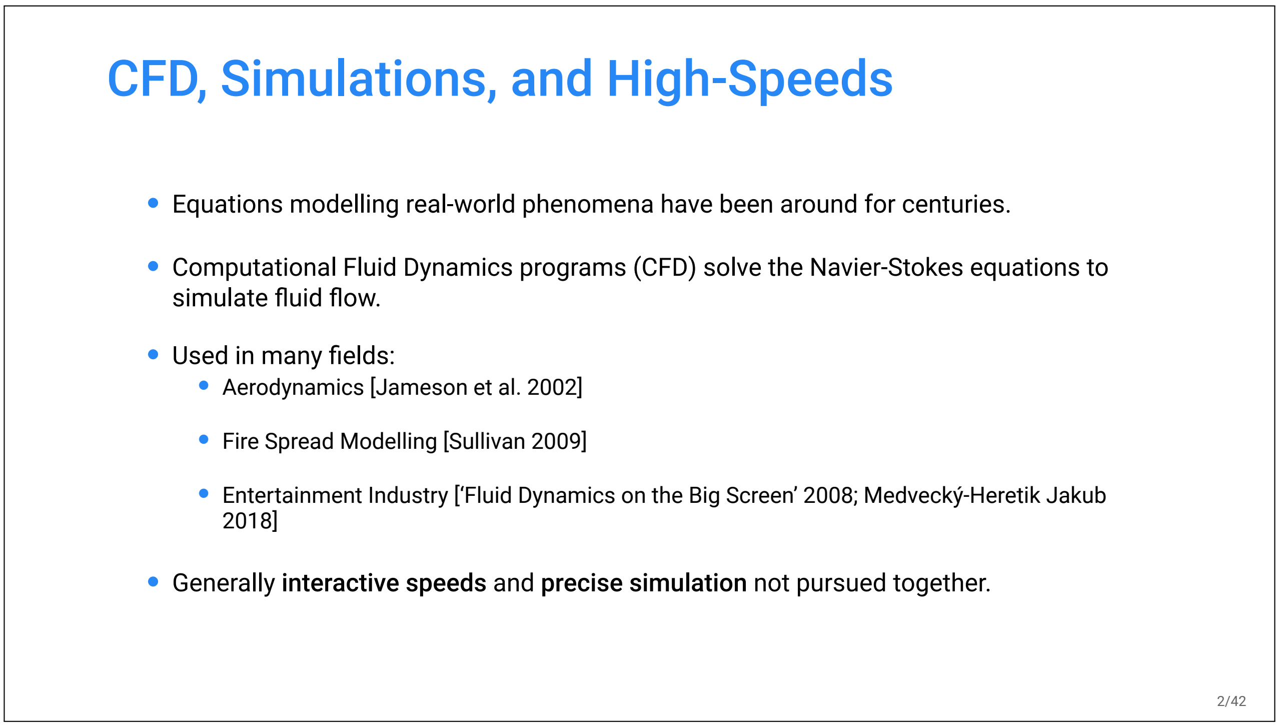
### Disclaimer

- The scope of this project is *huge!*
- 8,500 lines of code over 146 files (not including comments, blank space, libraries)
- I can't talk about everything interesting in 15 minutes.
- This is going to be a whistle-stop tour of the best bits.
- Ask me anything after the presentation and I can talk your ear off.

| | |
|---|---|
| Timestep calculations | Agnostic sim runners |
| CUDA Unified Memory | Origin-aware pointers |
| Parallel Reductions | CUDA Graphs |
| `const __restrict__` | Frame allocation |
| Image Layout Transfers | Vulkan Memory Model |
| CUDA Warps | Push Constants |
| Specialization Constants | Indirect Dispatch/Draw |
| Indexed Rendering | Semaphores |
| Fences | Vulkan Memory Allocation |
| Memory Alignment | Atomic Variables |
| and more! | |

Table 1: Interesting things I could talk about

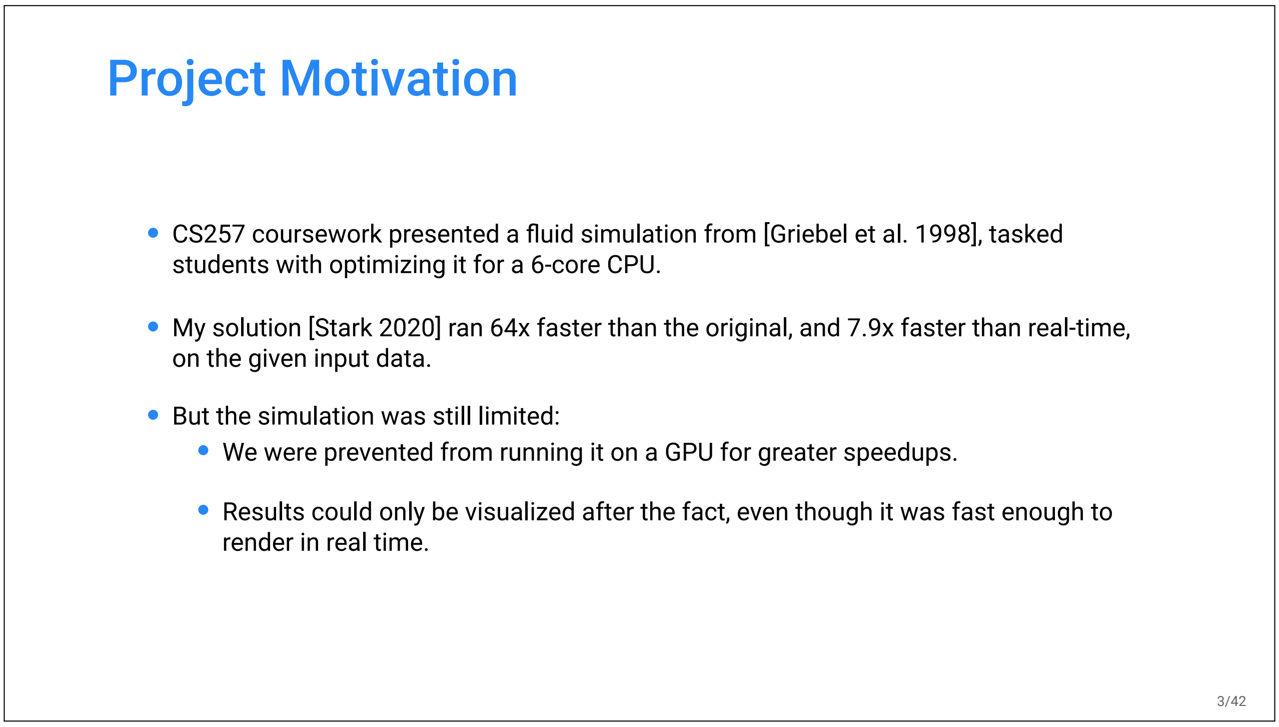
1/42



## CFD, Simulations, and High-Speeds

- Equations modelling real-world phenomena have been around for centuries.
- Computational Fluid Dynamics programs (CFD) solve the Navier-Stokes equations to simulate fluid flow.
- Used in many fields:
    - Aerodynamics [Jameson et al. 2002]
    - Fire Spread Modelling [Sullivan 2009]
    - Entertainment Industry ['Fluid Dynamics on the Big Screen' 2008; Medvecký-Heretik Jakub 2018]
- Generally **interactive speeds** and **precise simulation** not pursued together.

2/42



## Project Motivation

- CS257 coursework presented a fluid simulation from [Griebel et al. 1998], tasked students with optimizing it for a 6-core CPU.
- My solution [Stark 2020] ran 64x faster than the original, and 7.9x faster than real-time, on the given input data.
- But the simulation was still limited:
    - We were prevented from running it on a GPU for greater speedups.
    - Results could only be visualized after the fact, even though it was fast enough to render in real time.

3/42

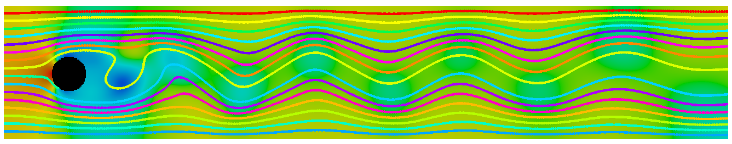
## Project Goals/Achievements

**Port the simulation to the GPU.**

**Exploit the speedup to improve accuracy and increase sim resolution.**

**Intuitively visualize the simulation in real time.**[1]

All goals were achieved!

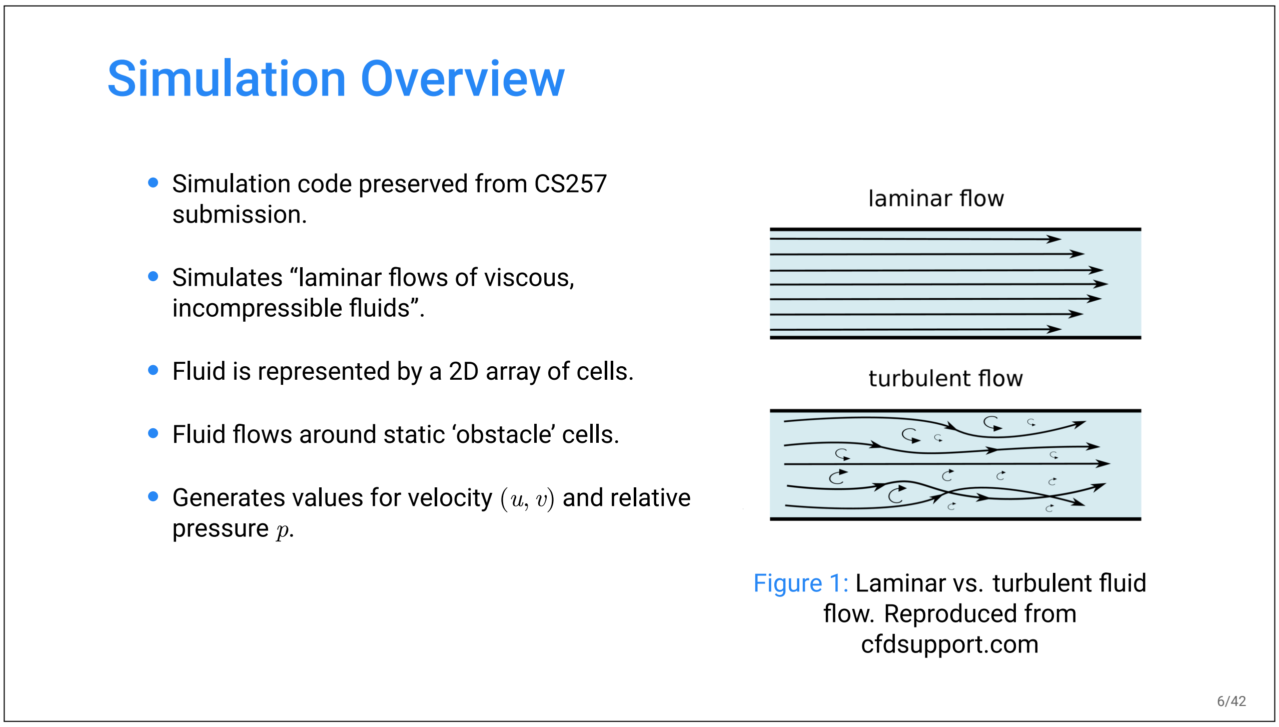


[1] Use games industry techniques for efficient rendering.

4/42

## Table of Contents

5/42



## Simulation Overview

- Simulation code preserved from CS257 submission.
- Simulates “laminar flows of viscous, incompressible fluids”.
- Fluid is represented by a 2D array of cells.
- Fluid flows around static ‘obstacle’ cells.
- Generates values for velocity $(u, v)$ and relative pressure $p$.

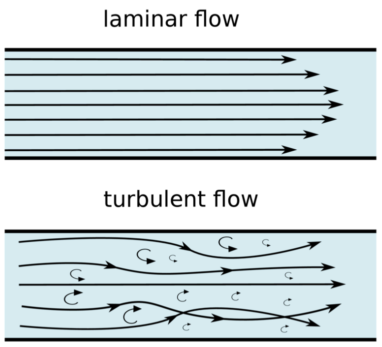
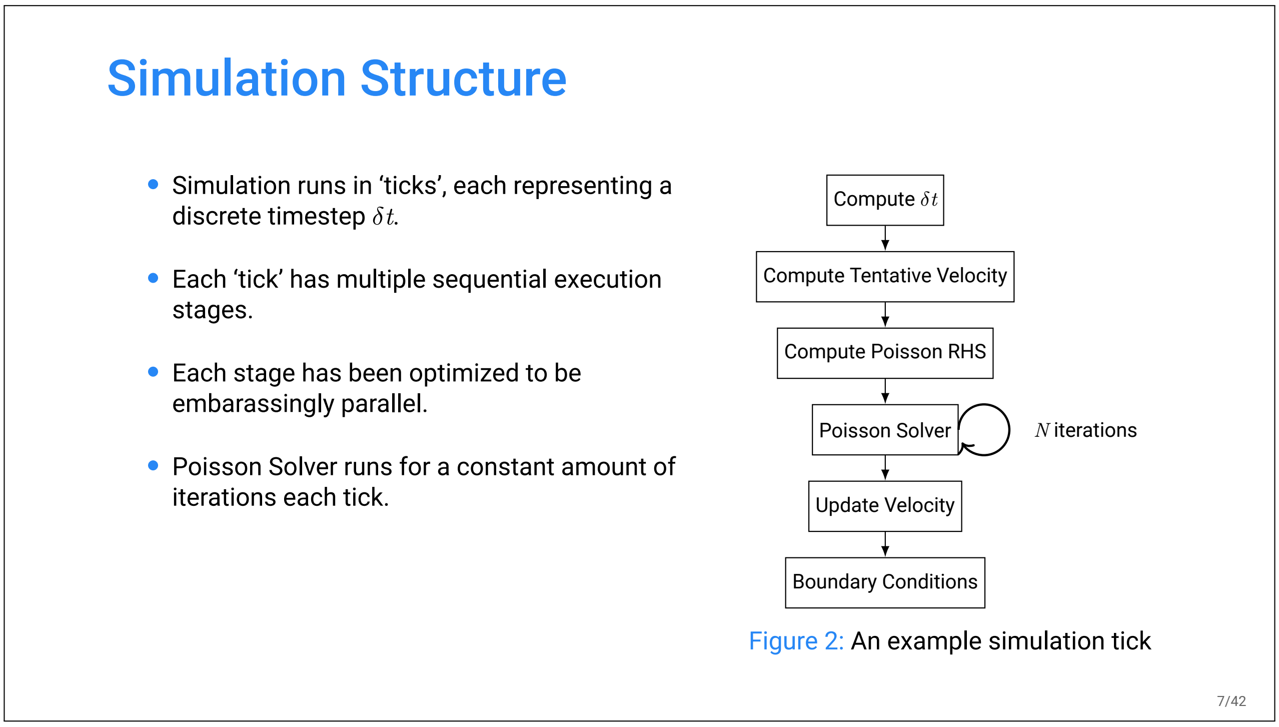


Figure 1: Laminar vs. turbulent fluid flow. Reproduced from cfdsupport.com

6/42



## Simulation Structure

- Simulation runs in ‘ticks’, each representing a discrete timestep $\delta t$.
- Each ‘tick’ has multiple sequential execution stages.
- Each stage has been optimized to be embarassingly parallel.
- Poisson Solver runs for a constant amount of iterations each tick.

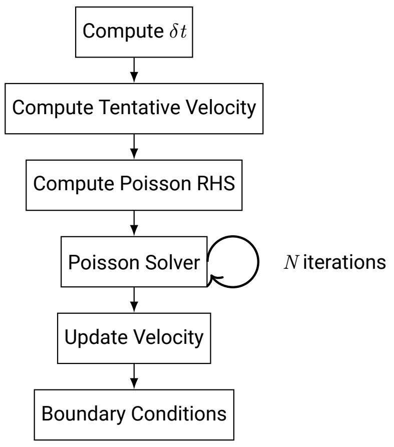
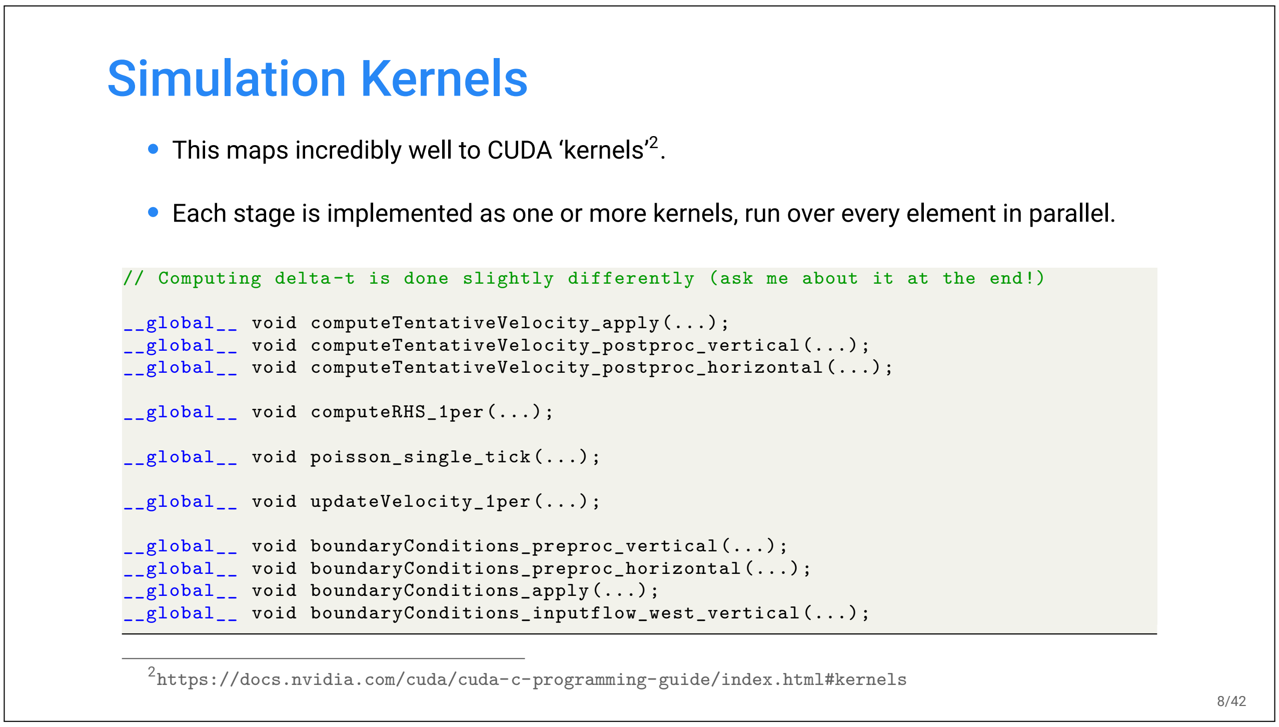


Figure 2: An example simulation tick

7/42



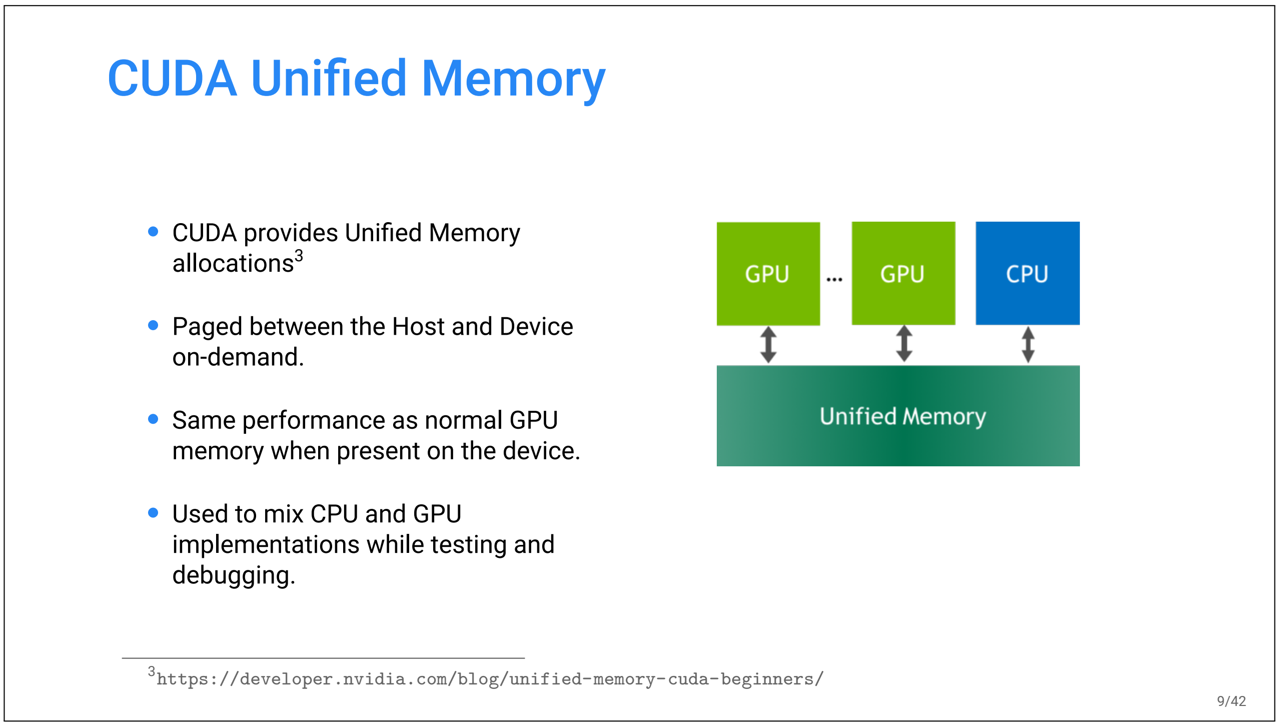
## Simulation Kernels

- This maps incredibly well to CUDA 'kernels'[2].
- Each stage is implemented as one or more kernels, run over every element in parallel.

```
// Computing delta-t is done slightly differently (ask me about it at the end!)

__global__ void computeTentativeVelocity_apply(...);
__global__ void computeTentativeVelocity_postproc_vertical(...);
__global__ void computeTentativeVelocity_postproc_horizontal(...);

__global__ void computeRHS_1per(...);

__global__ void poisson_single_tick(...);

__global__ void updateVelocity_1per(...);

__global__ void boundaryConditions_preproc_vertical(...);
__global__ void boundaryConditions_preproc_horizontal(...);
__global__ void boundaryConditions_apply(...);
__global__ void boundaryConditions_inputflow_west_vertical(...);
```

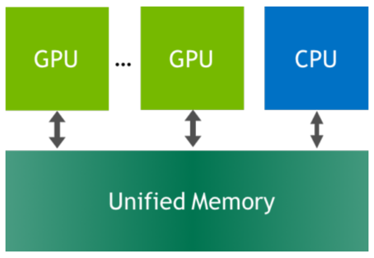
[2] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#kernels

8/42



## CUDA Unified Memory

- CUDA provides Unified Memory allocations[3]
- Paged between the Host and Device on-demand.
- Same performance as normal GPU memory when present on the device.
- Used to mix CPU and GPU implementations while testing and debugging.



[3] https://developer.nvidia.com/blog/unified-memory-cuda-beginners/

9/42

## `const __restrict__` pointers

- CUDA exposes a fast “read-only data cache”[4].
- To ensure the compiler knows memory is read only, use the `const` and `__restrict__` qualifiers on all pointers.
- Shown to speed up execution times in [Diarra 2018].

```
template<typename T>
using in_matrix =
    const T* const __restrict__;

template<typename T>
using out_matrix =
    T* const __restrict__;
```

Figure 3: Helper templates used in kernel definitions

Ask me about `const __restrict__` pointers at the end!

[4] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#global-memory-3-0

10/42

## Parallel Reductions

- Computing $\delta t$ requires the maximum values of $u$, $v$.
- We can do this in parallel on the GPU!
- Find the values on the GPU, then copy them to the CPU to calculate $\delta t$.
- Implementation taken from [Harris n.d.].

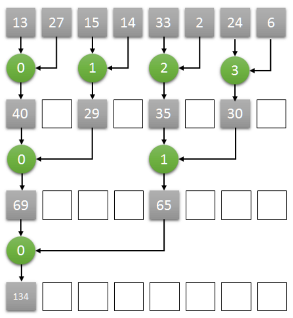
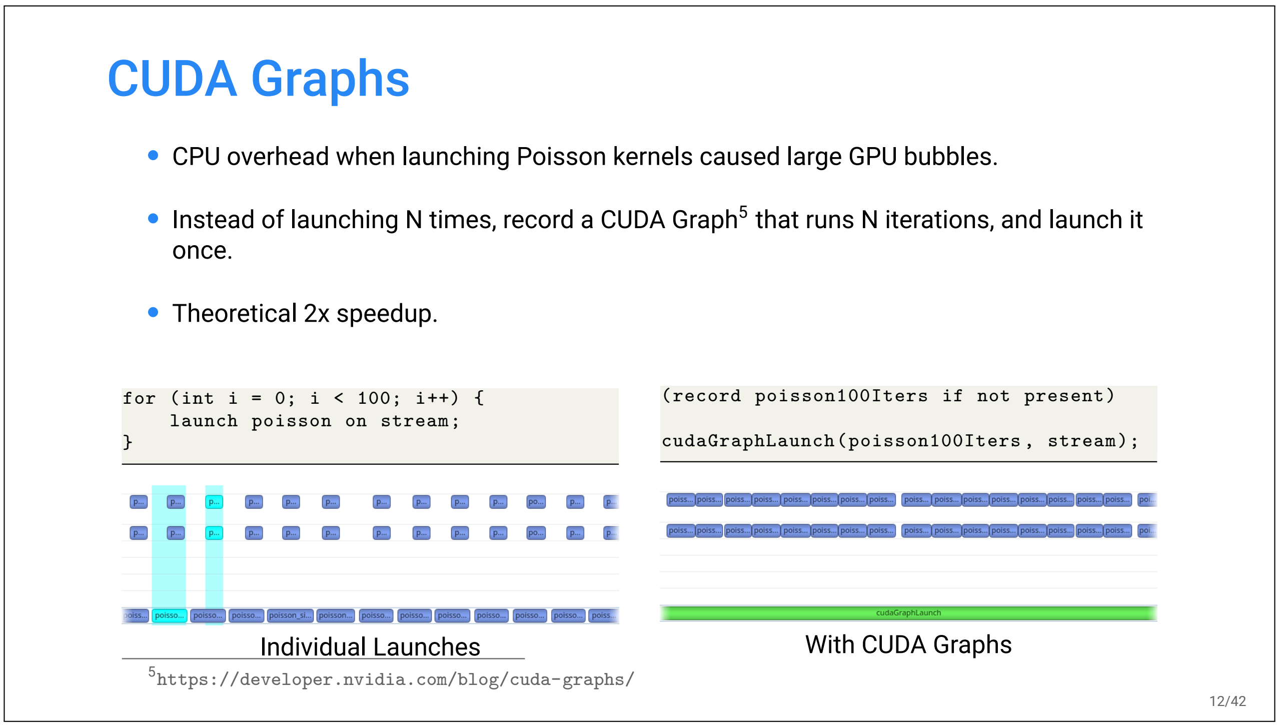


Figure 4: Example of parallel reduction for sum.
Reproduced from eximiaco.tech

11/42

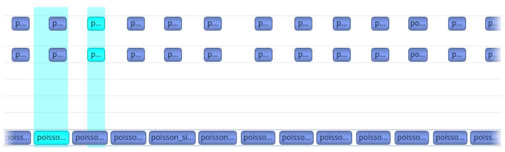


## CUDA Graphs

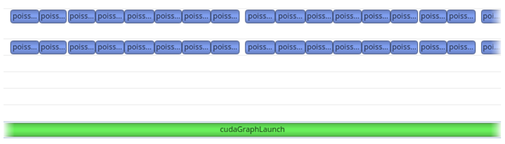
- CPU overhead when launching Poisson kernels caused large GPU bubbles.
- Instead of launching N times, record a CUDA Graph[5] that runs N iterations, and launch it once.
- Theoretical 2x speedup.

```
for (int i = 0; i < 100; i++) {
    launch poisson on stream;
}
```

```
(record poisson100Iters if not present)

cudaGraphLaunch(poisson100Iters, stream);
```



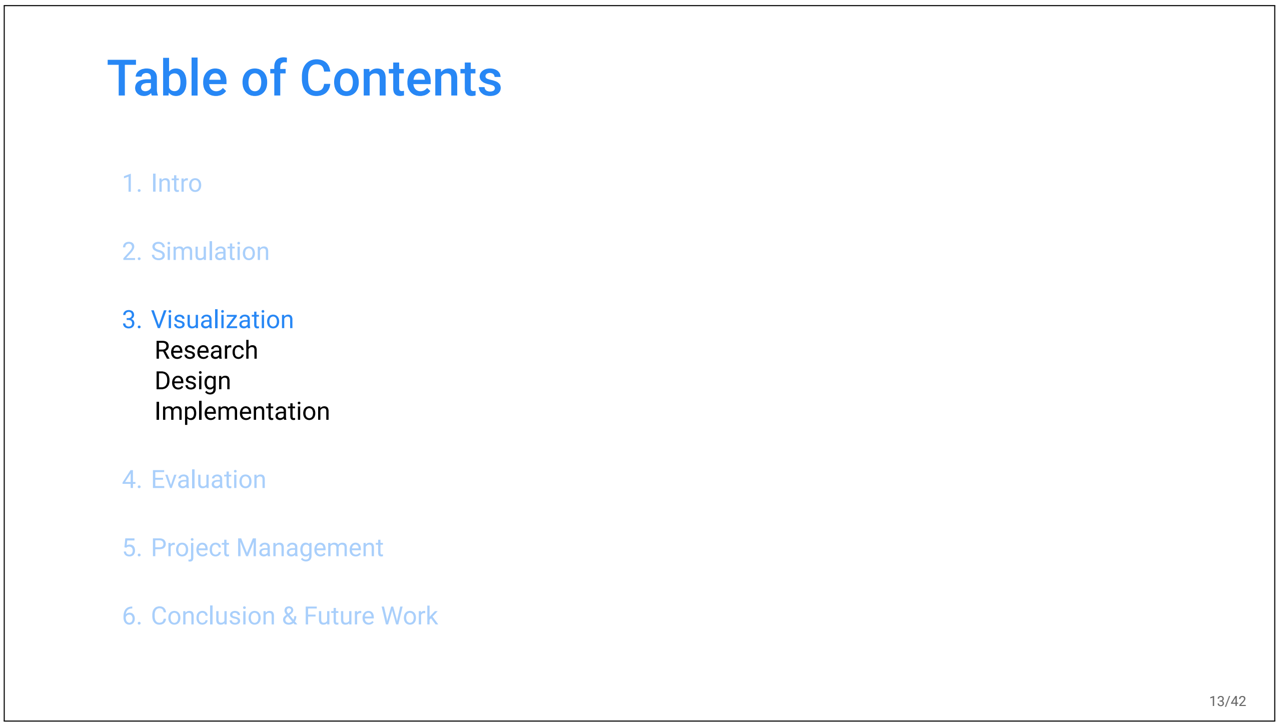
Individual Launches



With CUDA Graphs

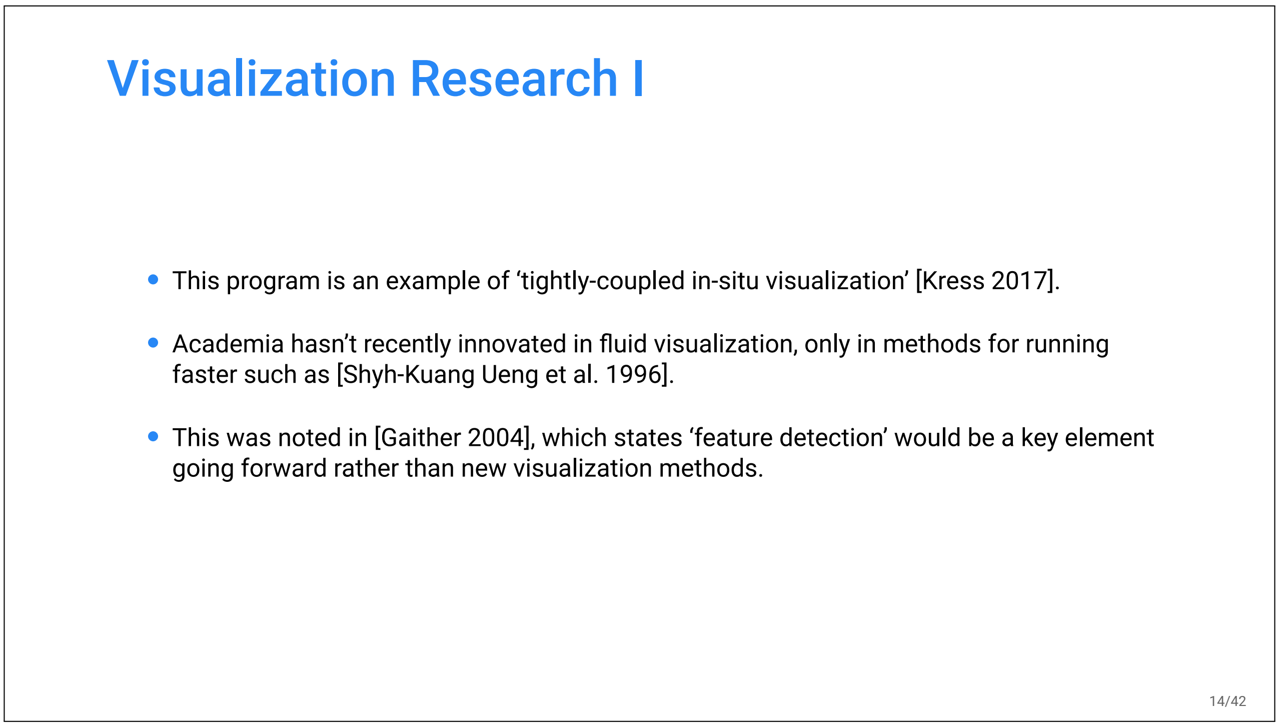
[5] https://developer.nvidia.com/blog/cuda-graphs/

12/42



## Table of Contents

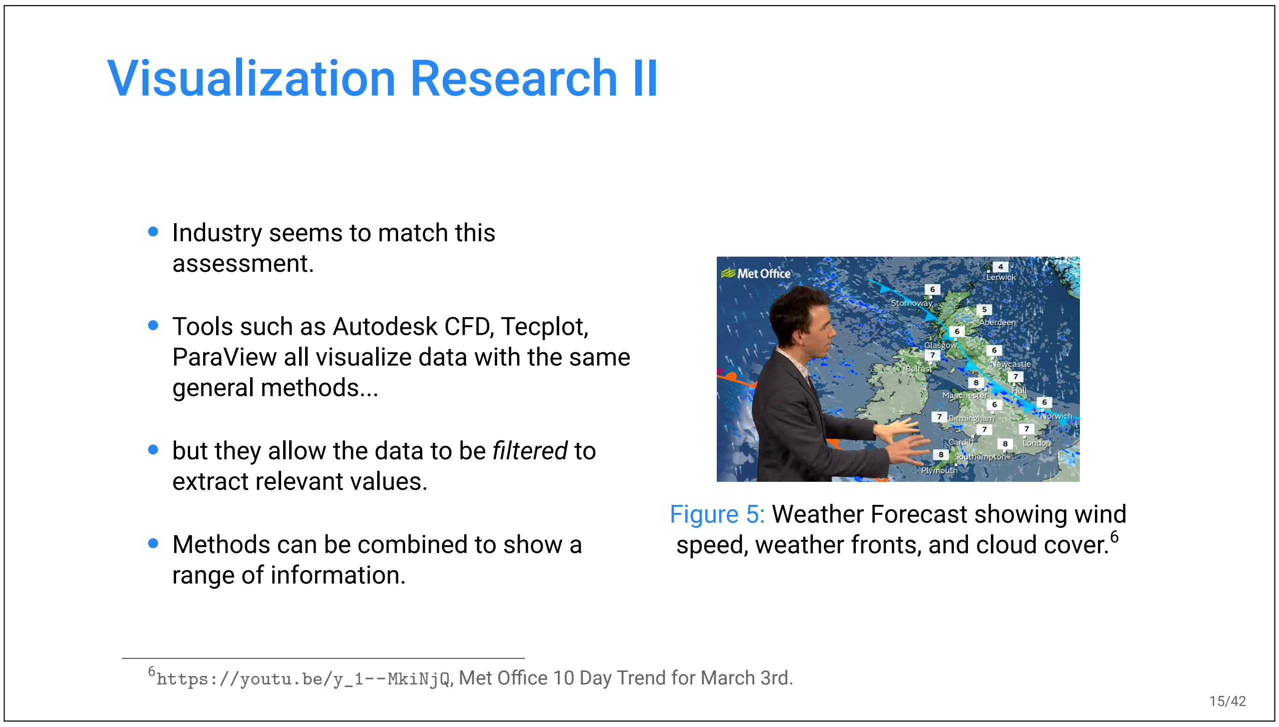
13/42



## Visualization Research I

- This program is an example of ‘tightly-coupled in-situ visualization’ [Kress 2017].
- Academia hasn’t recently innovated in fluid visualization, only in methods for running faster such as [Shyh-Kuang Ueng et al. 1996].
- This was noted in [Gaither 2004], which states ‘feature detection’ would be a key element going forward rather than new visualization methods.

14/42



## Visualization Research II

- Industry seems to match this assessment.
- Tools such as Autodesk CFD, Tecplot, ParaView all visualize data with the same general methods...
- but they allow the data to be *filtered* to extract relevant values.
- Methods can be combined to show a range of information.

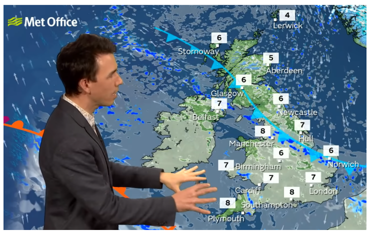


Figure 5: Weather Forecast showing wind speed, weather fronts, and cloud cover.[6]

[6]`https://youtu.be/y_1--MkiNjQ`, Met Office 10 Day Trend for March 3rd.

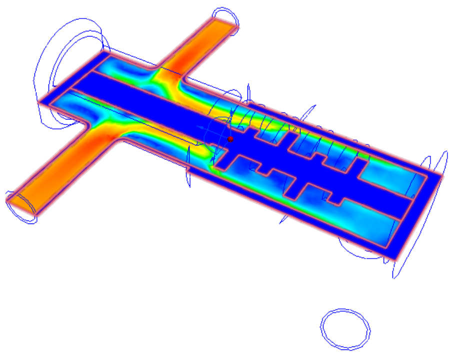
15/42

## Visualization Research

What can Autodesk CFD do?

### Result Planes - Scalar

- Place a plane in 3D space
- Select a scalar quantity (pressure, temperature etc.)
- The cross-section of the model shows the selected quantity, with a color scale



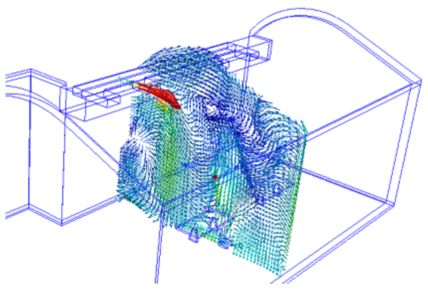
16/42

## Visualization Research

What can Autodesk CFD do?

### Result Planes - Vector

- Place a plane in 3D space
- Select a *vector* quantity (velocity etc.)
- The cross-section of the model shows a vector field of the selected velocity.



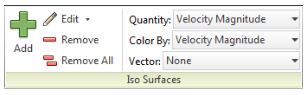
17/42

## Visualization Research

What can Autodesk CFD do?

### Isosurfaces

- Select a scalar quantity $X$.
- Select a value $X = x$.
- This surface is displayed with a color based on another quantity $Y$.
- A vector quantity can also be added to the surface.



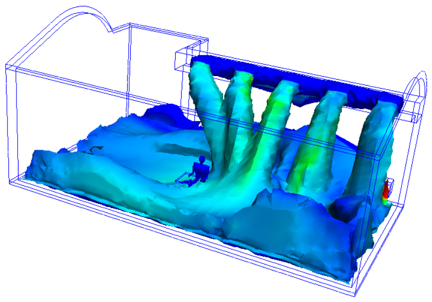
Add | Edit | Remove | Remove All

Quantity: Velocity Magnitude

Color By: Velocity Magnitude

Vector: None

Iso Surfaces

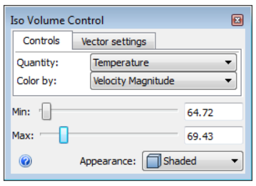


## Visualization Research

What can Autodesk CFD do?

### Isovolumes

- Select a scalar quantity $X$.
- Select a *range* $x_{min} \leq X \leq x_{max}$.
- This *volume* is displayed with a color based on another quantity $Y$.
- A vector quantity can also be added to the *volume*.



Iso Volume Control

Controls | Vector settings

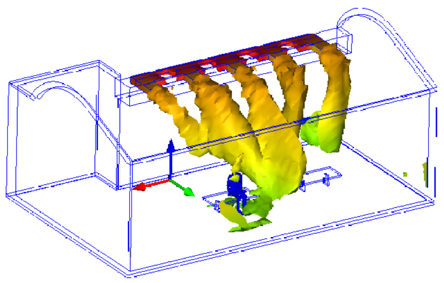
Quantity: Temperature

Color by: Velocity Magnitude

Min: 64.72

Max: 69.43

Appearance: Shaded

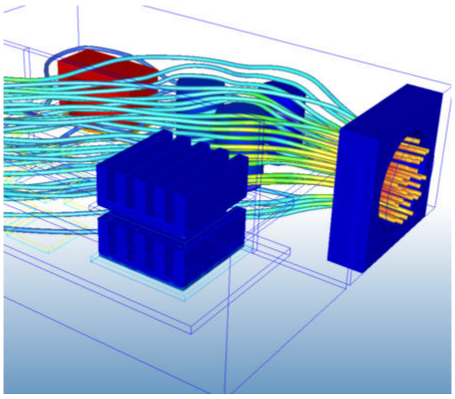


## Visualization Research

What can Autodesk CFD do?

### Particles

- Place particle spawn points ('seeds').
- Select a scalar quantity to display, or a solid color.
- Points along the particle paths show the specified quantity.
- Can choose many kinds of path:
  - Cylinders
  - Ribbons
  - Comets
  - etc.



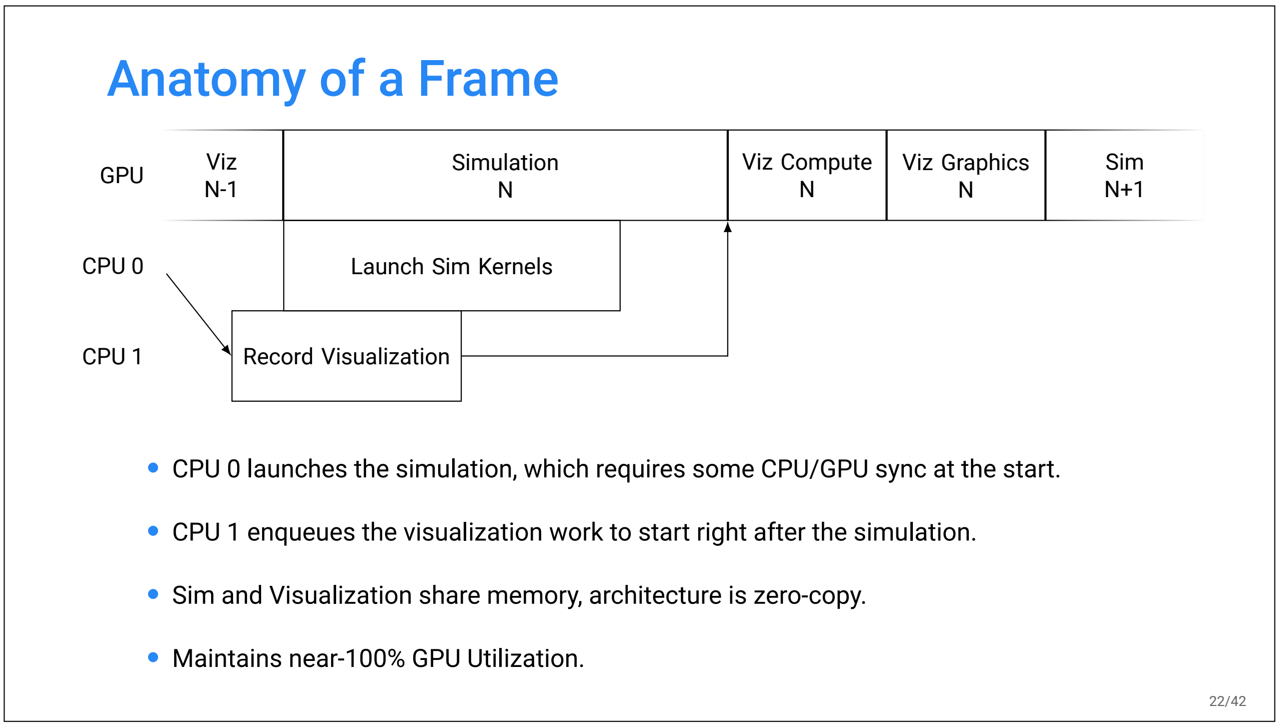
20/42

## Selected Features

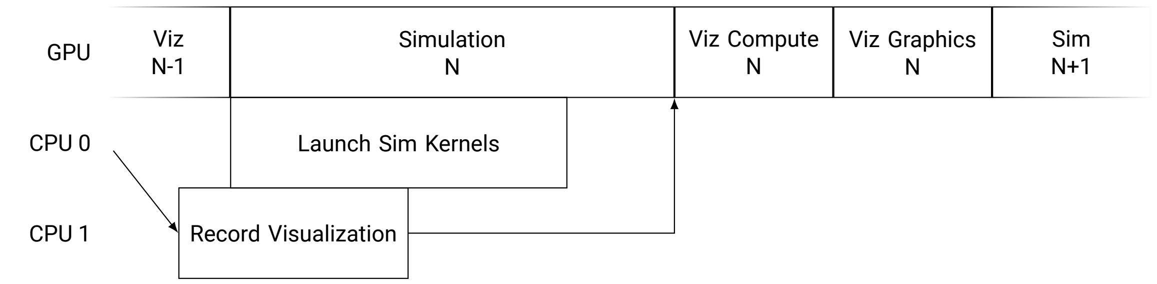
Separate the visualization into layers:

- Background
- Scalar Quantity
  - Display a quantity $X$ using a colormap when $x_{min} \leq X \leq x_{max}$
  - Allow the user to select a range, or calculate a range containing all values
  - Equivalent to Results Plane (Scalar) + 2D Isovolume
- Vector Quantity
  - Display a vector field of $X$ when $x_{min} \leq X \leq x_{max}$
  - Allow the user to select a range, or calculate a range containing all values
  - Equivalent to Results Plane (Vector) + 2D Isovolume
- Particles
  - Editable 'seeds'
  - Planned for particle trace options, didn't have time.

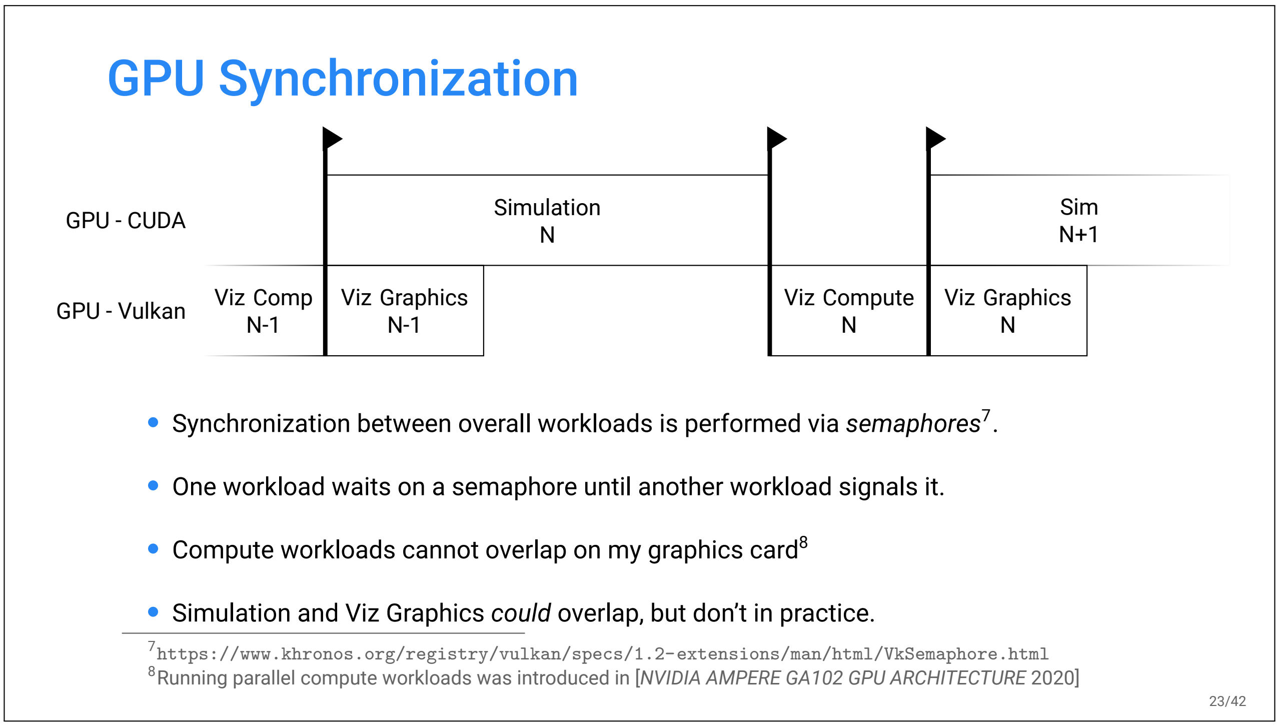
21/42



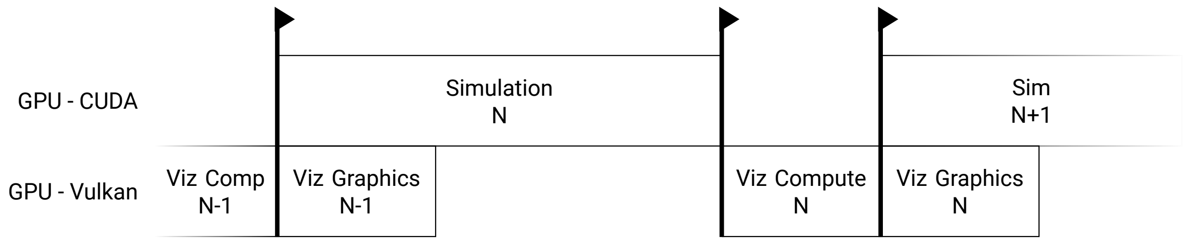
## Anatomy of a Frame



- CPU 0 launches the simulation, which requires some CPU/GPU sync at the start.
- CPU 1 enqueues the visualization work to start right after the simulation.
- Sim and Visualization share memory, architecture is zero-copy.
- Maintains near-100% GPU Utilization.

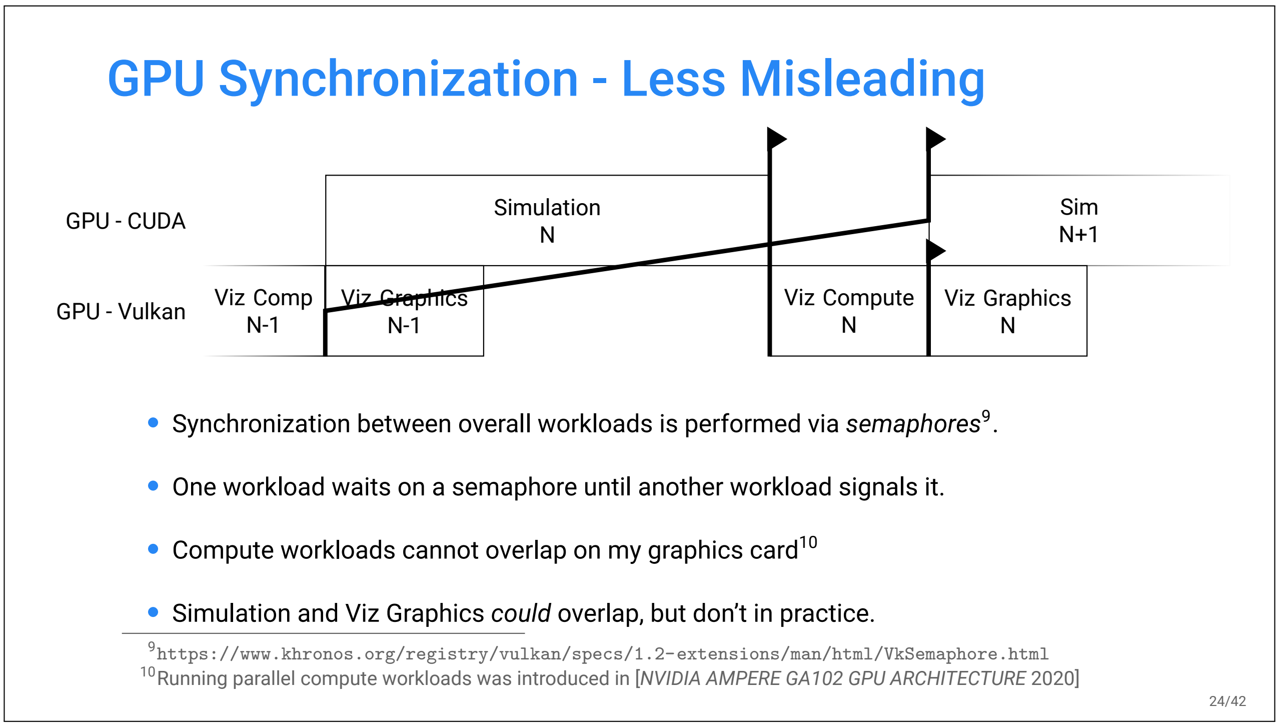


## GPU Synchronization

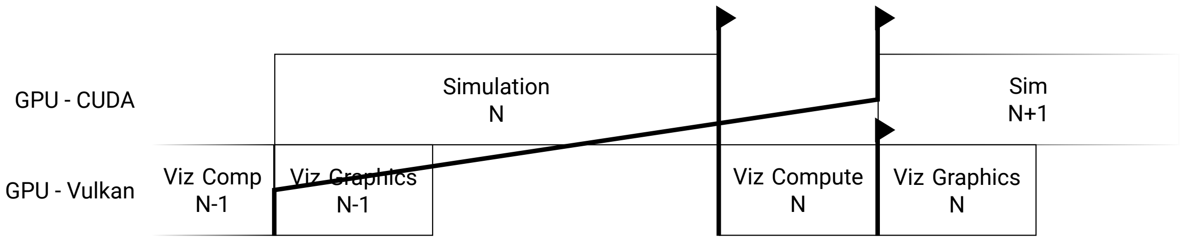


- Synchronization between overall workloads is performed via *semaphores*[7].
- One workload waits on a semaphore until another workload signals it.
- Compute workloads cannot overlap on my graphics card[8]
- Simulation and Viz Graphics *could* overlap, but don't in practice.

[7] `https://www.khronos.org/registry/vulkan/specs/1.2-extensions/man/html/VkSemaphore.html`
[8] Running parallel compute workloads was introduced in [*NVIDIA AMPERE GA102 GPU ARCHITECTURE* 2020]



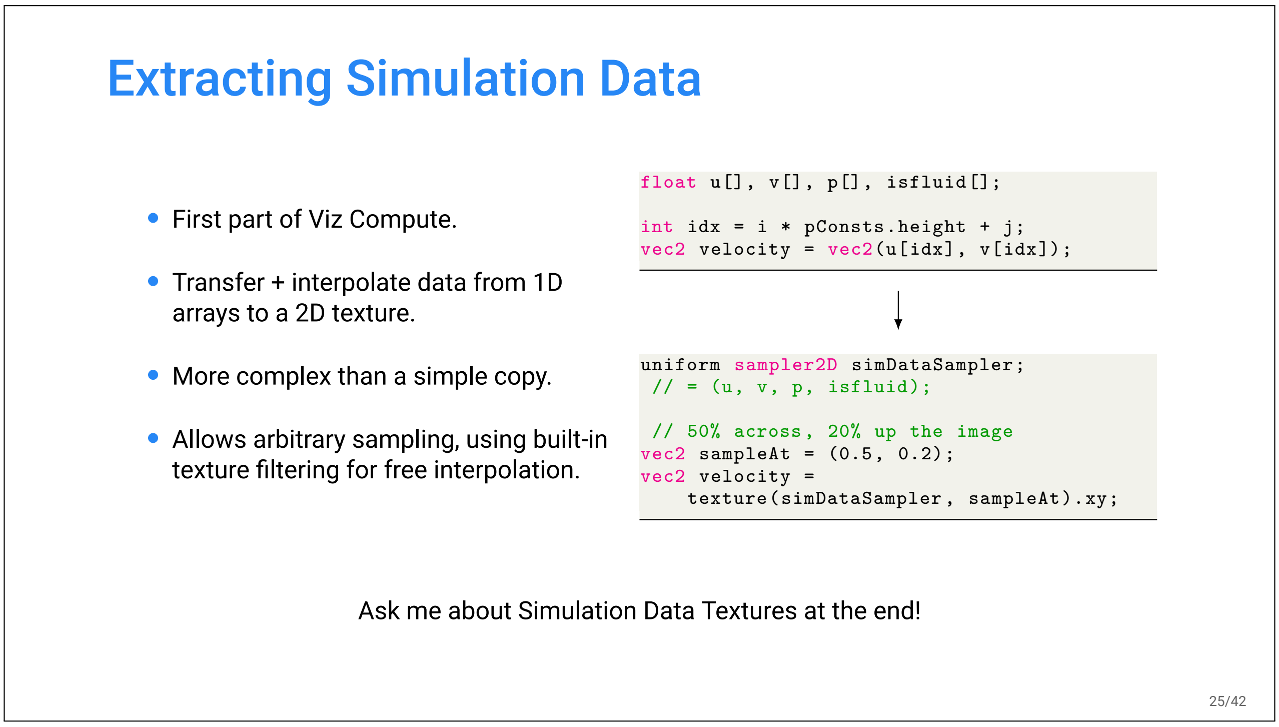
## GPU Synchronization - Less Misleading



- Synchronization between overall workloads is performed via *semaphores*[9].
- One workload waits on a semaphore until another workload signals it.
- Compute workloads cannot overlap on my graphics card[10]
- Simulation and Viz Graphics *could* overlap, but don't in practice.

[9] `https://www.khronos.org/registry/vulkan/specs/1.2-extensions/man/html/VkSemaphore.html`
[10] Running parallel compute workloads was introduced in [*NVIDIA AMPERE GA102 GPU ARCHITECTURE* 2020]

24/42



## Extracting Simulation Data

- First part of Viz Compute.
- Transfer + interpolate data from 1D arrays to a 2D texture.
- More complex than a simple copy.
- Allows arbitrary sampling, using built-in texture filtering for free interpolation.

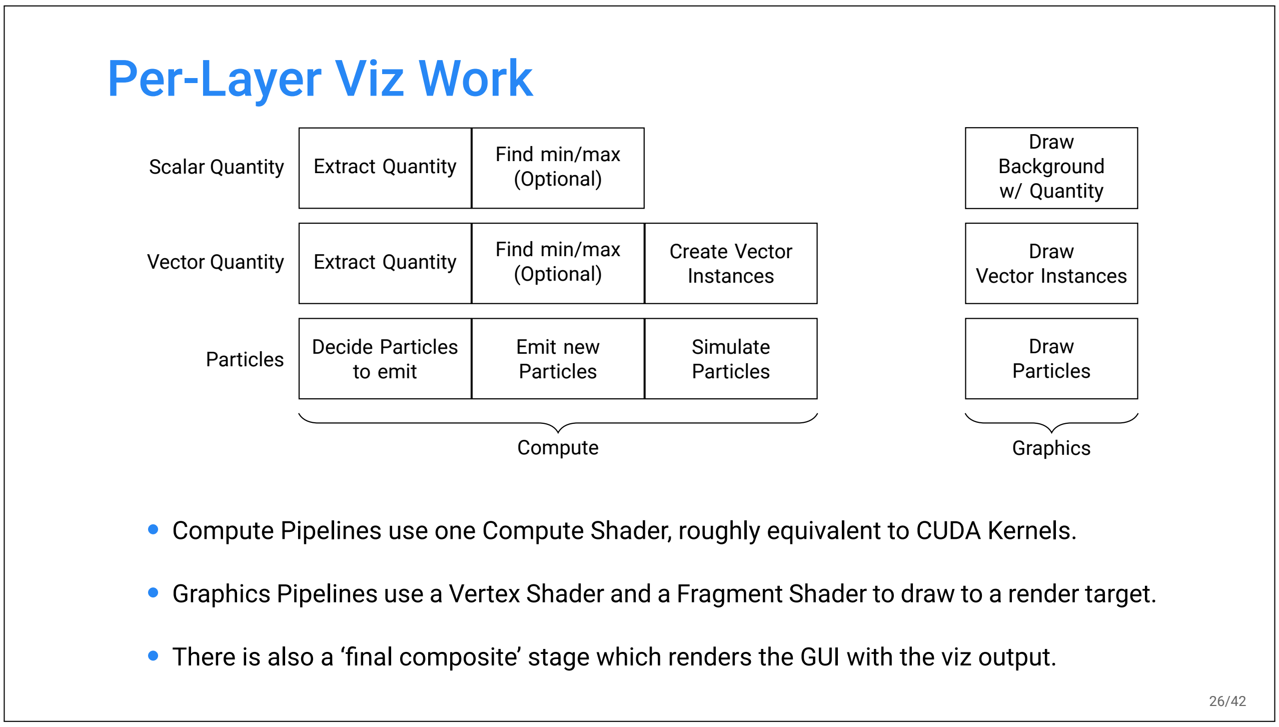
```
float u[], v[], p[], isfluid[];

int idx = i * pConsts.height + j;
vec2 velocity = vec2(u[idx], v[idx]);
```

↓

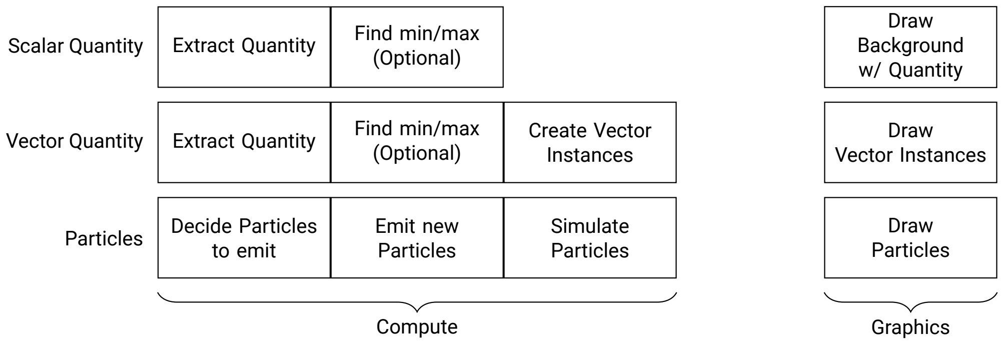
```
uniform sampler2D simDataSampler;
 // = (u, v, p, isfluid);

 // 50% across, 20% up the image
vec2 sampleAt = (0.5, 0.2);
vec2 velocity =
    texture(simDataSampler, sampleAt).xy;
```

Ask me about Simulation Data Textures at the end!

25/42

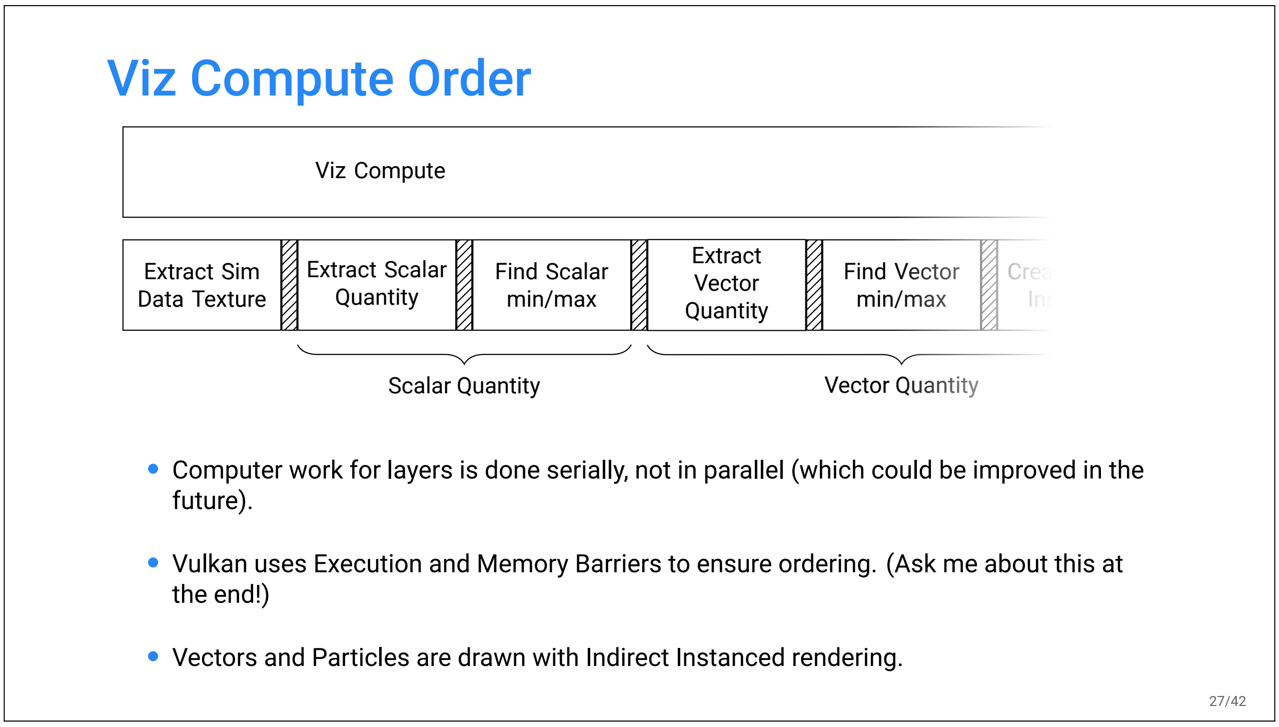


## Per-Layer Viz Work

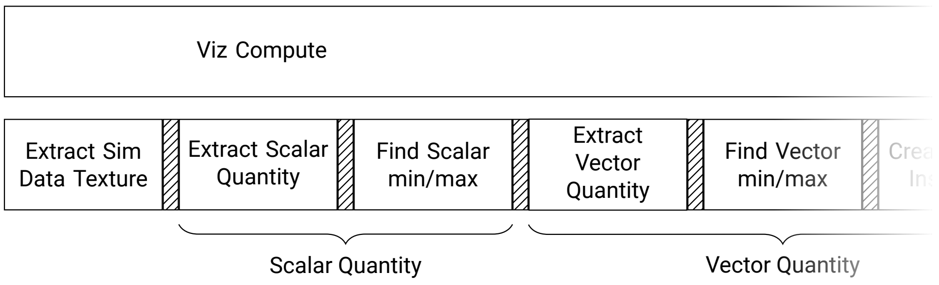


- Compute Pipelines use one Compute Shader, roughly equivalent to CUDA Kernels.
- Graphics Pipelines use a Vertex Shader and a Fragment Shader to draw to a render target.
- There is also a 'final composite' stage which renders the GUI with the viz output.

26/42



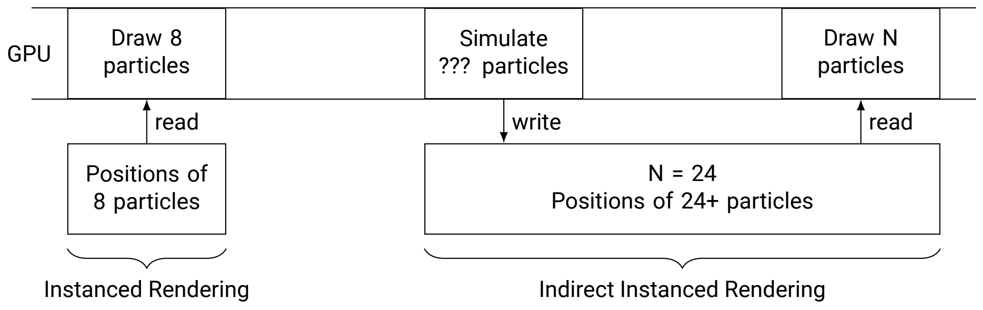
## Viz Compute Order



- Computer work for layers is done serially, not in parallel (which could be improved in the future).
- Vulkan uses Execution and Memory Barriers to ensure ordering. (Ask me about this at the end!)
- Vectors and Particles are drawn with Indirect Instanced rendering.

27/42

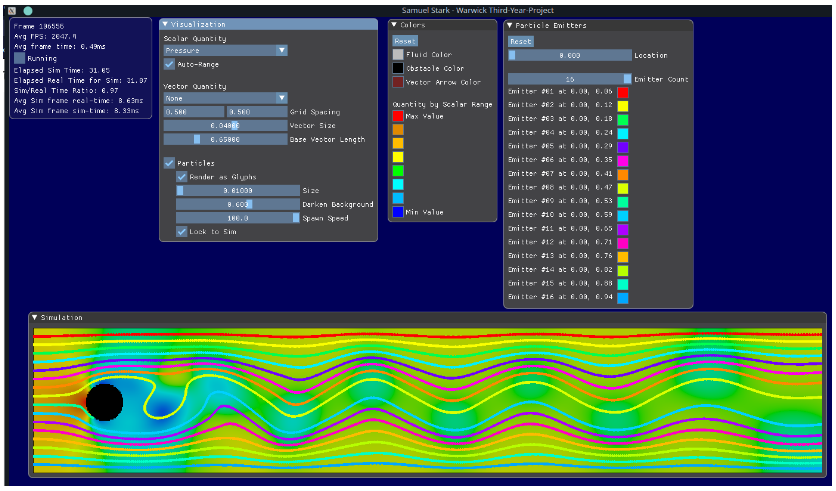
## Indirect Instanced Rendering



- We don't know how many Vectors/Particles exist at record time.
- Tell the GPU to look somewhere in memory to find how many copies to render.

Ask me about indirect/instanced/indexed rendering at the end!

## Result!


Samuel Stark - Warwick Third-Year-Project

```
Frame 106556
Avg FPS: 2047.8
Avg frame time: 0.49ms
Running
Elapsed Sim Time: 31.05
Elapsed Real Time for Sim: 31.87
Sim/Real Time Ratio: 0.97
Avg Sim frame real-time: 8.63ms
Avg Sim frame sim-time: 8.33ms
```

```
▼ Visualization
Scalar Quantity
Pressure
✓ Auto-Range

Vector Quantity
None
0.500   0.500   Grid Spacing
0.04000         Vector Size
0.65000         Base Vector Length

✓ Particles
  ✓ Render as Glyphs
  0.01000       Size
  0.600         Darken Background
  100.0         Spawn Speed
  ✓ Lock to Sim
```

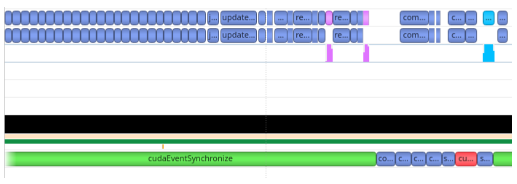
```
▼ Colors
Reset
Fluid Color
Obstacle Color
Vector Arrow Color

Quantity by Scalar Range
Max Value







Min Value
```

```
▼ Particle Emitters
Reset
0.000   Location

16      Emitter Count
Emitter #01 at 0.00, 0.06
Emitter #02 at 0.00, 0.12
Emitter #03 at 0.00, 0.18
Emitter #04 at 0.00, 0.24
Emitter #05 at 0.00, 0.29
Emitter #06 at 0.00, 0.35
Emitter #07 at 0.00, 0.41
Emitter #08 at 0.00, 0.47
Emitter #09 at 0.00, 0.53
Emitter #10 at 0.00, 0.59
Emitter #11 at 0.00, 0.65
Emitter #12 at 0.00, 0.71
Emitter #13 at 0.00, 0.76
Emitter #14 at 0.00, 0.82
Emitter #15 at 0.00, 0.88
Emitter #16 at 0.00, 0.94
```

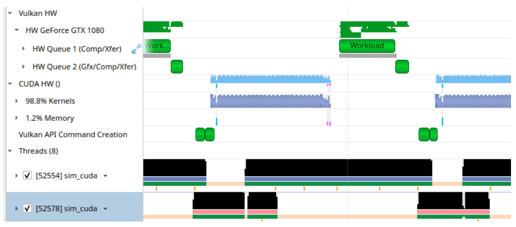
## Table of Contents

## GPU Utilization

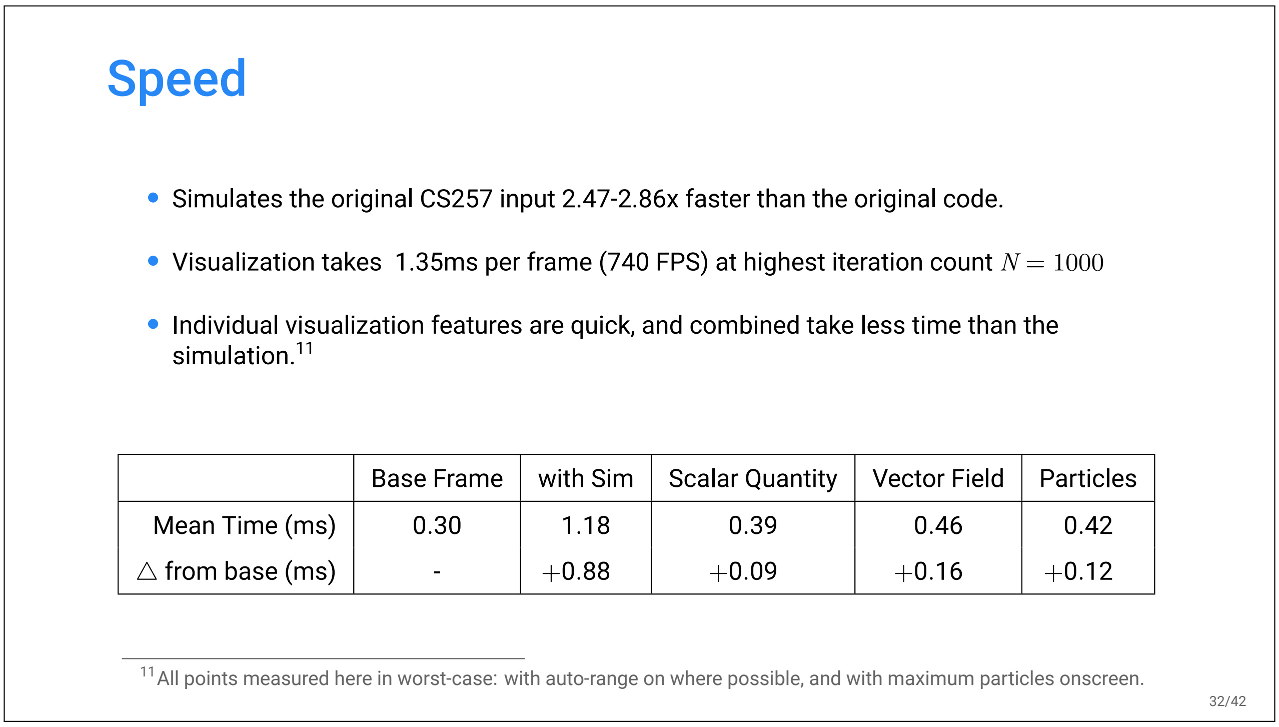
- GPU Utilization is close to 100% where possible.
- At tick boundaries some bubbles appear as the CPU calculates the next $\delta t$.
- When visualizing, the Vulkan work hides this.



Tick Boundary



Overall Visualization Pipeline



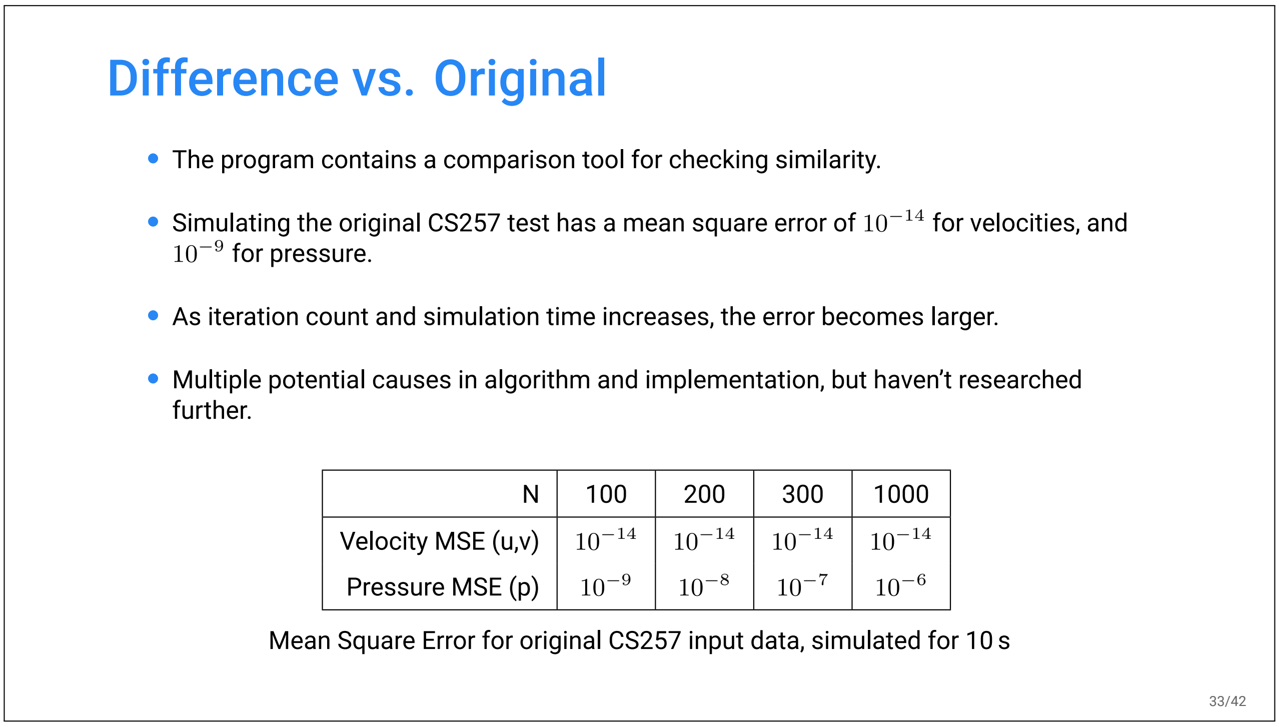
## Speed

- Simulates the original CS257 input 2.47-2.86x faster than the original code.
- Visualization takes 1.35ms per frame (740 FPS) at highest iteration count $N = 1000$
- Individual visualization features are quick, and combined take less time than the simulation.[11]

| | Base Frame | with Sim | Scalar Quantity | Vector Field | Particles |
|---|---|---|---|---|---|
| Mean Time (ms) | 0.30 | 1.18 | 0.39 | 0.46 | 0.42 |
| $\triangle$ from base (ms) | - | $+$0.88 | $+$0.09 | $+$0.16 | $+$0.12 |

[11] All points measured here in worst-case: with auto-range on where possible, and with maximum particles onscreen.

32/42



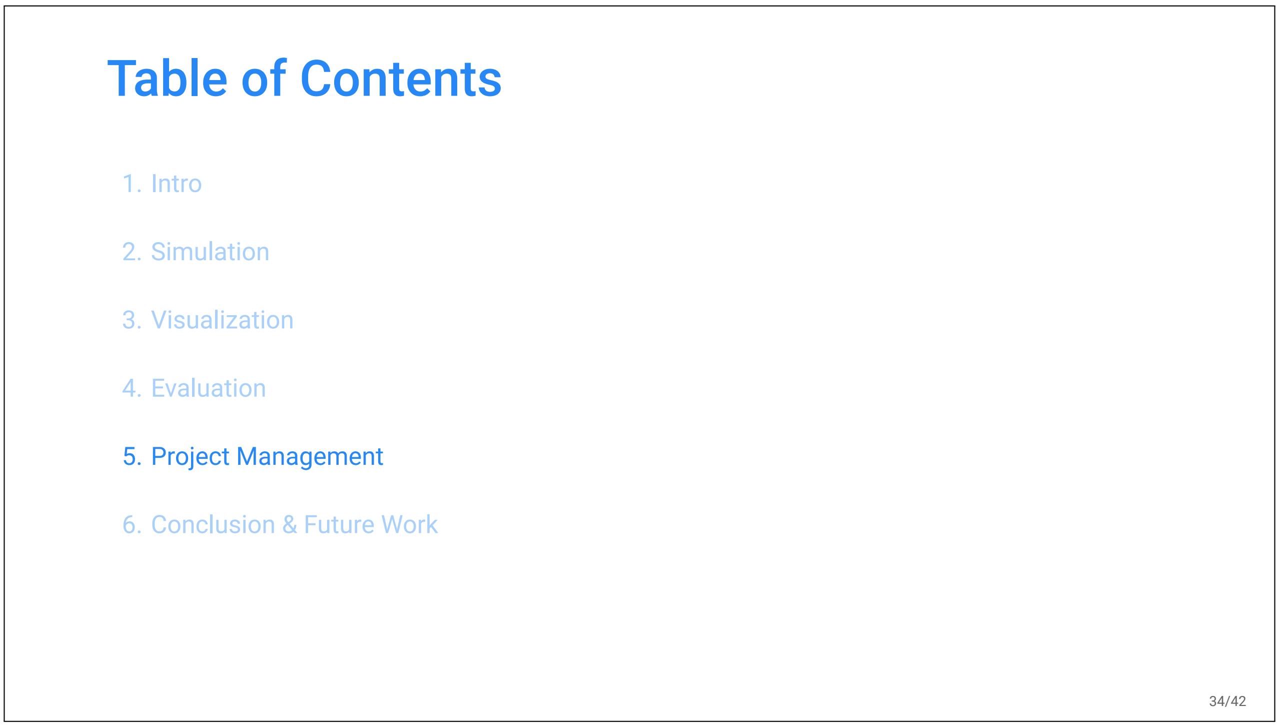
## Difference vs. Original

- The program contains a comparison tool for checking similarity.
- Simulating the original CS257 test has a mean square error of $10^{-14}$ for velocities, and $10^{-9}$ for pressure.
- As iteration count and simulation time increases, the error becomes larger.
- Multiple potential causes in algorithm and implementation, but haven't researched further.

| N | 100 | 200 | 300 | 1000 |
|---|---|---|---|---|
| Velocity MSE (u,v) | $10^{-14}$ | $10^{-14}$ | $10^{-14}$ | $10^{-14}$ |
| Pressure MSE (p) | $10^{-9}$ | $10^{-8}$ | $10^{-7}$ | $10^{-6}$ |

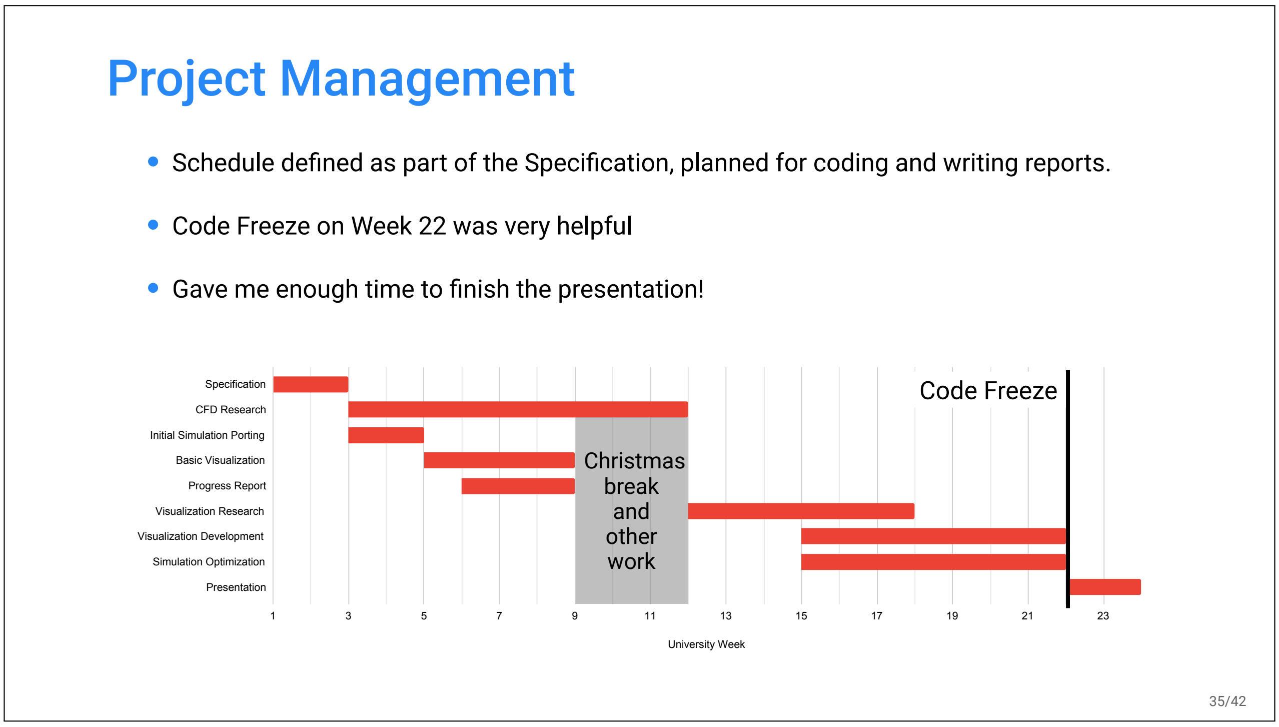
Mean Square Error for original CS257 input data, simulated for 10 s

33/42



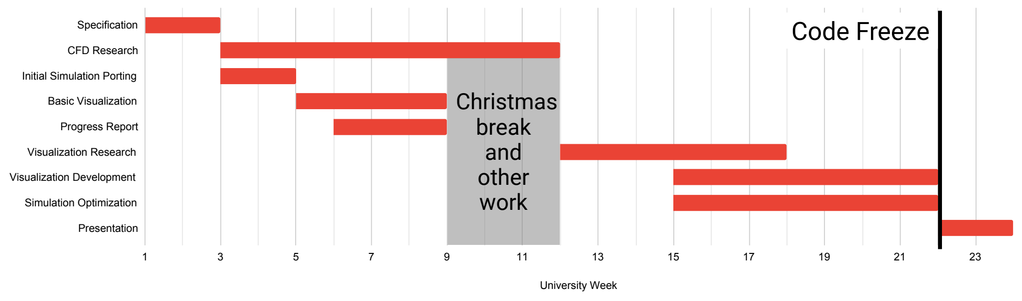
## Table of Contents

34/42

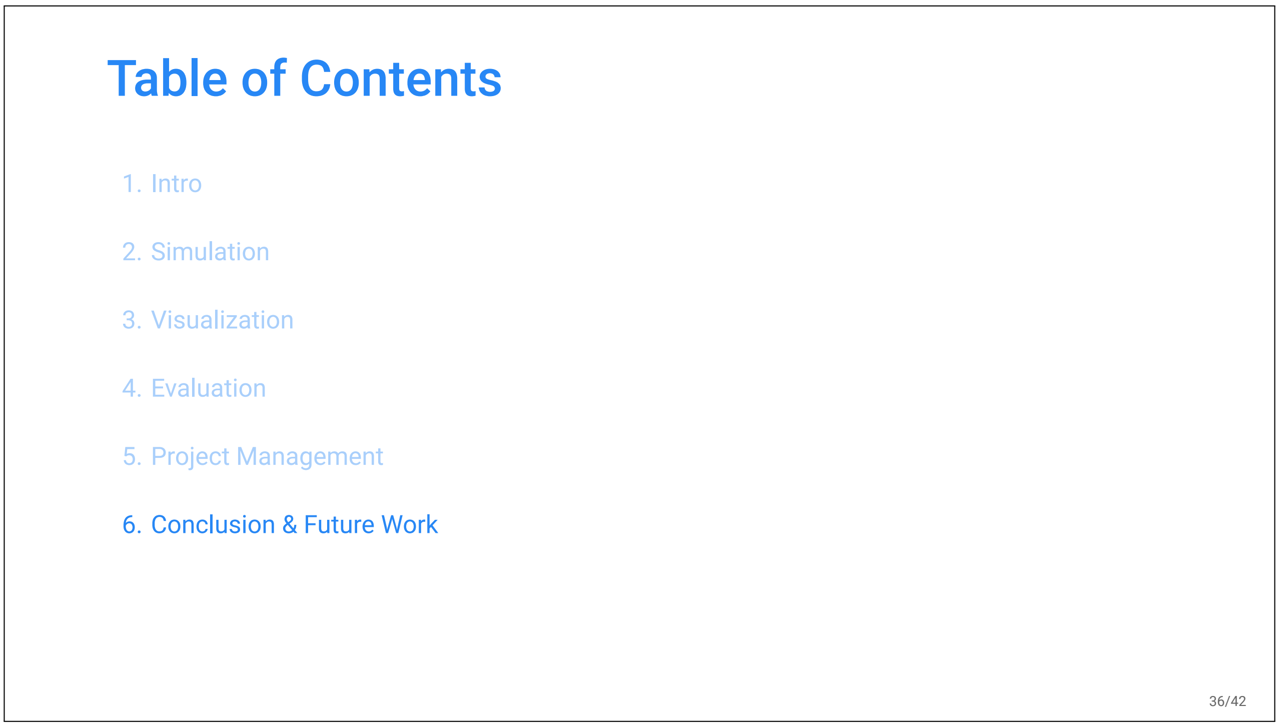


## Project Management

- Schedule defined as part of the Specification, planned for coding and writing reports.
- Code Freeze on Week 22 was very helpful
- Gave me enough time to finish the presentation!

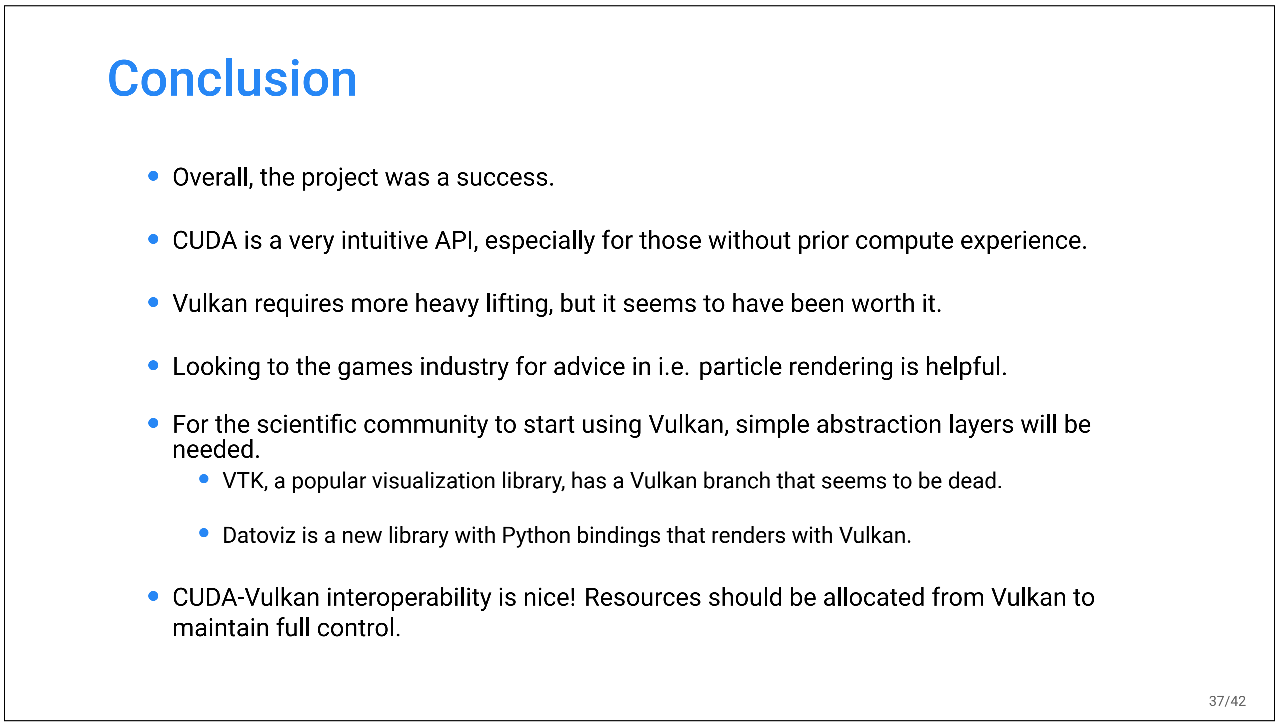


35/42



## Table of Contents

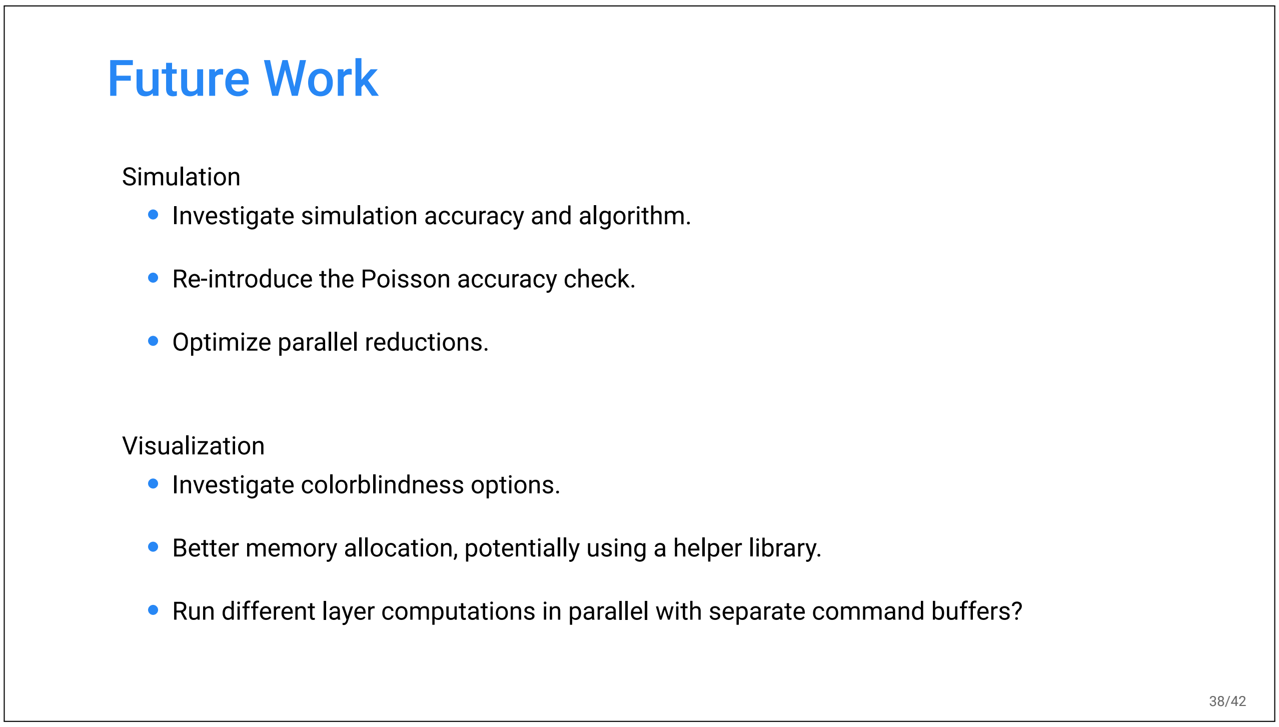
36/42



## Conclusion

- Overall, the project was a success.
- CUDA is a very intuitive API, especially for those without prior compute experience.
- Vulkan requires more heavy lifting, but it seems to have been worth it.
- Looking to the games industry for advice in i.e. particle rendering is helpful.
- For the scientific community to start using Vulkan, simple abstraction layers will be needed.
    - VTK, a popular visualization library, has a Vulkan branch that seems to be dead.
    - Datoviz is a new library with Python bindings that renders with Vulkan.
- CUDA-Vulkan interoperability is nice! Resources should be allocated from Vulkan to maintain full control.

37/42



## Future Work

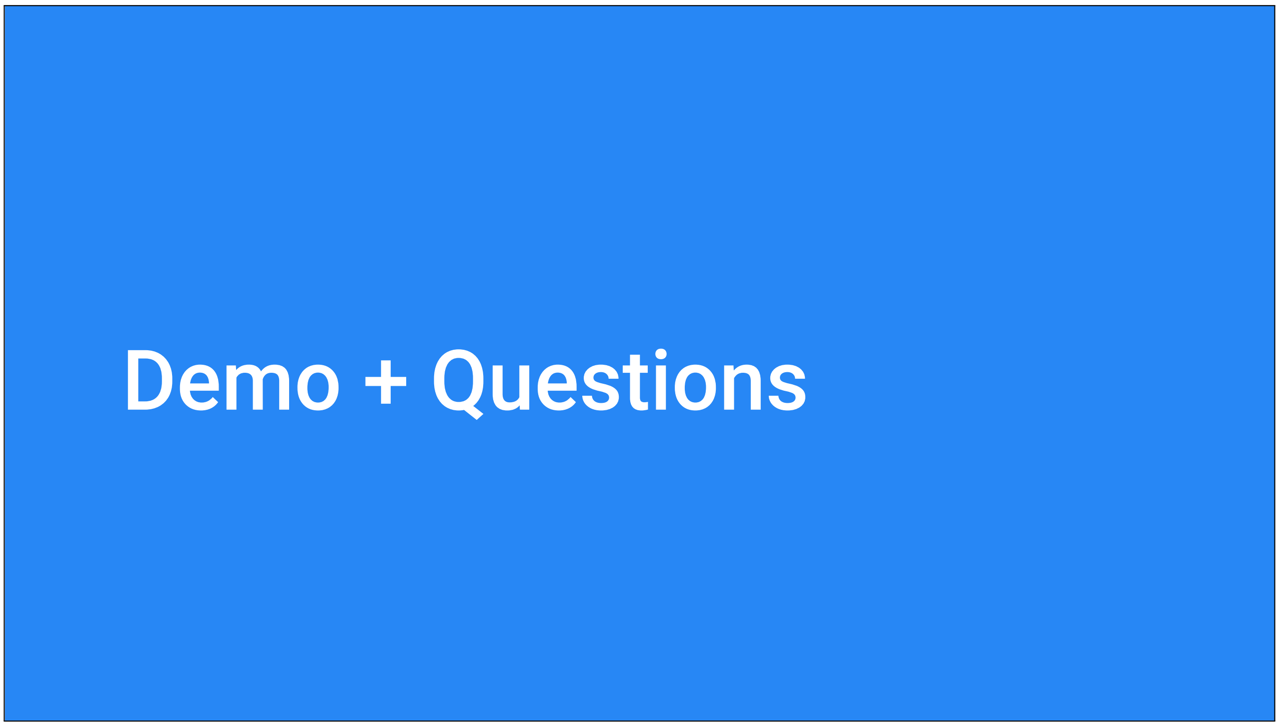
Simulation

- Investigate simulation accuracy and algorithm.
- Re-introduce the Poisson accuracy check.
- Optimize parallel reductions.

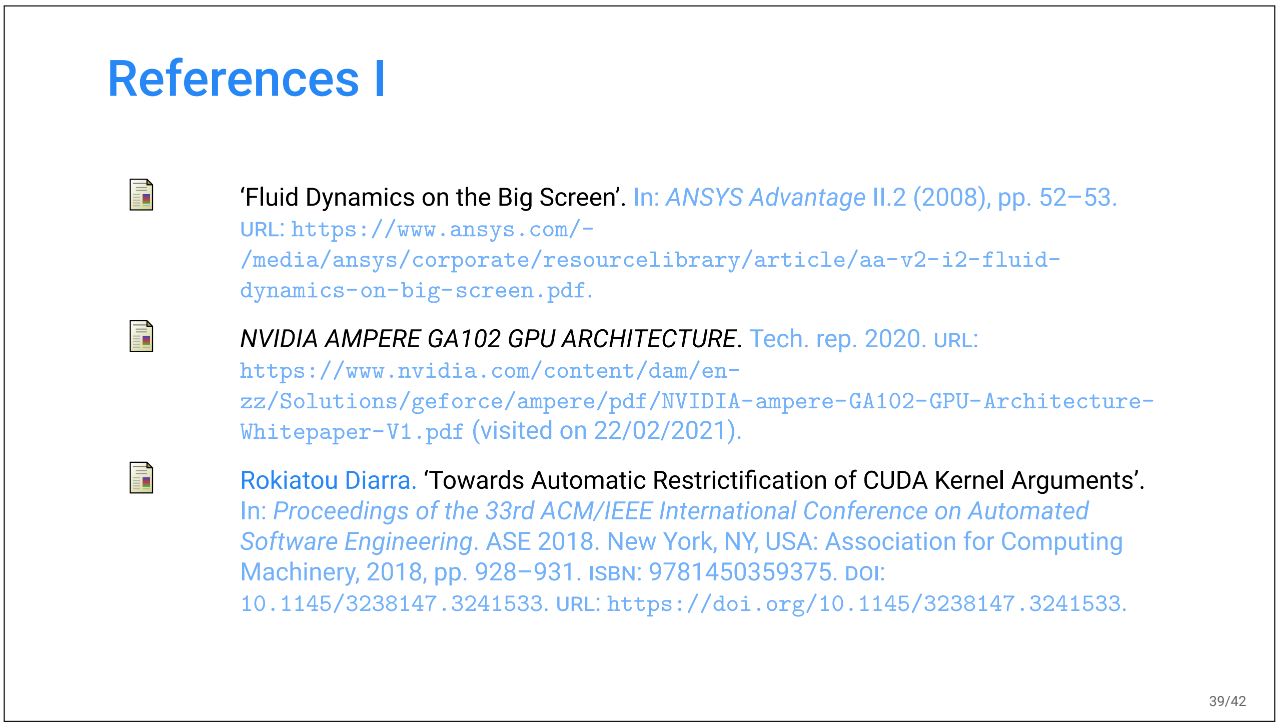
Visualization

- Investigate colorblindness options.
- Better memory allocation, potentially using a helper library.
- Run different layer computations in parallel with separate command buffers?

38/42



## Demo + Questions



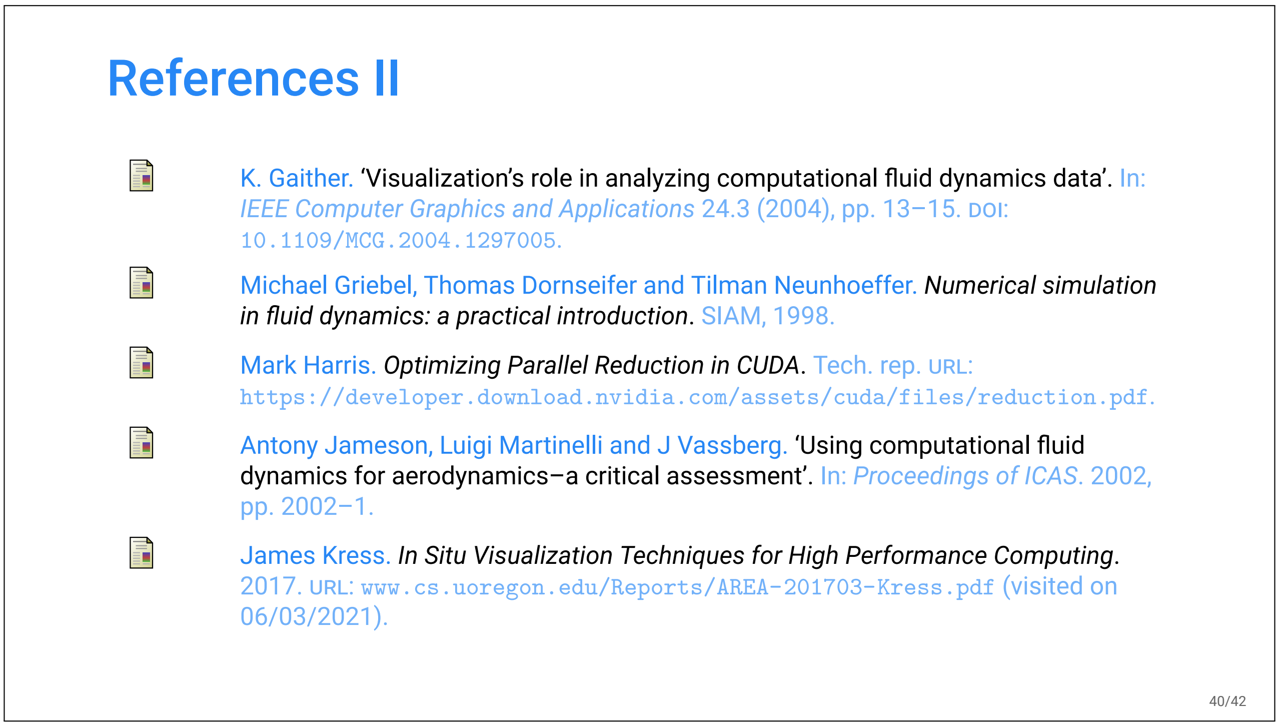
## References I

'Fluid Dynamics on the Big Screen'. In: *ANSYS Advantage* II.2 (2008), pp. 52–53. URL: `https://www.ansys.com/-/media/ansys/corporate/resourcelibrary/article/aa-v2-i2-fluid-dynamics-on-big-screen.pdf`.

*NVIDIA AMPERE GA102 GPU ARCHITECTURE*. Tech. rep. 2020. URL: `https://www.nvidia.com/content/dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.pdf` (visited on 22/02/2021).

Rokiatou Diarra. 'Towards Automatic Restrictification of CUDA Kernel Arguments'. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE 2018. New York, NY, USA: Association for Computing Machinery, 2018, pp. 928–931. ISBN: 9781450359375. DOI: `10.1145/3238147.3241533`. URL: `https://doi.org/10.1145/3238147.3241533`.

39/42



## References II

K. Gaither. 'Visualization's role in analyzing computational fluid dynamics data'. In: *IEEE Computer Graphics and Applications* 24.3 (2004), pp. 13–15. DOI: `10.1109/MCG.2004.1297005`.

Michael Griebel, Thomas Dornseifer and Tilman Neunhoeffer. *Numerical simulation in fluid dynamics: a practical introduction*. SIAM, 1998.

Mark Harris. *Optimizing Parallel Reduction in CUDA*. Tech. rep. URL: `https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf`.

Antony Jameson, Luigi Martinelli and J Vassberg. 'Using computational fluid dynamics for aerodynamics–a critical assessment'. In: *Proceedings of ICAS*. 2002, pp. 2002–1.

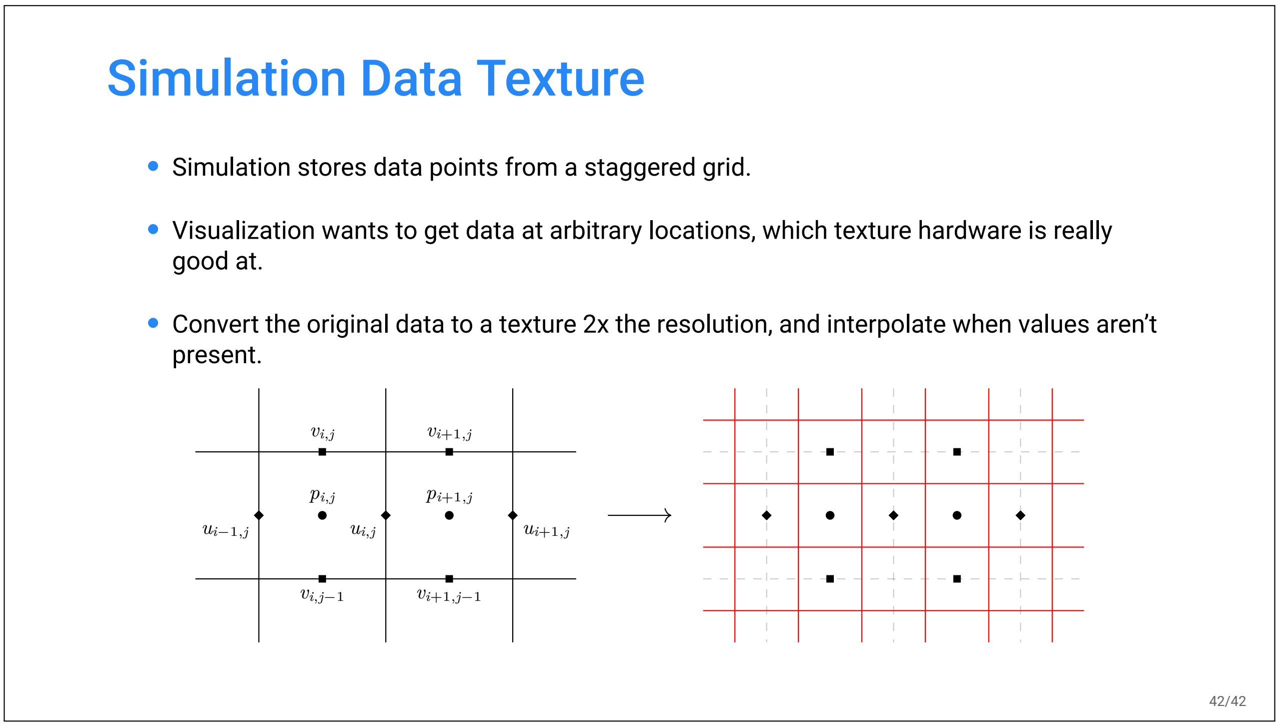
James Kress. *In Situ Visualization Techniques for High Performance Computing*. 2017. URL: `www.cs.uoregon.edu/Reports/AREA-201703-Kress.pdf` (visited on 06/03/2021).

40/42

## References III

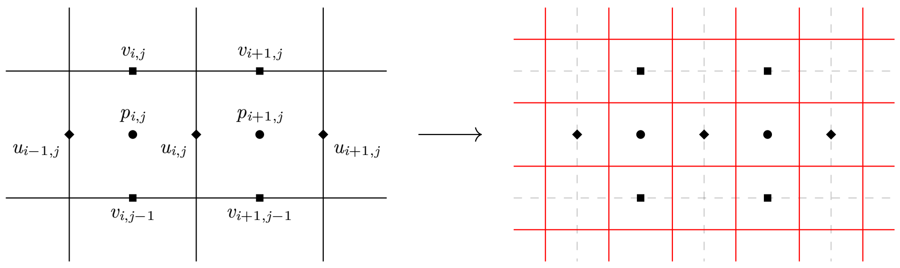
- Medvecký-Heretik Jakub. ‘Real-time Water Simulation in Game Environment’. PhD thesis. Masaryk University, Faculty of Informatics, 2018.
- Shyh-Kuang Ueng, C. Sikorski and Kwan-Liu Ma. ‘Efficient streamline, streamribbon, and streamtube constructions on unstructured grids’. In: *IEEE Transactions on Visualization and Computer Graphics* 2.2 (1996), pp. 100–110. DOI: 10.1109/2945.506222.
- S. Stark. ‘CS257 Report - Reducing the Execution Time of a Fluid Simulation Program’. In: (2020).
- Andrew L. Sullivan. ‘Wildland surface fire spread modelling, 1990 - 2007. 1: Physical and quasi-physical models’. In: *International Journal of Wildland Fire* 18.4 (2009), p. 349. ISSN: 1049-8001. DOI: 10.1071/wf06143. URL: http://dx.doi.org/10.1071/WF06143.

41/42



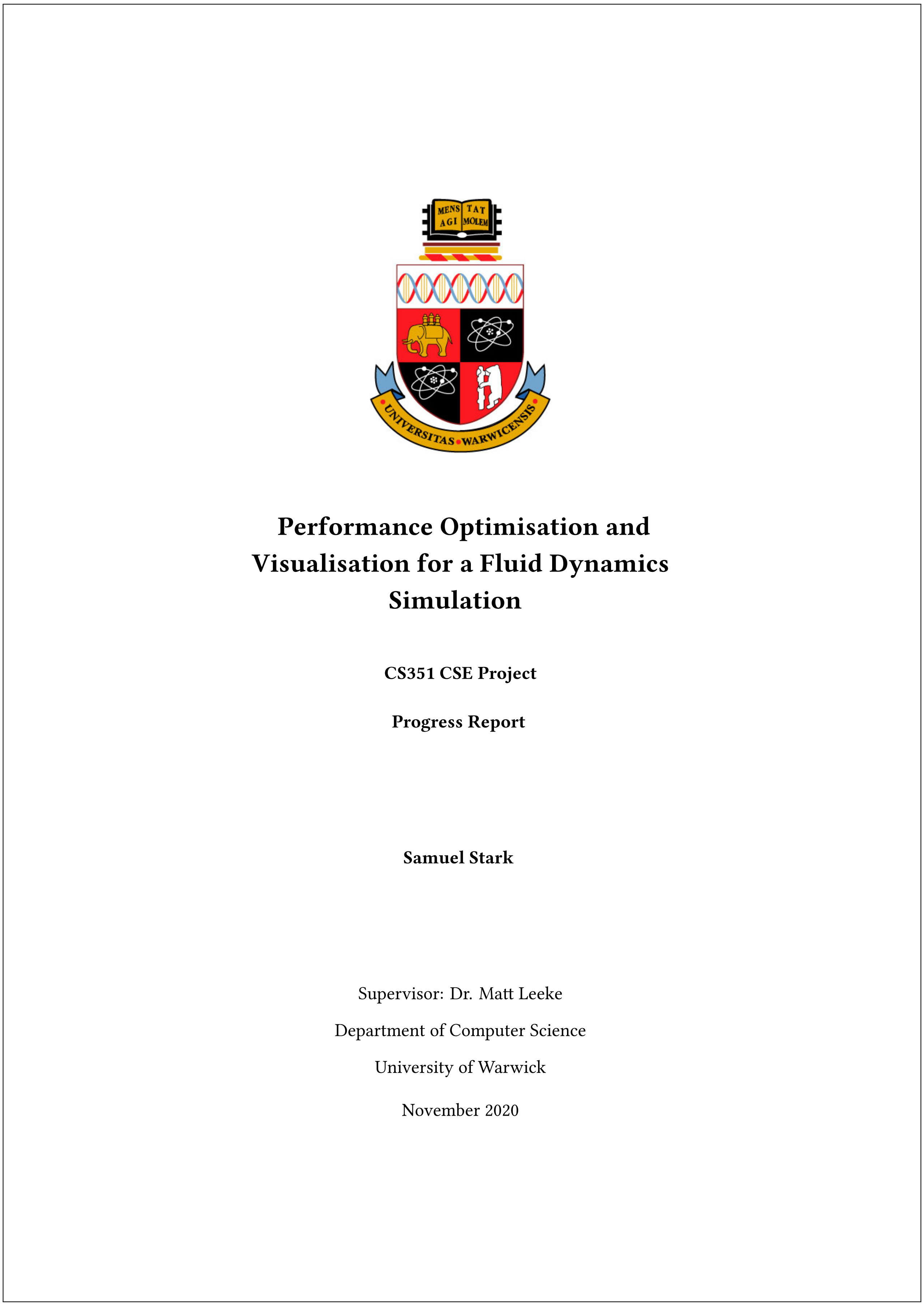
## Simulation Data Texture

- Simulation stores data points from a staggered grid.
- Visualization wants to get data at arbitrary locations, which texture hardware is really good at.
- Convert the original data to a texture 2x the resolution, and interpolate when values aren’t present.



42/42

## B.2 Progress Report



**Performance Optimisation and Visualisation for a Fluid Dynamics Simulation**

**CS351 CSE Project**

**Progress Report**

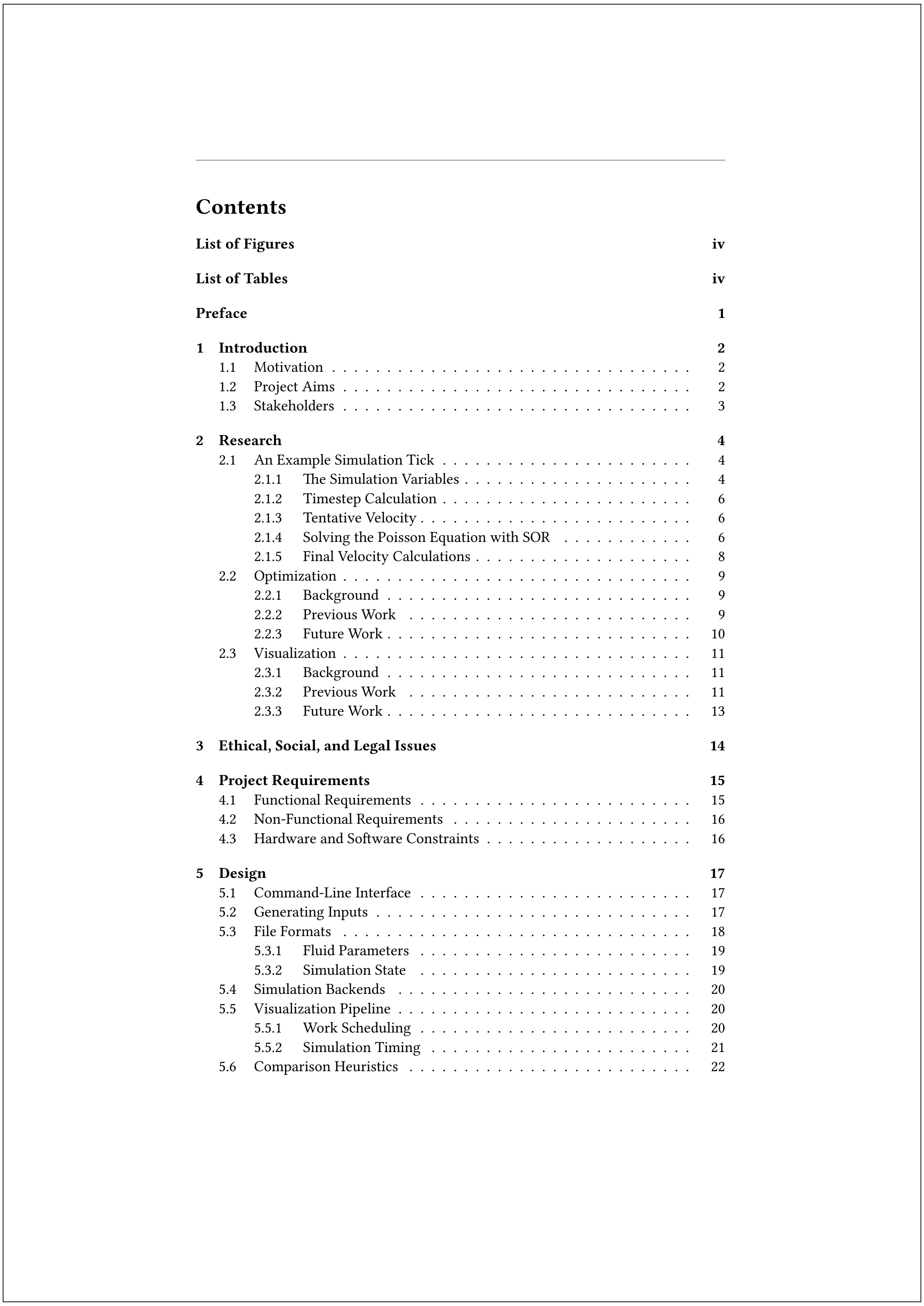
**Samuel Stark**

Supervisor: Dr. Matt Leeke

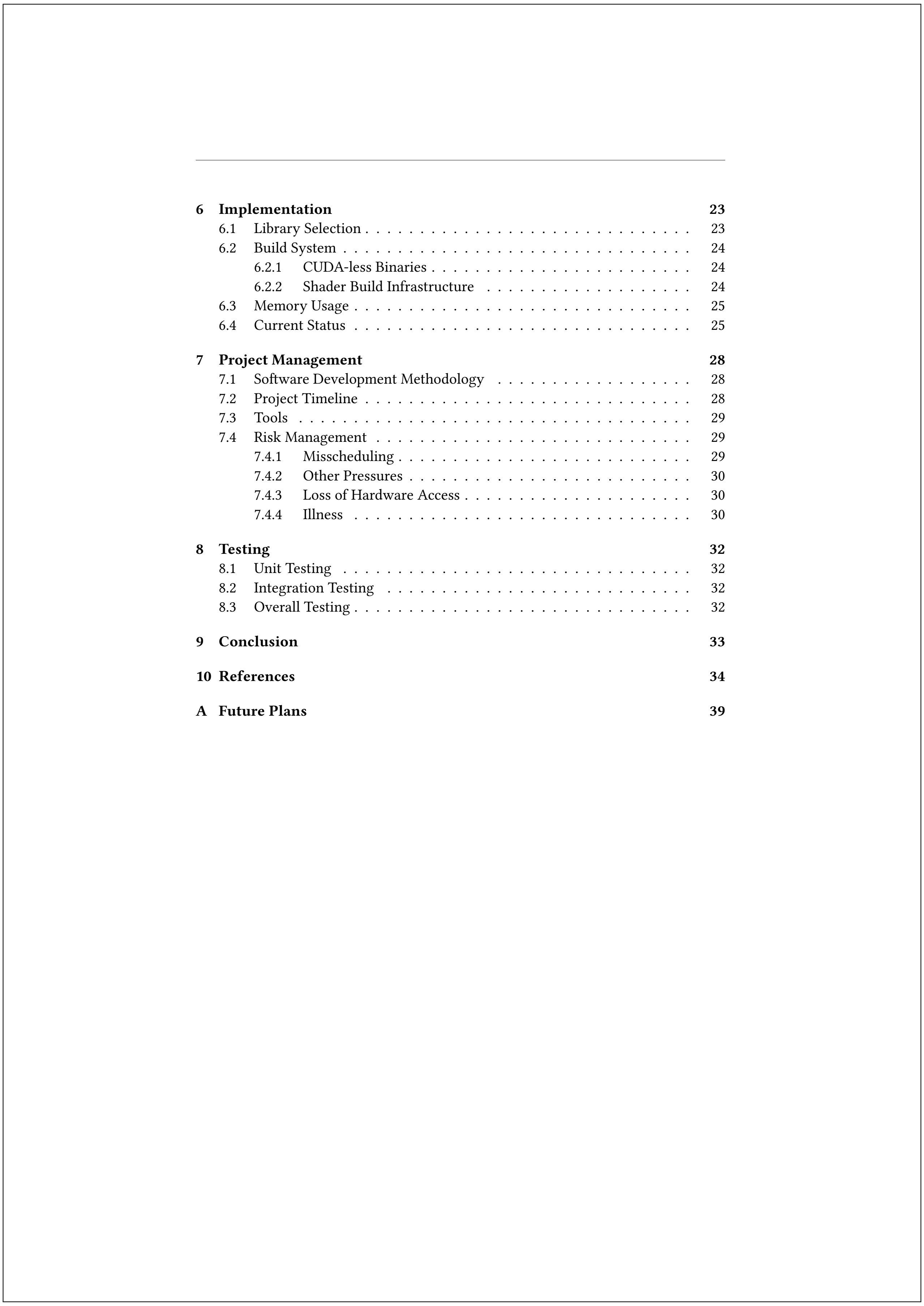
Department of Computer Science

University of Warwick

November 2020

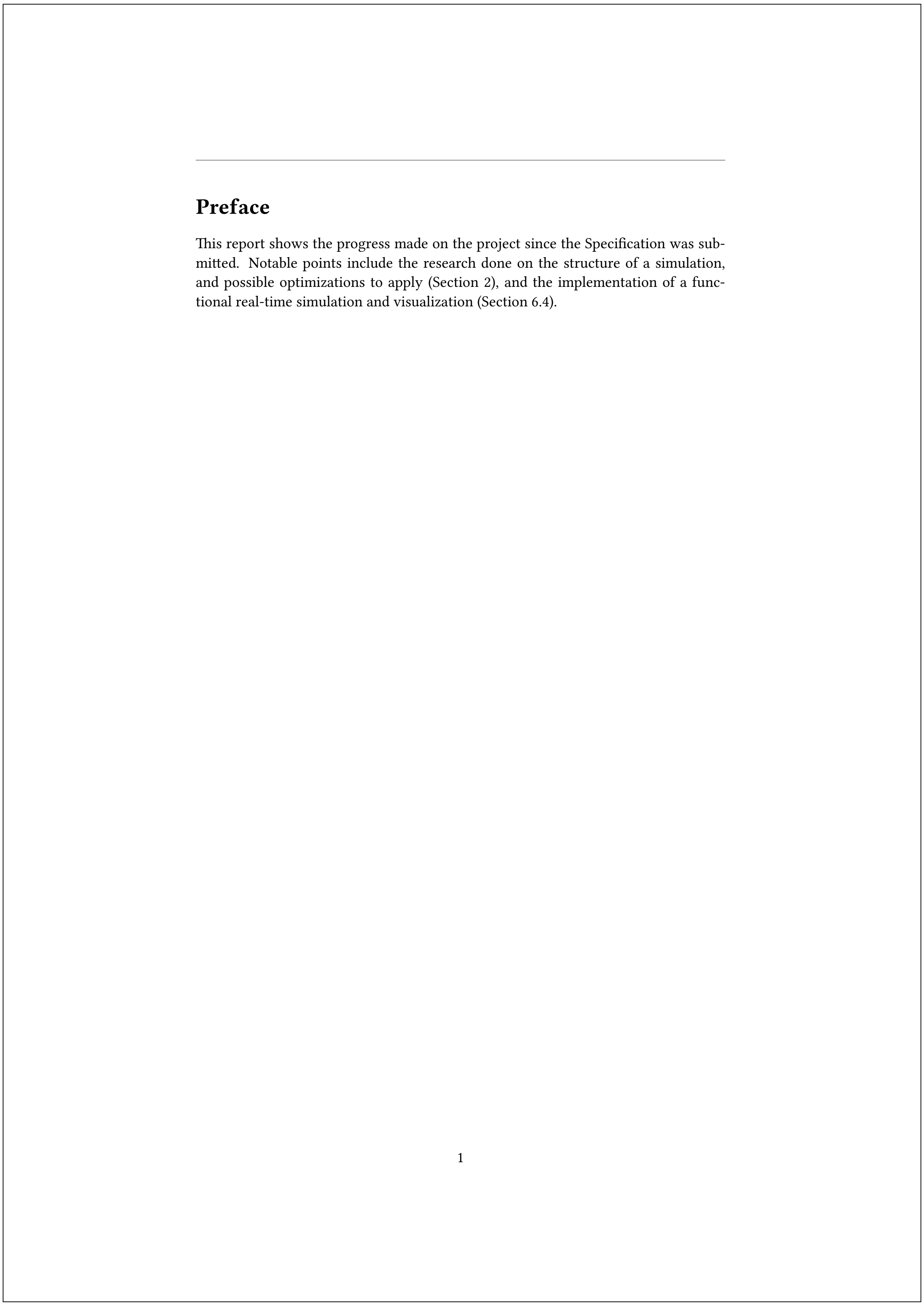


## Contents



## List of Figures

## List of Tables



## Preface

This report shows the progress made on the project since the Specification was submitted. Notable points include the research done on the structure of a simulation, and possible optimizations to apply (Section 2), and the implementation of a functional real-time simulation and visualization (Section 6.4).

1

# 1 Introduction

The development of equations and mathematical constructs that model natural phenomena has been a large research space for centuries. As digital computers have developed, programs have been built to use these equations and find the results much faster than previously possible[4]. Computational Fluid Dynamics (CFD) programs are programs that simulate fluid flow in some form, usually using the Navier-Stokes equations (reproduced in Eqs. (2.1) and (2.2)).

These fluid simulations have a variety of uses, including in aerodynamics[35],fire spread modelling[65], and in the entertainment industry (albeit with a focus on artistic input rather than physical accuracy[21]).

These cases generally do not require simulations at interactive speeds, except for those found in the games industry. While the games industry does use fluid simulation[45], many uses do not precisely integrate the Navier-Stokes equations but approximates them [63] using a Lagrangian method. An exception to this is [61], which uses a Jacobi solver for the Navier-Stokes equations. This is used to simulate character interaction with different substances floating on the water surface[61], not to simulate large blocks of water. By and large, interactive speeds and precise simulation for large fields are not pursued together.

## 1.1 Motivation

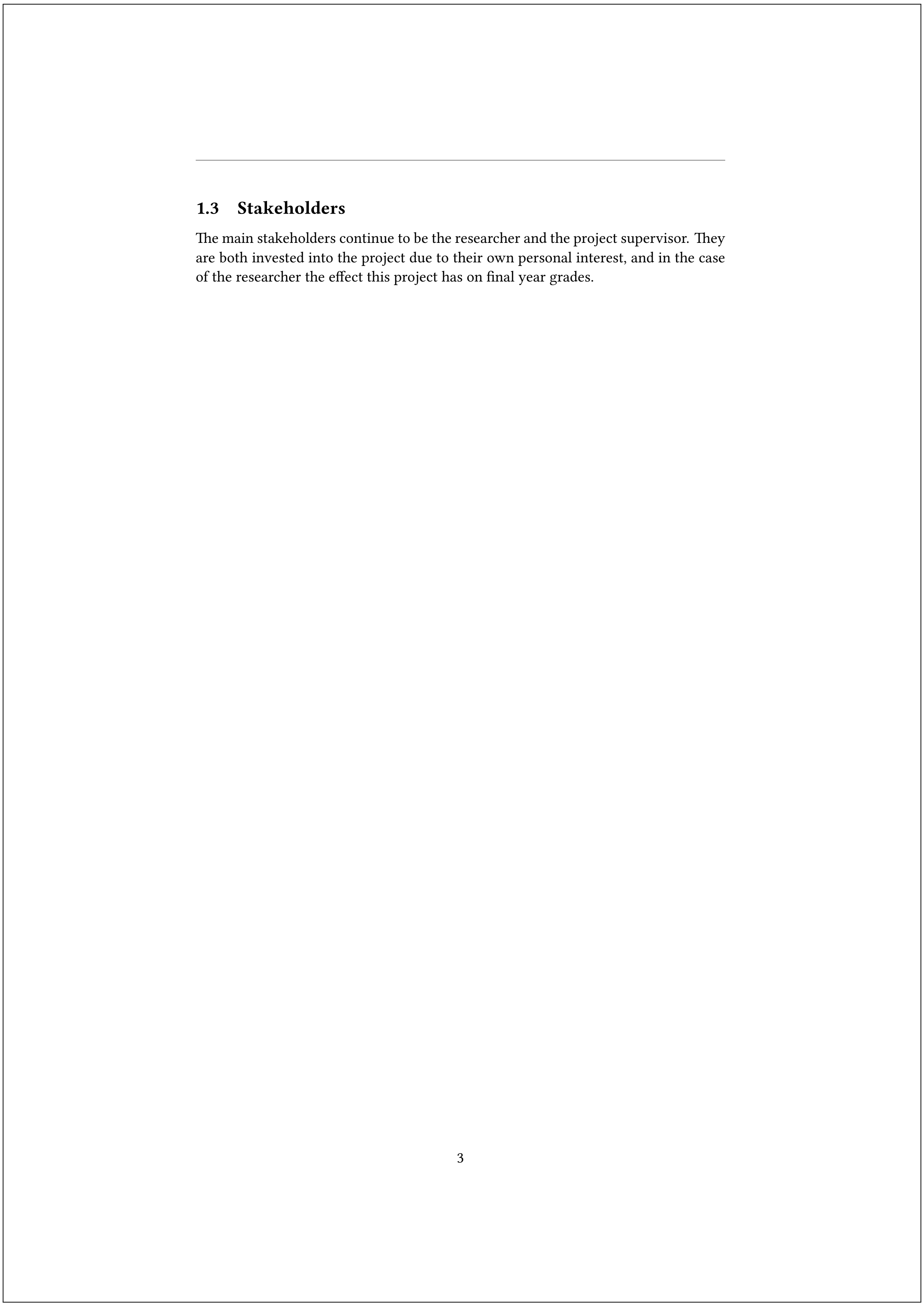
The Advanced Computer Architecture coursework last year presented a fluid simulation and tasked the students with optimizing it for a 6-core Intel i5-8500 CPU[9]. The original code ran very slowly, taking 80 seconds to simulate 10 seconds of time. After optimizations, the code simulated 10 seconds of time in just 1.26 seconds, 64x faster than the original and 7.9x faster than real time.[64]

However the simulation purposefully limited itself in some aspects, such as iteration count for an equation solver, which prevented it from converging to an accurate solution for the test data. Students were also explicitly prevented from accelerating the simulation using a GPU, which could have made it much faster as each simulation phase is embarrassingly parallel.

Another limitation was that the simulation state could only be visualized once the full simulation had completed, instead of in real time, even though the final simulation was fast enough. This made the results much more difficult to understand, especially for people who don't understand the underlying code or mathematics.

## 1.2 Project Aims

The first goal of this project has been to port the simulation to the GPU. This has provided a large speedup, with potential to improve it farther (see Section 2.2.3). The next goals of the project are to exploit this speedup in two ways: to make the simulation more detailed by increasing both the accuracy of the solver and the grid resolution; and to intuitively visualize the simulation in real time. The GPU simulation has be implemented in CUDA, and the visualization will be rendered in real time using Vulkan (see Section 6.1).



### 1.3 Stakeholders

The main stakeholders continue to be the researcher and the project supervisor. They are both invested into the project due to their own personal interest, and in the case of the researcher the effect this project has on final year grades.

# 2 Research

On top of the preliminary research performed for the Specification document, research of the underlying simulation structure and of the state of the art for optimizing a simulation has been done. Minor research has been also done for Visualization, although the schedule dictates this should start after Term 1 has ended.

## 2.1 An Example Simulation Tick

The 1998 book “Numerical simulation in fluid dynamics : a practical introduction”[29] defines a basic structure for a discrete simulated timestep (a.k.a. a “tick”) and provides a sample guide to implementing it in Fortran or C. To the best of the author’s knowledge this was used as the base of the ACA coursework, and continues to be the base of this project. This section will explain the general structure of the simulation as defined in [29].

The simulation described specifically simulates “*laminar* flows of *viscous, incompressible fluids*”[29] in 2D. *Laminar* flows can be treated as separate layers of particles that can slide past each other, which interact solely through friction forces. The opposite of this is Turbulent flow, where particles may move between layers due to small friction forces[29]. This adds extra viscosity (the turbulent eddy viscosity, as covered in more detail in [7]) which is much more difficult to accurately model.

*Incompressible* fluids have a uniform density across the entire flow, which greatly simplifies the calculations. This property can be assumed for low-velocity gases, and for most liquids[29].

*Viscous* fluids have high internal friction forces that will eventually bring a moving fluid to rest. The viscosity is controlled by a parameter known as the Reynolds number $Re$[19], which is constant over the fluid. As $Re \to 0$ the viscosity of the fluid approaches infinity, and as $Re \to \infty$ the fluid becomes *inviscid*, i.e. not viscous. Using high $Re$ this sim could be used to simulate inviscid fluids, although it is important for the fluid to still be laminar and incompressible.

Any forces acting throughout the bulk of the fluid i.e. gravity can be simulated using the $g = (g_x, g_y)$ vector. However the 2D variant of the simulation has been used in this project for top-down simulations with a level plane, so this is left unused.

### 2.1.1 The Simulation Variables

The simulation solves for three variables: horizontal velocity $u$, vertical velocity $v$, and pressure $p$. These variables are related by the Navier-Stokes momentum and continuity equations, which can be written as follows:

$$\begin{aligned} \frac{\partial u}{\partial t} + \frac{\partial p}{\partial x} &= \frac{1}{Re}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) - \frac{\partial (u^2)}{\partial x} - \frac{\partial (uv)}{\partial y} + g_x, \\ \frac{\partial v}{\partial t} + \frac{\partial p}{\partial y} &= \frac{1}{Re}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) - \frac{\partial (uv)}{\partial x} - \frac{\partial (v^2)}{\partial y} + g_y \end{aligned} \tag{2.1}$$

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \tag{2.2}$$

The values of the simulated quantities at tick $\#n$ are represented by $u^{(n)}, v^{(n)}, p^{(n)}$. These values are discretized by evaluating them at points on a staggered grid (see Fig. 2.1). This grid is indexed by $i$ in the x-direction and $j$ in the y-direction. It is important to note the variables $u, v$ represent the current velocity of the fluid within each grid space, **not** the velocity of the grid cells themselves. The grid does not move at any point during the simulation.

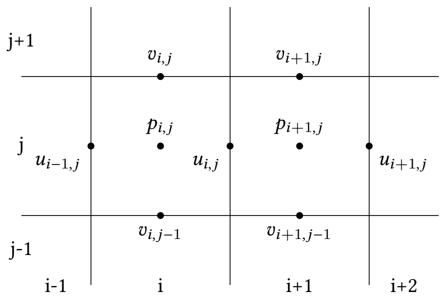


Figure 2.1: Discretization points for each variable on the staggered grid[29]

Each of the variables is located at a different position on the grid cell. Horizontal velocity $u_{i,j}$ is at the midpoint of the right cell edge, vertical velocity $v_{i,j}$ is at the midpoint of the top cell edge, and pressure $p_{i,j}$ is at the midpoint of the cell. This is used to solve odd-even decoupling[30]: for a fluid at rest (i.e. $u = v = 0$) the continuous solution is that the pressure $p$ is a constant across the grid. However were this to be discretized using central differences with all variables in the same locations, it would also be possible for a checkerboard of pressure values to form, and for oscillation to take place[29]. This is prevented by staggering the variables. [58] shows that this is also preventable through colocated grids, where a single grid is used for all variables and the velocities of each side of the cell are found using interpolation. These cell sides are implicitly staggered relative to the pressure and so avoid this problem.

To allow for derivatives to be accurately calculated for cells on the edges of the grid, boundary cells are added around each grid.The cells on the edges of any obstacles in the simulation are also marked as boundary squares. For a finite domain of size $(imax, jmax)$ this leads to a final grid size of $(imax + 2)$ by $(jmax + 2)$, where valid fluid values fall in the ranges $i \in \{1..imax\}$, $j \in \{1..jmax\}$.

The physical dimensions of each grid space are represented by $\delta x$, $\delta y$. This allows the derivatives of $u$ and $v$ to be calculated by finding the centered differences.

$$\left[\frac{\partial u}{\partial x}\right]_{i,j} := \frac{u_{i,j} - u_{i-1,j}}{\delta x}, \quad \left[\frac{\partial v}{\partial y}\right]_{i,j} := \frac{v_{i,j} - v_{i,j-1}}{\delta y} \tag{2.3}$$

The partial derivatives for pressure $\partial p/\partial x, \partial p/\partial y$ are found in the same way. The remaining derivatives, including second derivatives and $\partial uv/\partial x, \partial uv/\partial y$, can also be discretized by taking the difference across midpoints of their respective dimensions[50].

### 2.1.2 Timestep Calculation

Each simulation tick simulates a discrete amount of time known as a timestep $\delta t$. This timestep is not a fixed value, and typically one would want to select as large a timestep as possible. However, there are constraints on it's maximum value which depend on the simulation state.

As the derivatives are calculated between adjacent grid points, it is impossible to accurately simulate a timestep where fluid moves between non-adjacent grid cells . To prevent this the timestep $\delta t$ is calculated from the fluid velocities to make it impossible.

$$\delta t = \tau * \min\left(\frac{Re}{2}\left(\frac{1}{\delta x^2}+\frac{1}{\delta y^2}\right)^{-1}, \frac{\delta x}{|u_{max}|}, \frac{\delta y}{|v_{max}|}\right) \tag{2.4}$$

Because the new velocities calculated in this tick may be larger than $u_{max}$ and $v_{max}$, the safety factor $\tau \in [0,1]$ is used to ensure the timestep is large enough to account for it[68].

### 2.1.3 Tentative Velocity

The final values of $u$ and $v$ are defined as

$$\begin{aligned} u^{(n+1)} &= u^{(n)} + \delta t\left[\frac{1}{Re}\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right) - \frac{\partial (u^2)}{\partial x} - \frac{\partial (uv)}{\partial y} + g_x - \frac{\partial p}{\partial x}\right] \\ v^{(n+1)} &= v^{(n)} + \delta t\left[\frac{1}{Re}\left(\frac{\partial^2 v}{\partial x^2}+\frac{\partial^2 v}{\partial y^2}\right) - \frac{\partial (uv)}{\partial x} - \frac{\partial (v^2)}{\partial y} + g_y - \frac{\partial p}{\partial y}\right] \end{aligned} \tag{2.5}$$

However, as these depend on the partial derivatives of $p$, which itself depends on velocity, they cannot be solved analytically. In order to iteratively find $p$ the variables $f$ and $g$, for horizontal and vertical "tentative velocity", are introduced.

$$\begin{aligned} f^{(n)} &:= u^{(n)} + \delta t\left[\frac{1}{Re}\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right) - \frac{\partial (u^2)}{\partial x} - \frac{\partial (uv)}{\partial y} + g_x\right] \\ g^{(n)} &:= v^{(n)} + \delta t\left[\frac{1}{Re}\left(\frac{\partial^2 v}{\partial x^2}+\frac{\partial^2 v}{\partial y^2}\right) - \frac{\partial (uv)}{\partial x} - \frac{\partial (v^2)}{\partial y} + g_y\right] \end{aligned} \tag{2.6}$$

$$\begin{aligned} u^{(n+1)} &= f^{(n)} - \delta t\frac{\partial p^{(n+1)}}{\partial x} \\ v^{(n+1)} &= g^{(n)} - \delta t\frac{\partial p^{(n+1)}}{\partial y} \end{aligned} \tag{2.7}$$

### 2.1.4 Solving the Poisson Equation with SOR

For continuity to be achieved, the final velocity values must fulfil the continuity equation (Eq. (2.2)), the time discretization of which is shown below:

$$\frac{\partial u^{(n+1)}}{\partial x} + \frac{\partial v^{(n+1)}}{\partial y} = 0 \tag{2.8}$$

This means that the total amount of fluid entering a cell in tick $n+1$ is equal to the amount of fluid leaving, which must be the case otherwise the amount of fluid per cell wouldn't be constant and the fluid would be compressed.

Substituting the formulae in Eq. (2.7) into this relation and rearranging gives

$$\frac{\partial^2 p^{(n+1)}}{\partial x^2} + \frac{\partial^2 p^{(n+1)}}{\partial y^2} = \frac{1}{\delta t}\left(\frac{\partial f_{i,j}^{(n)}}{\partial x} + \frac{\partial g_{i,j}^{(n)}}{\partial y}\right) \tag{2.9}$$

The right hand side of this equation is constant for timestep $n$, so can be precalculated and assigned to its own variable *rhs*.

$$rhs_{i,j} := \frac{1}{\delta t}\left(\frac{\partial f_{i,j}^{(n)}}{\partial x} + \frac{\partial g_{i,j}^{(n)}}{\partial y}\right) \tag{2.10}$$

$$\frac{\partial^2 p^{(n+1)}}{\partial x^2} + \frac{\partial^2 p^{(n+1)}}{\partial y^2} = rhs_{i,j} \tag{2.11}$$

Discretizing this gives

$$\frac{p_{i+1,j}^{(n+1)} - 2p_{i,j}^{(n+1)} + p_{i-1,j}^{(n+1)}}{(\delta x)^2} + \frac{p_{i,j+1}^{(n+1)} - 2p_{i,j}^{(n+1)} + p_{i,j-1}^{(n+1)}}{(\delta y)^2} = rhs_{i,j} \tag{2.12}$$

and taking the simplest boundary conditions[29]

$$p_{0,j} = p_{1,j}, \quad p_{i_{max}+1,j} = p_{i_{max},j} \quad j \in \{1..j_{max}\} \tag{2.13}$$

$$p_{i,0} = p_{i,1}, \quad p_{i,j_{max}+1} = p_{i,j_{max}} \quad i \in \{1..i_{max}\} \tag{2.14}$$

$$f_{0,j} = u_{0,j}, \quad f_{i_{max},j} = u_{i_{max},j} \quad j \in \{1..j_{max}\} \tag{2.15}$$

$$g_{i,0} = v_{i,0}, \quad g_{i,j_{max}} = v_{i,j_{max}} \quad i \in \{1..i_{max}\} \tag{2.16}$$

resolves the equation to:

$$\frac{\epsilon_{i,j}^{E}(p_{i+1,j}^{(n+1)} - p_{i,j}^{(n+1)}) - \epsilon_{i,j}^{W}(p_{i,j}^{(n+1)} - p_{i-1,j}^{(n+1)})}{(\delta x)^2} + \frac{\epsilon_{i,j}^{N}(p_{i,j+1}^{(n+1)} - p_{i,j}^{(n+1)}) - \epsilon_{i,j}^{S}(p_{i,j}^{(n+1)} - p_{i,j-1}^{(n+1)})}{(\delta y)^2} = rhs_{i,j} \tag{2.17}$$

where $\epsilon_{i,j}^{\{N,S,E,W\}}$ represents the boundary squares (shown here for North, but it extends to the other directions)

$$\epsilon_{i,j}^{N} = \begin{cases} 0 & \text{The square directly above } i,j \text{ is a boundary} \\ 1 & \text{The square directly above } i,j \text{ is \textit{not} a boundary} \end{cases} \tag{2.18}$$

Over the whole grid, this results in a linear system of equations over the inputs $p_{i,j} \forall i \in \{1..i_{max}\}, j \in \{1..j_{max}\}$. These can be decoupled by partitioning $p$ into red and black squares by a checkerboard pattern (see Fig. 2.2). As each individual cell only depends on the adjacent values, iterations of Successive Over-Relaxation (SOR)

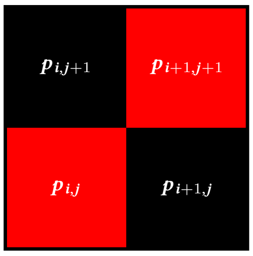


Figure 2.2: Example checkerboard pattern used for red/black splitting.

can be performed on red and black in turn to reach a final value[1][73]:

$$\beta_{i,j} := \frac{\omega}{\left(\frac{\epsilon^E_{i,j}+\epsilon^W_{i,j}}{(\delta x)^2} + \frac{\epsilon^N_{i,j}+\epsilon^S_{i,j}}{(\delta y)^2}\right)} \tag{2.19}$$

$$p^{it+1}_{i,j} := (1-\omega)p^{it}_{i,j} + \beta_{i,j} * \left(\frac{\epsilon^E_{i,j}p^{it}_{i+1,j} + \epsilon^W_{i,j}p^{it}_{i-1,j}}{(\delta x)^2} + \frac{\epsilon^N_{i,j}p^{it}_{i,j+1} + \epsilon^W_{i,j}p^{it}_{i,j-1}}{(\delta y)^2} - rhs_{i,j}\right) \tag{2.20}$$

These iterations are continued until the L2 norm[42] of the residuals (the difference between the left-hand side as calculated and the expected right-hand side of Eq. (2.20) for each cell) falls below a specific tolerance[2][29].

### 2.1.5 Final Velocity Calculations

Once the final values of $p$ have been calculated the velocity values $u, v$ can be found with Eq. (2.7). The boundary conditions for velocity must then be applied. There are four relevant types of boundary condition[3], which are applied depending on the type of boundary.

1. No-Slip condition - no fluid penetrates the boundary, and fluid does not move past it i.e. the boundary applies friction.
2. Free-Slip condition - fluid may not penetrate the boundary, but no friction is applied. Only tangential velocity is preserved for adjacent fluids.
3. Inflow - fluid is flowing in constantly, so the velocity is set to a constant value.
4. Outflow - velocity perpendicular to the surface is preserved and fluids may flow out.

[1]This could equally be done without partitioning $p$, but the partitioning splits the SOR into separate phases which can then be parallelized. Normal SOR cannot be parallelized[2].

[2]In the ACA coursework this tolerance was relative to the L2 norm of $p$, although this was not directly specified by the book.

[3]The book specifies five, including a Periodic Boundary Condition, which the ACA system does not support.

## 2.2 Optimization

Optimizing simulations is important in all cases, even those that are not real-time, as it allows the engineers using the software to iterate faster on their designs. When the extra constraint of real-time speeds is added, it becomes even more important. This research is mostly complete, although more can be done if the simulation needs to get even faster.

### 2.2.1 Background

One of the first papers on optimizing a CFD simulation was released in 1995[40]. This paper considered the effect of automatic compiler parallelization and optimization of a full CFD program, and the steps a programmer must take to guide the compiler i.e. avoiding false sharing. The program was only executed on the CPU, as General Purpose GPU computing (GPGPU) had not yet taken hold.

GPGPU was first used for CFD simulations in 2004 with this paper[20]. This used the "fragment shading" stage of the GPU rendering pipeline to perform the computation, as standalone "compute" pipelines were only exposed by APIs from 2007 onwards. Such APIs include CUDA (2007)[56], OpenCL (2008)[67], DirectX's DirectCompute (2009)[49], and OpenGL 4's compute shaders (2012)[60].

Since 2007, using GPGPU for CFD has become a large topic of study, as investigated in detail by [51]. While the concept of accelerating a fluid simulation on the GPU is not new, much of the novelty of our optimizations will stem from the interaction between the simulation and the other systems at work. As an example, if the simulation were to be modifiable with user input, introducing this new data and updating the boundary conditions in an efficient manner becomes a new problem.

### 2.2.2 Previous Work

As work on CFD progressed some optimizations were developed that change the simulation pipeline and provide an overall speedup. Some of these were adapted into my ACA coursework submission[64], which this project is based on, and carry over into the CUDA version.

Given the definition of $\beta$ in Eq. (2.19), the value of $\beta_{i,j}$ does not change over the course of the simulation and so can be precalculated before the simulation starts. Additionally if it can be guaranteed that for every boundary square $p = 0$, which can be done either by never updating their pressure values or by updating them with $\beta_{i,j} = 0$, then $\epsilon_{i,j}$ doesn't need to be evaluated during the simulation at all. These optimizations increased the runtime speed of the Poisson evaluation by 2.24x[4], and they have been kept in the CUDA program.

The book states an alternate solution where $\epsilon$ is set to 1 at all times and pressure values on boundaries are copied from adjacent fluid squares[29]. This apparently stops noncontinuous starting velocities from producing nonphysical pressure values.

As stated in Section 2.1.4, red/black SOR is used to iteratively solve the Poisson equation. In the initial ACA coursework the values ($f$, $g$, $p$, $rhs$) for red and black data were stored in the same arrays. This was problematic as data of the same color was never contiguous, and any iteration looking for just red values would get a cache

[4]The $\beta$ precalculation increased speed by 1.4x, and the removal of $\epsilon$ increased speed by 1.6x.[64]

line with both colors, leading to half of each cache line being wasted. To fix this, red and black data is split into separate arrays before starting the Poisson solver. This has been carried over into the CUDA implementation.

The ACA solution used OpenMP[57] to automatically parallelize the Poisson solver (and other program elements) by column. That is, each thread was given a group of columns to process. This was not needed in the CUDA version as each GPU kernel is implicitly parallelized over many GPU threads.

The ACA solution included optimizations exploiting properties of the original code, such as floating point precision, to speed up calculations while producing identical results. These optimizations include using fused multiply-add[47] in some places (but not all), precalculating divisions with double-precision floats, and skipping the residual calculation phase altogether. As this project is focused on improving upon the accuracy of the ACA submission, instead of producing bit-identical results, these optimizations have generally not been implemented into the CUDA program.[5]

### 2.2.3 Future Work

Along with the items mentioned in the previous section, there are some optimizations planned to be implemented over the Christmas break and during Term 2.

CUDA devices are split into many threads, which are split into groups of 32 that are executed concurrently as a warp[56]. If the threads in a warp attempt to access multiple words in the same cache line, the access is *coalesced*[55] and only one cache line needs to be fetched for the warp to continue. Otherwise if the accesses all touch different cache lines, every cache line needs to be fetched before execution can continue for any of the threads. The CUDA program attempts to arrange the threads such that they coalesce accesses, but it has not been verified to work yet.

The CUDA C Programming Guide[54] states that read-only memory can be read into a special data cache using the `__ldg()` intrinsic. The compiler may insert this automatically when it detects that data must be read-only. The use of `const` and `__restrict__` qualifiers on pointers that are read-only is encouraged to make read-only data obvious. In [17] it was found that introducing these qualifiers where possible led to large speedups in pointer heavy applications, and while our case may not use many pointers this should still be implemented wherever possible. In the CUDA implementation templates for input and output matrices are used that include these qualifiers automatically, and all kernels are assumed to restrict all pointer arguments. However it has yet to be verified that `__ldg()` is inserted in the correct places, which should be done in the future.

The ACA solution used Intel AVX and SSE instructions[34] to calculate four Poisson values at once[6]. Each CUDA core of a GPU has access to four-element vectors without any extensions, so this vectorization can be extended per CUDA core. This has not been implemented in the CUDA program, but is a future extension to test. This may or may not actually speed up computation, as the memory bandwidth would be quadrupled and the computation is already excessively parallel.

Calculating the simulation timestep and calculating the residual for a Poisson iteration both require a reduction over large blocks of data. Highly parallel GPU optimizations have already been studied extensively, so it should be trivial to implement

[5]The CUDA program still lacks a residual phase, but this is planned to be implemented later.
[6]Vectors of eight were tried but were found to be slower than four.

a very fast generic reduction kernel. In [31] seven kernels are described, in ascending order of speed. Currently the CUDA program uses the second kernel model, and this is planned to be moved up to the seventh kernel in the future.

## 2.3 Visualization

For the engineers and scientists developing simulations, it is important for a visualization to be completely accurate and show the data in as much detail as possible. However there are other groups that may not have as deep of an understanding, but whose actions and decisions should still be informed by the simulation results. Currently the research is focused on learning lessons from the ACA coursework's provided visualization. For a visualization to cater to these groups well further research in this space is required.

### 2.3.1 Background

One of the earliest CFD interactive visualizations was in 2002, which had a simulation running slower than real time on a separate computer to the real-time visualization[39]. Decoupling the simulation speed from the visualization speed allowed for high framerates to be achieved for the user interface, however any changes made from the user interface had a delay of 0.5 seconds before being reflected in the simulation.

Many scientific visualizations of fluid flow exist already. To name two examples, streak lines and fluid colors are used for visualizing fluid flow[6]. These methods are perfectly fine for those who understand what these elements mean, i.e. what the colors represent, and what the optimal airflow would look like. However, for those unfamiliar with the simulation these methods can be difficult to understand.

This project aims to develop new visualization techniques for two-dimensional simulations that are more intuitive than the current offerings, that can be extended to three dimensions easily. Using high-speed rendering APIs like Vulkan[72] will allow these visualizations to be made even more complex while maintaining high speeds. Furthermore, our approach may allow for slight inaccuracies to be introduced for the sake of intuitivity, which has not been explored in research to the author's knowledge.

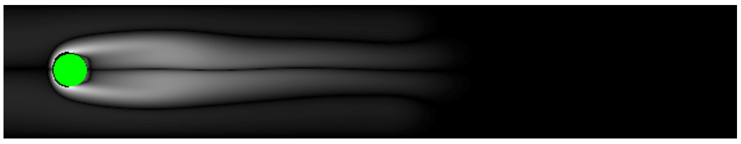
### 2.3.2 Previous Work

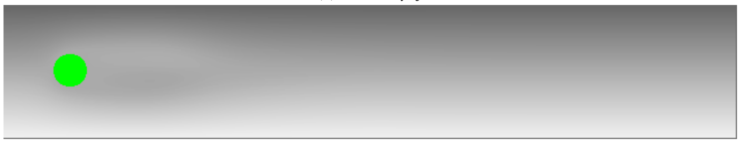
The original coursework[9] provided a simple image visualizer for a simulation state, which evaluated one of two quantities over the grid and produced a `.ppm` image with the result. These quantities were Vorticity ($\zeta$), the strength of vortical (a.k.a. rotational) motion at each point in the grid; and Stream Function ($\psi$), the contours of which define streamlines. Streamlines are lines that are parallel to the velocity vector at each point, allowing the long-term flow of particles to be represented with a single line, and thus in a static image.[48] The quantities are defined by Eqs. (2.21) and (2.22), as specified in [29]. Examples of these modes are shown in Fig. 2.3.

$$\zeta(x, y) := \frac{\delta u}{\delta y} - \frac{\delta v}{\delta x} \tag{2.21}$$

$$\frac{\delta\psi(x, y)}{\delta x} := -v, \quad \frac{\delta\psi(x, y)}{\delta y} := u \tag{2.22}$$

11


(a) Vorticity $\zeta$


(b) Stream Function $\psi$

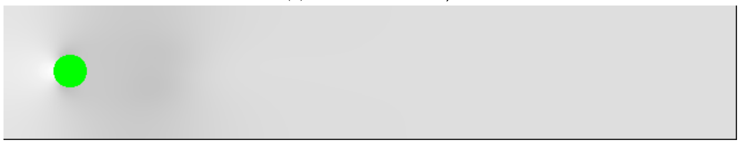

(c) Pressure $p$

Figure 2.3: Examples of the three outputs available from the ACA visualizer, all visualizing the same state.

The vorticity image in Fig. 2.3a competently shows which areas of the grid contain particle movement. However near the edges of the obstacle circle (shown in green) the edges are black, implying no movement or rotation, which is incorrect and also a distracting artifact for the viewer. These are due to the imprecise nature of the original code, which only uses the differences to the East and South to find $\zeta$. This breaks down when the squares in these directions are boundaries, and the program defaults to zero. A better solution would be to take the central difference whenever possible, and to fall back to using only one side when adjacent to an boundaries. This would mean the only points where this breaks down are where a square is surrounded by boundaries on opposite sides, which is much less likely and would also likely break other areas of the simulation.

The Stream Function visualization (Fig. 2.3b) is nearly impossible to visually parse, which makes sense as the velocity information is encoded in the differences between adjacent squares and not directly in the colors. The Stream Function is not intended to be directly visualized, but instead used to find streamlines which can be visualized directly.

During program development a third mode was added which directly visualized the pressure values to aid in debugging, but this was not a very useful visualization as seen in Fig. 2.3c. Pressure is only ever referenced in the Navier-Stokes equation (and subsequently the algorithm) as a relative value. However, the simulation in practice ends up increasing all cells by a small amount each iteration. This overall increase in pressure values is ignored by the simulation, but the visualization doesn't adjust for it. In this example, the pressure values have all increased so even the lowest pressure value is a mid-gray. If the program simulated for too long, the pressure values would become too high and the visualization would be entirely white. This pressure mode has been carried over to the CUDA program as a placeholder visualization, but will be replaced.

### 2.3.3 Future Work

While a purely image-based approach to visualizing properties can be useful, other approaches allow for i.e. multidimensional quantities such as velocity to be expressed much more easily. In the case of velocity, vector fields and particle tracing are both shown in [29] to be effective.

Given that our simulation is realtime, we can also add changes over time to the mix. Tracing particle paths and rendering them as a line could be replaced by actually watching the particles move over time. Particle movement could also be enhanced with extra behaviour similar to that of BOIDs[59], which among other things implement Collision Avoidance. This would prevent particles from overlapping and getting visually lost. Vorticity/rotational movement could be visualized by adding particles to the grid that rotate over time based on he vorticity at their location. This could allow the vorticity to be represented in the same view as the other parts of the simulation, instead of creating a dedicated view separate from velocity/pressure.

## 3 Ethical, Social, and Legal Issues

As stated in the Specification, there are (and continue to be) no ethical or social issues with the development of this simulation and visualization. The simulation has been derived from code provided to the students for the ACA coursework[9], which itself is directly derived from a book[29] available at the Warwick Library[43].

During the development of the visualization, feedback may be gathered as to which elements are most intuitive. Any such feedback will be restricted to the opinion of friends and family, and as such comes under the category of "Student projects with primarily an educational purpose"[70], so does not require ethical review. This feedback would be gathered according to the University guidelines[18], and any gathering will follow the Data Protection Act 2018[16].

To ensure the work can be trusted, and to maintain professional standards, the BCS Code of Conduct[14] has been followed. Professional standards will continue to be maintained during development, and research performed will be effectively referenced to the same high standard achieved so far.

## 4 Project Requirements

These basic functional and non-functional requirements define the baseline the final result will be measured against. These are mostly unchanged from the Specification document, and may be expanded upon in the final report as the project evolves and more testable features are added. Requirements F5.4 to F5.6 have been added as the Specification was unclear as to the speed of the visualization. Functional requirements for visualization are intentionally not included, as an intuitive visualization can take many forms that must be investigated further before being specified.

### 4.1 Functional Requirements

**F1** The system **must** store simulation state in a file or set of files.

**F2** The system **must** be able to load the initial state of a simulation from these file(s).

**F3** The system **must** be able to generate initial simulation state files.

**F4** The system **must** be able to simulate from an initial state for a set amount of time without visualizing.

- **F4.1** This mode **must** be able to store the final state to output file(s).

**F5** The system **must** be able to simulate from an initial state for an indeterminate amount of time while visualizing.

- **F5.1** This mode **must** allow the user to pause and resume the simulation.
- **F5.2** This mode **should** be able to save it's state to output file(s) when requested.
- **F5.3** This mode **should** allow the user to manipulate the simulation state while simulating.
- **F5.4** This mode **should** be able to run at a locked frame-rate.
- **F5.5** This mode **should** be able to run as fast as possible, without locking the framerate.
- **F5.6** This mode **must** be able to perform at least one of Requirements F5.4 and F5.5.

**F6** Both methods of simulation **must** be capable of using the GPU for simulating.

**F7** The system **must** be able to compare how similar two simulation states are.

- **F7.1** This comparison **should** produce a binary SIMILAR/NOT SIMILAR verdict using heuristics.

## 4.2 Non-Functional Requirements

**NF1** The simulation **must** produce similar results to the original coursework when equivalent initial state is used.

**NF2** The visualized simulation **must** run in real-time at framerates $\geq$ 30 FPS for some outputs.

**NF3** The visualized simulation **should** intuitively represent the fluid flow such that it can be understood by someone unfamiliar with fluid simulation.

**NF4** The system **must** be fully documented and maintainable.

**NF5** The system **should** have a simple guide to common operations for new users to refer to.

**NF6** The system **must** be capable of operating on large datasets (e.g. 4096x4096 grids) without failing.

**NF7** The system **should** be fully compilable and executable from a DCS machine with minimal extra installations.

## 4.3 Hardware and Software Constraints

As this simulation uses a GPU, the developer must have one available for debugging and testing the program. As the CUDA API is used to implement the simulation (see Section 6.1), the program requires an NVIDIA GPU to run.

The high-speed rendering requirements of the program necessitated the use of Vulkan over OpenGL. Vulkan gives the developer more fine control over scheduling, and allows the hardware to take shortcuts that it may not be able to do under OpenGL. For more on this decision see Section 6.1.

## 5 Design

### 5.1 Command-Line Interface

The compiled binary uses a command-line interface to configure and run one of many subcommands available. These subcommands are:

- `makeinput`, which generates simulation input files, fulfilling Requirement F3.
- `fixedtime`, which runs a headless simulation for a fixed time, fulfilling Requirement F4.
- `compare`, which compares two simulation states for equality, fulfilling Requirement F7.
- `renderppm`, which visualizes a simulation state in the same way the original ACA coursework did.
- `convert2newbinary`, for converting ACA simulation state files to a potential new format (see Section 5.3). Currently a no-op.
- `run`, which starts a real-time visualized simulation, fulfilling Requirement F5.

Splitting the program into subcommands was inspired by Git[23], and avoids creating separate binaries for each operation. Each subcommand can be configured with command-line options conforming to POSIX standard[33]. Examples of using the program are in Fig. 5.1.

```
# Create an input file based on simple_layout with a size of 1x2 metres
./sim_cuda makeinput ./simple_layout.png 1 2 ./initial.bin

# Run it in headless mode for 10 seconds
./sim_cuda fixedtime --backend=cuda ./fluid.json ./initial.bin 10 \
                        -o ./output_after_10.bin

# Compare it to the expected output
./sim_cuda compare ./output_after_10.bin ./expected_after_10.bin

# Render it out to an image
./sim_cuda renderppm ./output_after_10.bin zeta ./output_after_10.ppm

# Try visualizing it in real-time
./sim_cuda run --backend=cuda ./fluid.json ./initial.bin
```

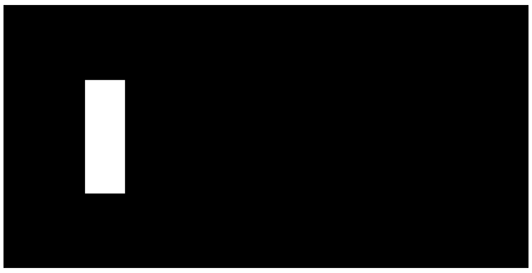
Figure 5.1: Example usage of the simulation program

### 5.2 Generating Inputs

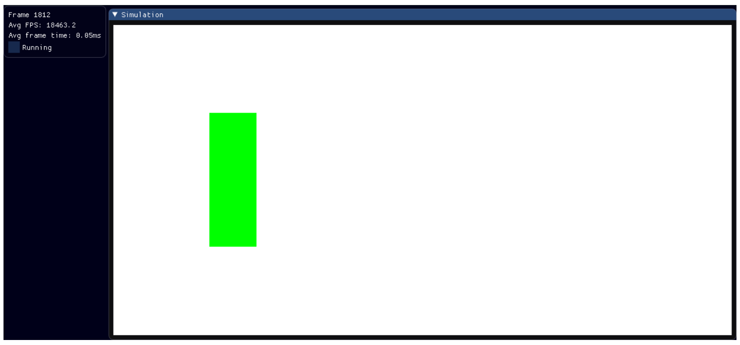
The `makeinput` subcommand allows input simulation states to be generated from image files. Each pixel of the input image represents a cell of the grid, including padding cells[7], where non-black pixels are denoted as boundary cells and are fluid cells

[7]This can allow the padding cells to be fluids rather than boundaries, which is incorrect. In the future this will be changed to add padding cells once the image is parsed.

otherwise. The example in Fig. 5.2 shows an example file which creates a rectangular obstacle, and the visualization of the generated state.



(a) Base Image



```
Frame 1812
Avg FPS: 18463.2
Avg frame time: 0.05ms
```

(b) Simulation

Figure 5.2: An example of converting an image to a simulation state.

Velocities and pressure in every cell are currently set to constant default values. For velocities, this is $1\,\mathrm{m/s}$ east, equal to the default flow of incoming fluid, which may cause issues with correctness. An example would be a situation where fluid is occluded from the input direction by an obstacle, but moves east anyway with no reason to do so. This will likely be changed to zero out initial velocity, requiring some simulation to take place before the fluid begins to move.

The exact initial value of pressure is inconsequential as the simulation only cares about the difference between cells. This means the only significant point is that the pressure is equal over all cells, so the system should be in equilibrium. This may be inconsistent with the nonzero velocities mentioned above, which is another reason to zero them out instead.

## 5.3 File Formats

To fulfil Requirement F1 two file formats have been defined to store simulation data and parameters.

### 5.3.1 Fluid Parameters

Parameters that are characteristic of a particular fluid or simulation type are stored in a "Fluid Parameters" file. This includes the Reynolds number, the timestep safety factor, and the maximum iteration count for the Poisson solver. They are stored in a JSON format to be human-readable, are reusable for different simulation states, and can be easily edited by the end user. An example is shown in Fig. 5.3.

```
{
    "Re": 150.0,
    "initial_velocity_x": 1.0,
    "initial_velocity_y": 0.0,
    "timestep_divisor": 60,
    "max_timestep_divisor": 480,
    "timestep_safety": 0.5,
    "gamma": 0.9,
    "poisson_max_iterations": 100,
    "poisson_error_threshold": 0.001,
    "poisson_omega": 1.7
}
```

Figure 5.3: An example Fluid Parameters file.

### 5.3.2 Simulation State

Data unique to an individual state such as simulation resolution, physical size, and velocity fields are stored in a binary format reused from the ACA project. As the data is much more sensitive to individual modifications[8], it makes more sense to store this data in a binary format where it cannot be easily modified by a user. Additionally the binary format is much smaller than any text-based format, which helps as the volume of data stored is much larger than that stored in the fluid parameters.

There is no magic string at the start of the file, which may be introduced in a new version. The header consists of a pair of unsigned 32-bit integers specifying the resolution of the simulation, and a pair of 32-bit floating point numbers specifying the physical dimensions of the simulation. From there, four sets of data for each column are stored, including the boundary padding squares:

1. Horizontal Velocity $u$ (`float32`)
2. Vertical Velocity $v$ (`float32`)
3. Pressure $p$ (`float32`)
4. Cell Flags, defining which adjacent squares are boundaries (`uint8`)

This structure is somewhat unintuitive and error-prone, an example being the Cell Flags which may end up being inconsistent between adjacent cells, but for the sake of compatibility with ACA data it has been kept. In the future it may be updated to a safer format.

[8]i.e. changing a single value in the velocity field can introduce discontinuities

## 5.4 Simulation Backends

To allow easy comparisons between CPU and GPU simulations the program contains multiple simulation backends which can be requested when running a headless simulation[9]. The headless simulation uses a `--backend` command line option to allow the user to choose the backend from this selection:

- Null, a backend which does no simulation for testing purposes.
- CPU Simple, equivalent to pre-optimization ACA code.
- CPU Optimized, equivalent to the submission for ACA, bit-equivalent to CPU Simple.
- CPU Optimized Adapted, a version of CPU Optimized slightly modified to be closer to the GPU version.
- CUDA Backend V1, the only GPU-based backend.

Currently the only modification present in the CPU Optimized Adapted backend is the removal of double-precision floating point logic, which is not present on the GPU for speed concerns. However once the GPU introduces residual checking for the iterative solver, or any other major changes to the pipeline, they will be introduced into this backend to ensure a like-for-like comparison.

## 5.5 Visualization Pipeline

This section details the extra code implemented in order to efficiently and effectively visualize simulation results in real time. This is not related to the `renderppm` subcommand, which uses CPU code to render a single simulation state as an image.

### 5.5.1 Work Scheduling

The most important tasks are run on the GPU, and are the limiting factor for performance. This means it's important that the GPU is running at all times. Points where the GPU is doing no useful work are known as "bubbles".

To avoid these bubbles a GUI thread is spawned to prepare the current frame's draw commands and handle user input. This runs in parallel with the simulation thread, which dispatches CUDA kernels for the simulation tick(s). Once the sim is finished, it waits on the GUI thread (which should always be done by this point) and dispatches the draw commands through Vulkan. It then dispatches the GUI thread again, waits for the render to finish and then immediately starts the next simulation.

Figure 5.4 shows the scheduling in more detail. The GUI work can execute anywhere within the dashed lines and still not delay the final draw. Note that the GPU is always doing useful work, and there are no bubbles.

Currently it is assumed that the rendering of one frame and the simulation of the next frame cannot happen in parallel. This is also enforced by the fact that velocity and pressure buffers are used by both the simulation and the render, meaning that a simulation could not update those buffers without invoking a race condition. However if the GPU has spare cores available while rendering a frame, then the simulation

[9]The realtime visualization currently only supports the CUDA-based backend, violating Requirement F6.

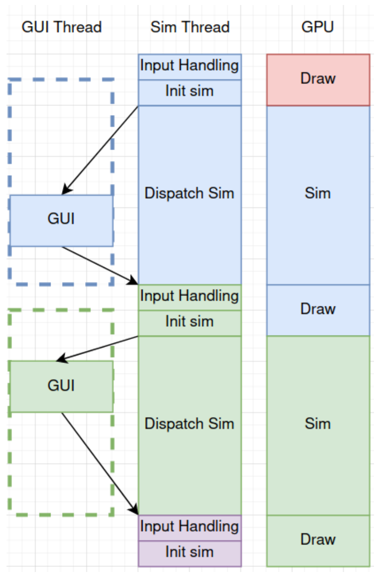


Figure 5.4: Thread utilization diagram.

could use these cores on the next simulation at the same time. Furthermore, most of the simulation can take place without writing to the velocity and pressure buffers, so those parts of the simulation could be run in parallel with the rendering without any worries. Or, of course, the simulation could be double-buffered and run entirely in parallel with the rendering.

### 5.5.2 Simulation Timing

As specified by Requirement F5.6 there are two acceptable modes that the visualization can run in: Flat Out (Requirement F5.5), where the simulation runs as fast as possible; and Locked Framerate (Requirement F5.4), where a frame-rate is selected and the visualization only produces that many frames per second. The definition of a flat-out speed is that if Frame $N$ takes some amount of real-world time $t_N$, then the next frame should simulate $t_N$ seconds of simulation-time before it is presented. This way the simulation runs as fast as possible, but it could lead to situations where if the simulation takes too long, the next simulation will have to simulate even more time and take longer etc.

With a locked frame-rate the simulation selects a timestep $\delta t$ to simulate for each frame, and limits the speed at which frames are produced. If the frame-rate is 60 frames per second, then the visualization would potentially have to delay itself so that each frame takes $1/60 = 16.67\,\text{ms}$.

Currently the visualization does not take either of these approaches, but simulates a fixed $\delta t = 1/60$ without limiting the frame-rate. This means on the researcher's current setup, which has a monitor capable of showing 120fps, the visualization runs 120 ticks per second which results in a simulation that's 2x faster than real-time. This shows the simulation is fast, which is promising, but it fails the requirements.

## 5.6 Comparison Heuristics

In the comparison subcommand heuristics are used to judge if one simulation is accurate and precise with respect to the other. This does not quite fulfil Requirement F7.1, as there are two results and two heuristics used instead of just one. To fix this, it is planned that the comparison will produce `SIMILAR` if the simulations are both accurate and precise, and `NOT SIMILAR` otherwise. The program may then provide additional information to help the user determine the cause of the problem.

This assumes one of the supplied states is a known-valid simulation state, and the other is not. Two simulation state files are provided, and the velocity and pressure values $u, v, p$ are compared separately. The simulation states must be of the same resolution, and should use the same boundary squares (although this is not currently checked).

The comparison is performed by calculating the mean and standard deviation of the square error between the datasets. These are then compared to tolerance values to produce two binary outputs: ACCURATE if the mean is below tolerance, and PRECISE if the standard deviation is below tolerance. Examples are shown in Fig. 5.5.

The tolerance for the mean was derived from an expected error magnitude of $\pm 10^{-7}$, which was squared to produce $10^{-14}$. It is assumed that the standard deviation should always be smaller than the mean, so the tolerance for standard deviation is also $10^{-14}$.

```
Velocity X:
    Sq. Error Mean:     0
      ACCURATE
    Sq. Error Std. Dev: 0
      PRECISE
Velocity Y:
    Sq. Error Mean:     0
      ACCURATE
    Sq. Error Std. Dev: 0
      PRECISE
Pressure:
    Sq. Error Mean:     0
      ACCURATE
    Sq. Error Std. Dev: 0
      PRECISE
```

(a) Comparison of Equal States

```
Velocity X:
    Sq. Error Mean:     0.0233842
       INACCURATE
    Sq. Error Std. Dev: 0.0996487
       IMPRECISE
Velocity Y:
    Sq. Error Mean:     0.00566354
       INACCURATE
    Sq. Error Std. Dev: 0.0139529
       IMPRECISE
Pressure:
    Sq. Error Mean:     0.0214799
       INACCURATE
    Sq. Error Std. Dev: 0.0511252
       IMPRECISE
```

(b) Comparison of Unequal States

Figure 5.5: Examples outputs from the comparison tool.

# 6 Implementation

## 6.1 Library Selection

| | OpenGL | Vulkan |
|---|---|---|
| OpenCL | Y | N |
| CUDA | Y | Y |
| OpenGL | Y | N |
| Vulkan | N | Y |

Figure 6.1: Graphics and Compute Backend Interoperability Matrix

CUDA and Vulkan had already been highlighted in the Specification as likely choices of backends, but to be complete other backends were also considered. As the simulation would have to run on DCS systems (Requirement NF7), and thus run on Linux, the only possible GPU rendering backends were OpenGL and Vulkan. However there were still multiple choices of compute backend:

- OpenCL[67] is an "Open Standard for Parallel Programming of Heterogeneous Systems"[66].
- CUDA[52] is a closed-source library for running parallel code on NVIDIA GPUs.
- OpenGL has Compute Shaders[60] which can execute computations outside of the graphics pipeline.
- Vulkan also has Compute capability[38], similar in function to OpenGL.

To decide on the compute backend to use, an interoperability matrix was drawn (Fig. 6.1) to show which libraries could share data without copying it between buffers. As the researcher was already experienced with Vulkan, and the more granular control it provides would be beneficial to performance, Vulkan was selected as the rendering backend. This prevented OpenCL and OpenGL from being used as compute backends, as they are not compatible with Vulkan. CUDA and Vulkan have comparable ability, but CUDA was chosen as the compute backend. The Vulkan compute shaders are still a very graphics-oriented view of computation, and CUDA would give the researcher experience with other kinds of libraries. A Vulkan compute backend may still be used for the visualization portion of the code.

In other cases there were clear choices: the SDL2[62] window and input library and the Dear ImGUI[15] UI library were chosen due to personal experience. The `stb_image.h` header was found to be a simple method of importing image color data as byte arrays, used for the input generator (Requirement F3).

There are a great many options for Command-Line parsing libraries, even more so because C++ is used instead of C. A recent survey of the possibilities[5] was whittled down to five options.

`getopt`[22], `argp`[25], and `gopt`[71] are C libraries that use arrays of structures to define the required arguments. Of them, only `argp` can automatically generate a `--help` argument, which is a very valuable feature.

`cxxopts`[36] was considered as a C++ alternative, but used very odd syntax for defining arguments. Ultimately CLI11[11] was chosen as a modern C++11 library that had native support for subcommands, which were used heavily for separating program components (see Section 5.1).

## 6.2 Build System

The build system is implemented in CMake[13] as specified in Section 7.3. This section highlights a few changes that were made to an otherwise standard setup to accommodate the project.

### 6.2.1 CUDA-less Binaries

The project can be built to produce both CUDA and CUDA-less binaries, in case it needs to be run on CUDA-less computers. The list of regular C++ source files and CUDA source files are maintained separately. A CUDA-less binary (`sim_nocuda`) will only build the C++ files while a CUDA binary (`sim_cuda`) will build both. When building the `sim_cuda` target the preprocessor macro `CUDA_ENABLED` is defined throughout all source files, including the C++ files. This allows support for CUDA backends in C++ code (i.e. as selectable options on the command-line) to be conditionally enabled without maintaining two copies of the relevant source files. In Fig. 6.2 (which has been amended for brevity), the switch statement only contains a case for CUDA if the directive is set, triggering a fatal error otherwise.

```
switch(backendType) {
    case Null:
        return SimFixedTimeRunner<NullSimulation, Host2DAllocator>();
    case CpuSimple:
        return SimFixedTimeRunner<CpuSimpleSimBackend, Host2DAllocator>();
    case CpuOptimized:
        return SimFixedTimeRunner<CpuOptimizedSimBackend, Host2DAllocator>();
    case CpuAdapted:
        return SimFixedTimeRunner<CpuOptimizedAdaptedSimBackend, Host2DAllocator>();
#if CUDA_ENABLED
    case CUDA:
        return SimFixedTimeRunner<CudaBackendV1<true>, CudaUnified2DAllocator>();
#endif
    default:
        FATAL_ERROR("Enum val %d doesn't have an ISimFixedTimeRunner!\n", backendType);
}
```

Figure 6.2: An example of conditionally supporting CUDA based on a preprocessor directive.

### 6.2.2 Shader Build Infrastructure

The shaders used for visualization are written in GLSL, with appropriate extensions to be compatible with Vulkan. They are separated by file type, with Vertex shaders in `.vert` files and Fragment shaders in `.frag` files. As Vulkan does not natively support GLSL, they must be compiled to SPIR-V before they can be used. CMake does not support GLSL as a first-class language, so a custom build command was used to compile them with `glslc`[28] when they change. This allows them to be treated just like any other source file from the programmer's perspective. The SPIR-V files are placed in a `shaders` directory next to the binaries, where they can be easily accessed and passed to Vulkan.

## 6.3 Memory Usage

The CUDA simulation backend makes use of CUDA Unified Memory[32] whenever possible. This allows the memory to be accessed from the CPU and the GPU without having to manually map it, and instead pages the memory between devices on request. Once the pages are on the GPU, it can be accessed with the same speed as manually allocated GPU memory. The main benefit of this is that any functionality that has not yet been implemented on the GPU, or that may be faulty, can be easily replaced with pre-existing CPU code to verify the simulation correctness. Moving memory between the CPU & GPU does decrease performance, as any memory movement would, but as it is only intended for development this is OK.

Memory that needs to be shared between CUDA and Vulkan is allocated through Vulkan and then mapped to a CUDA pointer via the `VK_KHR_external_memory_fd` Vulkan extension[37] and the CUDA External Memory API[53]. It is possible for memory to be imported from CUDA into Vulkan, but allocating through Vulkan gives more control over where memory is allocated. Because of the limited flexibility of Vulkan-controlled memory compared to Unified Memory, this is used sparingly and only for data that absolutely must be shared between the APIs.

Vulkan memory is only used when Vulkan is present i.e. during the real-time visualization. In the headless mode, all of the memory is CUDA Unified Memory. This could lead to problems if the simulation assumes memory is Unified and CPU-accessible when it would be Vulkan memory during the visualization. In this case the code would break only during the real-time visualization when the pointer is accessed, and not during the headless simulation. To avoid this, a templated smart-pointer class `CUDAUnified2DArray<class T, bool IsUnified>` is used for each pointer to CUDA-usable memory. This automatically frees the memory when possible, and provides `as_gpu()` and `as_cpu()` functions which the programmer uses to access the data. C++ `static_assert` and `if constexpr` logic is used to create a compilation error when non-Unified memory is accessed through `as_cpu()`, preventing the issue from ever occurring.

The CUDA backend itself is also templated on whether it is being run with Vulkan-exported memory. This allows simulation code to detect whether the memory would be CPU-accessible or not, and take action accordingly. In Fig. 6.3, this is used to conditionally copy data into a Unified Memory buffer so that it can be used with a part of the algorithm that has not yet been implemented on the GPU. If the memory is already Unified, the copy is skipped, but if the memory is Vulkan-based the copy is performed to prevent a SEGFAULT. This logic uses `if constexpr`, so all branches are eliminated at compile-time adding a very slight performance boost.

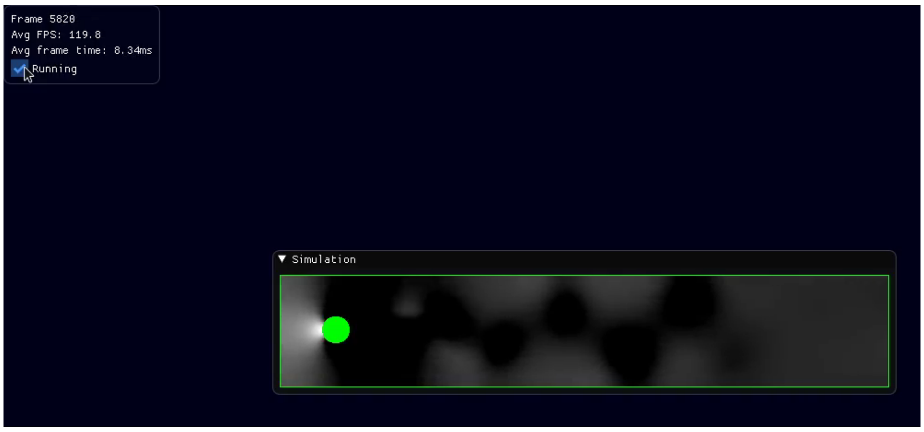
## 6.4 Current Status

The `makeinput`, `compare`, and `renderppm` subcommands are all functional. More functionality may be added if necessary for efficient development, and potential changes have been outlined in the previous sections.

```
if constexpr (UnifiedMemoryForExport) {
    // The buffer is Unified Memory, so use it directly
    OriginalOptimized::splitToRedBlack(p.joined.as_cpu(),
                                       p_buffered.red.as_cpu(),
                                       p_buffered.black.as_cpu(),
                                       imax, jmax);
} else {
    // The buffer is not unified memory,
    // so create a new Unified memory buffer and copy the data in,
    // then use that instead.
    CudaUnified2DArray<float, true> p_unified(unifiedAlloc.get(),
    matrix_size);
    p_unified.memcpy_in(p.joined);
    OriginalOptimized::splitToRedBlack(p_unified.as_cpu(),
                                       p_buffered.red.as_cpu(),
                                       p_buffered.black.as_cpu(),
                                       imax, jmax);
}
```

Figure 6.3: An example of conditionally changing code based on memory type.

The `fixedtime` headless simulation is functional, with each backend performing as intended. More optimization is intended (Section 2.2), but the current program is fast enough to simulate the original ACA input at 120 ticks per second[10] which is a good baseline.



```
Frame 5820
Avg FPS: 119.8
Avg frame time: 8.34ms
Running
```

Figure 6.4: An example of the real-time visualization running on the ACA input.

The visualized simulation creates a 1280x720 window and displays the simulation in a subwindow, which the user can move around. The visualization is a simple display of the current pressure values, which is subpar (see Section 2.3.2), but this is planned to change. A second window displays statistics about the last frame, and shows a checkbox which controls if the simulation is running as per Requirement F5.1. As seen

[10]On the researcher's GTX 1080. The program hasn't been tested on other systems.

in this window, the simulation is running at 120 frames per second. Each simulation tick is 1/60th of a second of simulation time, so the simulation is running at 2x real time. This will be changed to account for Requirement F5.6.

# 7 Project Management

## 7.1 Software Development Methodology

Plan-driven solutions depend on a rigid specification being completed before development[3], which does not fit with the more abstract goals of the visualization portion. Additionally some of the main advantages of plan-driven approaches only apply when introducing new team members and handling large teams. Neither scenario applies here, as only one person is undertaking active development.

For these reasons, an Agile approach was taken with a development cycle completing every two weeks. The goals for each development cycle were documented using Trello[69]. It was planned that the supervisor would be contacted every week with the current status of the project and the progress made in the current cycle. These contacts would either take place over e-mail if there were no pressing questions to ask, and otherwise take place on Microsoft Teams[46]. Unfortunately for the first few weeks this did not happen, as other work was vying for attention and preventing project work from taking place. This has been resolved in Week 5, and there is now frequent email correspondence.

## 7.2 Project Timeline

The project was split into multiple tasks to schedule it effectively. These tasks are scheduled on both a Gantt Chart in Fig. 7.1, and as a table in Table 7.1. The timeline has been well followed, and this schedule has been left unchanged from the Specification. Note that over the Christmas break no work is scheduled, this is to allow time on the other courseworks the researcher will have due over that period.

Also note that the development of the visualization and optimization of the simulation are scheduled concurrently - this is to account for the fact that some strides in visualization may require extra optimizations to run in real-time. Both of these blocks end on Week 22, where development is then completely focused on the presentation.

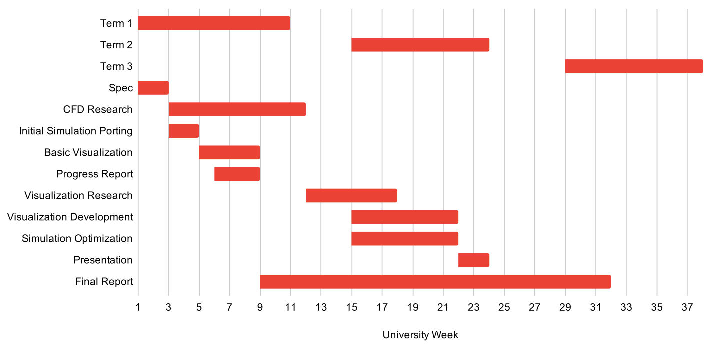


Figure 7.1: Project Schedule as a Gantt Chart

| Task | Start Week | End Week |
|---|---|---|
| Spec | 1 | 3 |
| CFD Research | 3 | 12 |
| Initial Simulation Porting | 3 | 5 |
| Basic Visualization | 5 | 9 |
| Progress Report | 6 | 9 |
| Visualization Research | 12 | 18 |
| Visualization Development | 15 | 22 |
| Simulation Optimization | 15 | 22 |
| Presentation | 22 | 24 |
| Final Report | 9 | 32 |

Table 7.1: Project Schedule Tasks

## 7.3 Tools

`gcc 9+`[26] is used to compile the program. This version has stable support for the C++14 and C++17 standards[8], allowing modern techniques to be used in the program. CMake[13] is used to handle building the program source files. Versions 3.8 and up support CUDA as a first-class language[44], which simplifies the compilation process. The CLion IDE[10] is used on the researcher's personal machine, as the researcher is familiar with the other IDEs in this family. If DCS machines are used, GNU Emacs[27] will be used to edit files instead.

Git[23] is used for source control, synchronized to a private GitHub[24] repository to avoid data loss. LaTeX[41] is used to create the various reports and non-program deliverables required by the project, which are hosted on Overleaf[1] so they can be accessed on Windows and Linux without an installed LaTeX environment.

Trello[69] is used to track bugs and upcoming features in each development cycle. Google Drive[12] is used to host other documents, i.e. scanned notes, that have been generated during development.

## 7.4 Risk Management

As the project continues, there are risks that may impede progress and even prevent the project from succeeding. Being aware of these risks allows them to be predicted ahead of time, avoided, or in the worst case mitigated once they arrive. Risk can be calculated with the following equation, where Severity and Likelihood are graded between 1 and 5.

$$Risk = Severity * Likelihood$$

Some of these risks have been encountered, including a new risk (Other Pressures) which was not accounted for in the Specification. Thankfully this risk did not cause a large delay.

### 7.4.1 Misscheduling

It is possible that the features outlined in Section 4 are too great to be implemented in the allotted time. In that case, the quality of work may have to be reduced to meet deadlines, or the schedule may need to be changed. This is especially relevant to the Visualization portion of the project, which cannot be fully scheduled yet.

**Risk** = 2 * 2 = 4

**Avoidance:** Previous projects should be used as a yardstick to predict how long implementing features will take, and inform the schedule. As new Visualization features are discussed, the impact on timing they each have should be considered.

**Contingency:** The scope of the project could be reduced to allow the report to be completed in time. A "code freeze" will be implemented close to the presentation deadline to ensure enough time is spent polishing the presentation and report.

### 7.4.2 Other Pressures

While the project schedule may have been well estimated based on the work required for the project, the amount of work required for other modules may be larger than expected. This manifested in Term 1, where the researcher took more modules than usual. Additionally, the removal of in-person lectures due to COVID-19 led to a lack of overall structure, which made organizing the other work more difficult.

**Risk** = 2 * 1 = 2

**Avoidance:** A more balanced set of modules between Term 1 and Term 2 could have helped resolve this, however on the other side of the coin the researcher now has fewer modules in Term 2 so this is unlikely to happen again. Next term the researcher will try to maintain a schedule for working on other module content, which should make up for a lack of in-person lectures.

**Contingency:** As before, the scope of the project could be reduced to allow the report to be completed in time. If module work is taking more time than expected by week 20, the code freeze could be pulled forwards to week 20 or 21 to spend more time on the presentation.

### 7.4.3 Loss of Hardware Access

As noted in Section 4.3, a GPU is required for the project to be tested and developed. Currently, the main development environment used is the researcher's personal computer, which has a suitable GPU. However, if this computer breaks down or is stolen, there is no readily available alternate environment. Under normal circumstances the Department of Computer Science labs would be used instead, as they also have suitable GPUs, but the current virus situation prevents this.

**Risk** = 5 * 1 = 5

**Avoidance:** Not possible.

**Contingency:** Student insurance could be used to purchase a new GPU/computer if it is stolen. Failing this, the DCS clusters could be used, but these will likely have high contention from other students who need to use GPUs remotely.

### 7.4.4 Illness

It is always prudent to consider the possibility that the stakeholders may fall ill and be unable to work on the project for some time. This is exacerbated by the current situation with COVID-19, making potential illnesses more dangerous than usual.

This risk manifested during Week 7, and delayed work on the project by three days. However the bulk of the current work had been completed by that point, and mostly other modules were affected.

**Risk** = 4 * 2 = 8

**Avoidance:** Not possible.

**Contingency:** The schedule would need to be changed to account for lack of time spent working. Some requirements may need to be reduced or removed entirely.

## 8 Testing

In order to measure the degree of success a project achieves, testing must be performed to verify the behaviour of the program is correct. Building tests also allows further development of the program to easily identify when new bugs are introduced. This section proposes potential effective tests, and documents the results of tests already performed since they were described in the Specification.

### 8.1 Unit Testing

The separate phases of the simulation are effective units of code. They could be automatically tested individually, or individual units could be swapped out for known working versions in order to pinpoint bugs found in wider tests. The latter method has been used for debugging during the simulation implementation.

The `makeinput` (Requirement F3), `compare` (Requirement F7), and `renderppm` program modes can also be tested as individual units with input/expected output combinations. These tests have not yet been implemented, but are planned as an extension.

### 8.2 Integration Testing

The "headless mode" outlined in Requirement F4 has functioned as an integration test for all of the simulation phases. Initially the CPU Simple and Optimized backends (Section 5.4) were added and tested against the original ACA program[9] and the submitted coursework[64] using the provided testing tools. The Compare mode (Requirement F7) was then implemented and tested against the ACA testing tools to ensure it's behaviour was correct. The Optimized Adapted and CUDA backends (Section 5.4) were then added and compared to the required output to ensure that any deviations were small.[11]

While the visualization cannot be tested without some simulation data to visualize, that data does not necessarily need to be continuously simulated. Static simulation states may be created in order to test separate parts of the visualization, or multiple parts at once.

### 8.3 Overall Testing

The "visualization mode" from Requirement F5 should function as a full system test of the simulation with the visualization. Assuming the headless simulations are accurate, there should be a negligible difference in results from a visualized simulation.

[11]Because the simulation operates on single-precision floating point numbers, small changes to orders of operation or compiler optimizations could introduce small discrepancies at the bit level.

## 9 Conclusion

As planned, enough research has been done into the problem space to produce an effectively optimized program. There has been enough time to think through the design and implementation, and the schedule has been followed well to this point. Testing, where implemented, has been helpful. Enough future tests have been planned so that the final program will be robust.

The simulation program and associated state has been shown to fulfil many of the requirements already. The visualization fulfils the base Requirement F5, but not Requirements F5.2 to F5.6. These will be added next term. Furthermore, there are possible program extensions listed in Appendix A.

All in all, this is a very good place to be in with respect to the schedule. The work done here should allow plenty of time next term to be devoted to the visualization, which should produce a very polished final program, report and presentation.

## 10 References

All URLs accessed on October 14th 2020 unless otherwise specified.

[1] *About us - Overleaf, Online LaTeX Editor*. 2020. URL: `https://www.overleaf.com/about`.

[2] L. Adams and J. Ortega. 'A multi-color SOR method for parallel computation'. In: *ICPP*. 1982, pp. 53–56.

[3] James Archbold. *CS261 Software Engineering*. 2020.

[4] Atomic Heritage Foundation. *Computing and the Manhattan Project*. URL: `https://www.atomicheritage.org/history/computing-and-manhattan-project`.

[5] attractivechaos. *A survey of argument parsing libraries in C/C++*. August 2018. URL: `https://attractivechaos.wordpress.com/2018/08/31/a-survey-of-argument-parsing-libraries-in-c-c/`.

[6] *Autodesk Flow Design - A Virtual Wind Tunnel On Your Desktop*. 2014. URL: `https://www.youtube.com/watch?v=2RBOtd-Z8O8`.

[7] R.B. Bird, W.E. Stewart and E.N. Lightfoot. *Transport Phenomena*. Transport Phenomena v. 1. Wiley, 2006. ISBN: 9780470115398.

[8] *C++ Standards Support in GCC*. 2020. URL: `https://gcc.gnu.org/projects/cxx-status.html`.

[9] Adam Chester and Graham Martin. *CS257 Advanced Computer Architecture Coursework*. 2020.

[10] *CLion: A Cross Platform IDE for C and C++*. 2020. URL: `https://www.jetbrains.com/clion/`.

[11] CLIUtils. *CLI11*. URL: `https://github.com/CLIUtils/CLI11`.

[12] *Cloud Storage for Work and Home - Google Drive*. 2020. URL: `https://www.google.com/intl/en_in/drive/`.

[13] *CMake*. 2020. URL: `https://cmake.org/`.

[14] *CODE OF CONDUCT FOR BCS MEMBERS*. 2015. URL: `http://www.bcs.org/upload/pdf/conduct.pdf`.

[15] Omar Cornut. *Dear ImGui*. URL: `https://github.com/ocornut/imgui`.

[16] *Data Protection Act 2018, c. 12*. 2018. URL: `http://www.legislation.gov.uk/ukpga/2018/12/contents/enacted`.

[17] Rokiatou Diarra. 'Towards Automatic Restrictification of CUDA Kernel Arguments'. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE 2018. New York, NY, USA: Association for Computing Machinery, 2018, pp. 928–931. ISBN: 9781450359375. DOI: `10.1145/3238147.3241533`. URL: `https://doi.org/10.1145/3238147.3241533`.

[18] *Ethical Consent for Undergraduate Projects*. 2020. URL: `https://warwick.ac.uk/fac/sci/dcs/teaching/material/cs310/ethics`.

[19] G. Falkovich. *Fluid Mechanics.* Cambridge University Press, 2018. ISBN: 9781107129566.

[20] Zhe Fan et al. ‘GPU Cluster for High Performance Computing’. In: *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing.* SC ’04. USA: IEEE Computer Society, 2004, p. 47. ISBN: 0769521533. DOI: `10.1109/SC.2004.26`. URL: `https://doi.org/10.1109/SC.2004.26`.

[21] ‘Fluid Dynamics on the Big Screen’. In: *ANSYS Advantage* II.2 (2008), pp. 52–53. URL: `https://www.ansys.com/-/media/ansys/corporate/resourcelibrary/article/aa-v2-i2-fluid-dynamics-on-big-screen.pdf`.

[22] Free Software Foundation. *getopt(3): Parse options - Linux man page.* URL: `https://linux.die.net/man/3/getopt`.

[23] *Git.* 2020. URL: `https://git-scm.com/`.

[24] *GitHub - About.* 2020. URL: `https://github.com/about`.

[25] GNU Project. *Argp (The GNU C Library).* URL: `https://www.gnu.org/software/libc/manual/html_node/Argp.html`.

[26] GNU Project. *GCC 9 Release Series.* 2020. URL: `https://gcc.gnu.org/gcc-9/`.

[27] GNU Project. *GNU Emacs.* 2020. URL: `https://www.gnu.org/software/emacs/`.

[28] Google LLC. *shaderc.* URL: `https://github.com/google/shaderc/tree/main/glslc`.

[29] Michael Griebel, Thomas Dornseifer and Tilman Neunhoeffer. *Numerical simulation in fluid dynamics: a practical introduction.* SIAM, 1998.

[30] Francis H. Harlow and J. Eddie Welch. ‘Numerical Calculation of Time-Dependent Viscous Incompressible Flow of Fluid with Free Surface’. In: *Physics of Fluids* 8.12 (1965), p. 2182. ISSN: 00319171. DOI: `10.1063/1.1761178`. URL: `https://aip.scitation.org/doi/10.1063/1.1761178`.

[31] Mark Harris. *Optimizing Parallel Reduction in CUDA.* Tech. rep. URL: `https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf`.

[32] Mark Harris and NVIDIA. *Unified Memory for CUDA Beginners | NVIDIA Developer Blog.* June 2017. URL: `https://developer.nvidia.com/blog/unified-memory-cuda-beginners/`.

[33] IEEE and The Open Group. ‘Utility Conventions’. In: *The Open Group Base Specifications* 1.7 (2018). URL: `https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap12.html`.

[34] Intel Corporation. *Introduction to Intel® Advanced Vector Extensions.* URL: `https://software.intel.com/content/www/us/en/develop/articles/introduction-to-intel-advanced-vector-extensions.html`.

[35] Antony Jameson, Luigi Martinelli and J Vassberg. ‘Using computational fluid dynamics for aerodynamics–a critical assessment’. In: *Proceedings of ICAS*. 2002, pp. 2002–1.

[36] jarro2783. *cxxopts: Lightweight C++ command line option parser*. URL: `https://github.com/jarro2783/cxxopts`.

[37] James Jones and Jeff Juliano. *VK_KHR_external_memory_fd*. The Khronos Group Inc, 2016. URL: `https://www.khronos.org/registry/vulkan/specs/1.2-extensions/man/html/VK_KHR_external_memory_fd.html`.

[38] Khronos. *Vulkan 1.1 Reference Guide*. Tech. rep. URL: `www.khronos.org/vulkan`.

[39] O Kreylos et al. ‘Interactive Visualization and Steering of CFD Simulations’. In: *Proceedings of the Symposium on Data Visualisation 2002*. VISSYM ’02. Goslar, DEU: Eurographics Association, 2002, pp. 25–34. ISBN: 158113536X.

[40] M Kuba, C D Polychronopoulos and K Gallivan. ‘The Synergetic Effect of Compiler, Architecture, and Manual Optimizations on the Performance of CFD on Multiprocessors’. In: *Supercomputing ’95:Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*. 1995, p. 72.

[41] *LaTeX - A document preparation system*. 2020. URL: `https://www.latex-project.org/`.

[42] ‘L2 norm’. In: *Encyclopedia of Biometrics*. Ed. by Stan Z. Li and Anil Jain. Boston, MA: Springer US, 2009, pp. 883–883. ISBN: 978-0-387-73003-5. DOI: `10.1007/978-0-387-73003-5_1070`. URL: `https://doi.org/10.1007/978-0-387-73003-5_1070`.

[43] *Library Search*. 2020. URL: `http://encore.lib.warwick.ac.uk/iii/encore/record/C__Rb1204273`.

[44] Robert Maynard. *[CMake] [ANNOUNCE] CMake 3.8.0 available for download*. 2017. URL: `https://cmake.org/pipermail/cmake/2017-April/065294.html`.

[45] Medvecký-Heretik Jakub. ‘Real-time Water Simulation in Game Environment’. PhD thesis. Masaryk University, Faculty of Informatics, 2018.

[46] *Microsoft Teams | Group Chat, Team Chat & Collaboration*. 2020. URL: `https://www.microsoft.com/en-gb/microsoft-365/microsoft-teams/group-chat-software`.

[47] Jean-Michel Muller et al. ‘The Fused Multiply-Add Instruction’. In: *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010, pp. 151–179. DOI: `10.1007/978-0-8176-4705-6{\_}5`. URL: `https://link.springer.com/chapter/10.1007/978-0-8176-4705-6_5`.

[48] NASA. *Definition of Streamlines*. URL: `https://www.grc.nasa.gov/WWW/k-12/airplane/stream.html`.

[49] Tianyun Ni. ‘Direct Compute - Bring GPU Compute to the Mainstream’. 2009.

[50] B. D. Nichols and C.W. Hirt. ‘Methods for Calculating Multi-Dimensional, Transient, Free Surface Flows Past Bodies’. In: *First International Conference on Numerical Ship Hydrodynamics* (20th–22nd October 1975). Ed. by Joanna W. Schot and Nils Salvesen. David W. Taylor Naval Ship Research and Development Center, 1975, pp. 253–278.

[51] Kyle E Niemeyer and Chih-Jen Sung. ‘Recent Progress and Challenges in Exploiting Graphics Processors in Computational Fluid Dynamics’. In: *J. Supercomput.* 67.2 (February 2014), pp. 528–564. ISSN: 0920-8542. DOI: 10.1007/s11227-013-1015-7. URL: https://doi.org/10.1007/s11227-013-1015-7.

[52] NVIDIA. *CUDA Zone | NVIDIA Developer.* 2020. URL: https://developer.nvidia.com/cuda-zone.

[53] NVIDIA. ‘External Resource Interoperability’. In: *CUDA Toolkit Documentation.* Vol. 11. URL: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__EXTRES__INTEROP.html.

[54] NVIDIA. ‘Global Memory - CUDA C++ Programming Guide’. In: v11.1.1 (). URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#global-memory-3-0.

[55] NVIDIA. *How to Access Global Memory Efficiently in CUDA C/C++ Kernels | NVIDIA Developer Blog.* URL: https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/.

[56] NVIDIA. *NVIDIA CUDA Programming Guide.* 2007.

[57] OpenMP. *Home - OpenMP.* URL: https://www.openmp.org/.

[58] M Perić, R Kessler and G Scheuerer. ‘Comparison of finite-volume numerical methods with staggered and colocated grids’. In: *Computers & Fluids* 16.4 (1988), pp. 389–403.

[59] Craig W. Reynolds. ‘Flocks, herds, and schools: A distributed behavioral model’. In: *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1987.* New York, New York, USA: Association for Computing Machinery, Inc, August 1987, pp. 25–34. ISBN: 0897912276. DOI: 10.1145/37401.37406. URL: http://portal.acm.org/citation.cfm?doid=37401.37406.

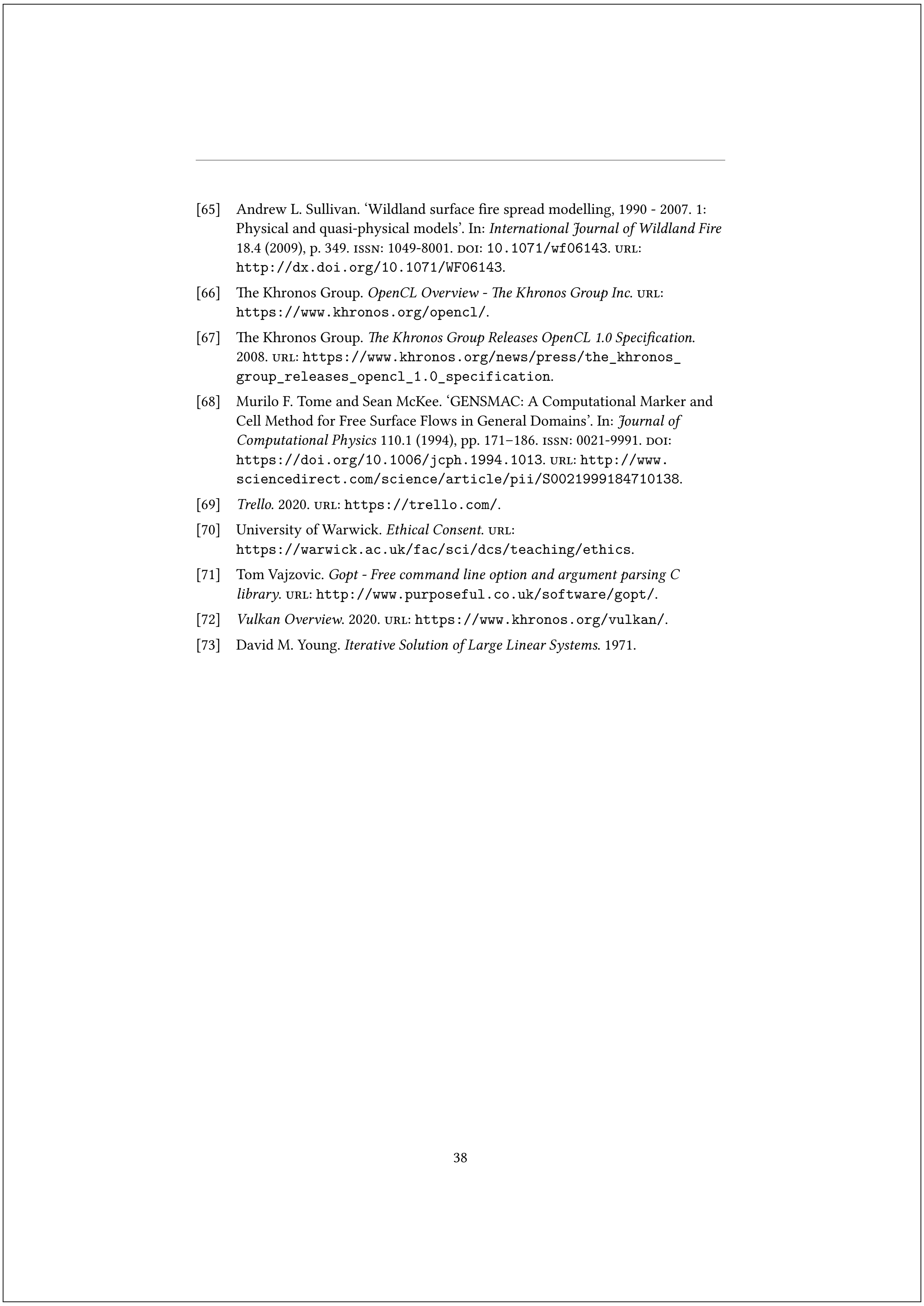
[60] Graham Sellers et al. *ARB_compute_shader.* URL: https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_compute_shader.txt.

[61] Peter Sikachev. ‘Real-Time Fluid Simulation in Shadow of the Tomb Raider’. 2018.

[62] *Simple DirectMedia Layer - Homepage.* URL: https://www.libsdl.org/.

[63] Jos Stam. ‘Stable Fluids’. In: *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques.* SIGGRAPH ’99. USA: ACM Press/Addison-Wesley Publishing Co., 1999, pp. 121–128. ISBN: 0201485605. DOI: 10.1145/311535.311548. URL: https://doi.org/10.1145/311535.311548.

[64] S. Stark. ‘CS257 Report - Reducing the Execution Time of a Fluid Simulation Program’. In: (2020).



[65] Andrew L. Sullivan. ‘Wildland surface fire spread modelling, 1990 - 2007. 1: Physical and quasi-physical models’. In: *International Journal of Wildland Fire* 18.4 (2009), p. 349. ISSN: 1049-8001. DOI: `10.1071/wf06143`. URL: `http://dx.doi.org/10.1071/WF06143`.

[66] The Khronos Group. *OpenCL Overview - The Khronos Group Inc.* URL: `https://www.khronos.org/opencl/`.

[67] The Khronos Group. *The Khronos Group Releases OpenCL 1.0 Specification.* 2008. URL: `https://www.khronos.org/news/press/the_khronos_group_releases_opencl_1.0_specification`.

[68] Murilo F. Tome and Sean McKee. ‘GENSMAC: A Computational Marker and Cell Method for Free Surface Flows in General Domains’. In: *Journal of Computational Physics* 110.1 (1994), pp. 171–186. ISSN: 0021-9991. DOI: `https://doi.org/10.1006/jcph.1994.1013`. URL: `http://www.sciencedirect.com/science/article/pii/S0021999184710138`.

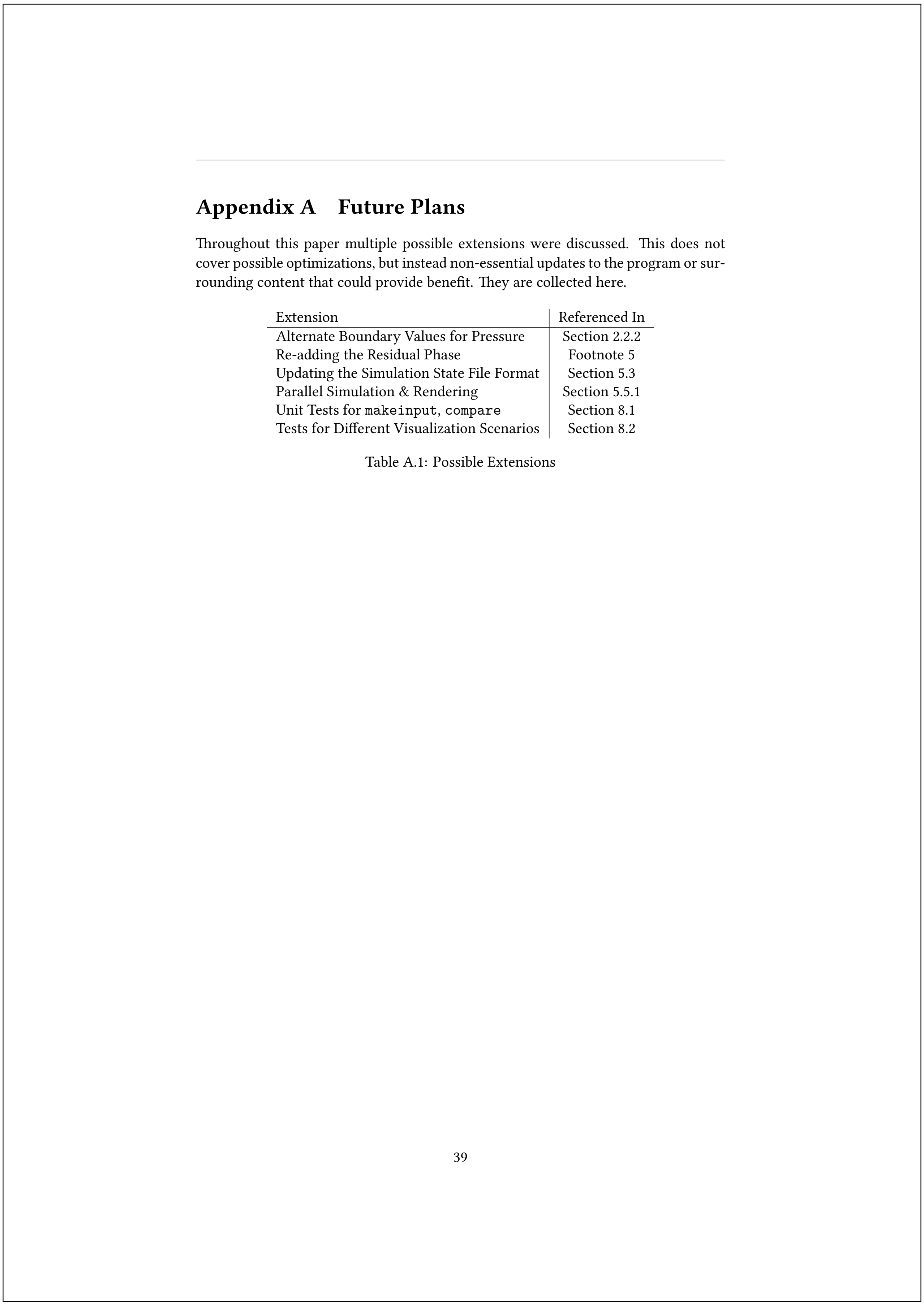
[69] *Trello.* 2020. URL: `https://trello.com/`.

[70] University of Warwick. *Ethical Consent.* URL: `https://warwick.ac.uk/fac/sci/dcs/teaching/ethics`.

[71] Tom Vajzovic. *Gopt - Free command line option and argument parsing C library.* URL: `http://www.purposeful.co.uk/software/gopt/`.

[72] *Vulkan Overview.* 2020. URL: `https://www.khronos.org/vulkan/`.

[73] David M. Young. *Iterative Solution of Large Linear Systems.* 1971.



## Appendix A Future Plans

Throughout this paper multiple possible extensions were discussed. This does not cover possible optimizations, but instead non-essential updates to the program or surrounding content that could provide benefit. They are collected here.

| Extension | Referenced In |
|---|---|
| Alternate Boundary Values for Pressure | Section 2.2.2 |
| Re-adding the Residual Phase | Footnote 5 |
| Updating the Simulation State File Format | Section 5.3 |
| Parallel Simulation & Rendering | Section 5.5.1 |
| Unit Tests for `makeinput`, `compare` | Section 8.1 |
| Tests for Different Visualization Scenarios | Section 8.2 |

Table A.1: Possible Extensions