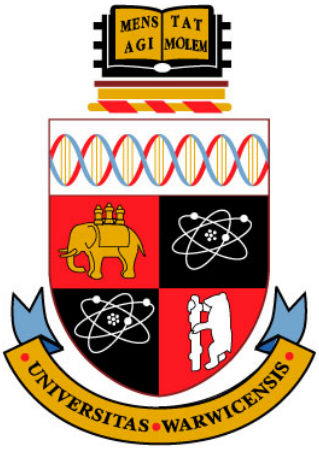


Performance Optimisation and Visualisation for a Fluid Dynamics Simulation

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 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 $Re \longrightarrow 0$ the viscosity of the fluid approaches infinity, and as $Re \longrightarrow \infty$ 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.

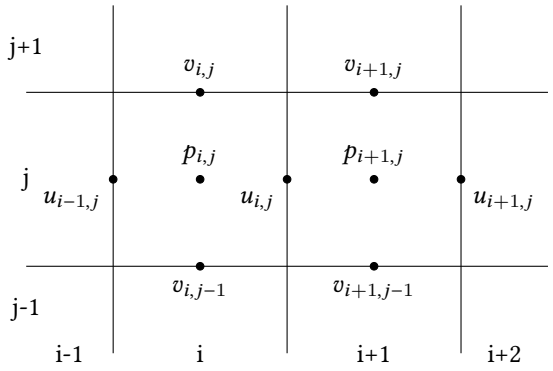


Figure 2.1: Discretization points for each variable on the staggered grid[29]

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 δx, δy. 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 δt. 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 δt 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]

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

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

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

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

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)

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

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)

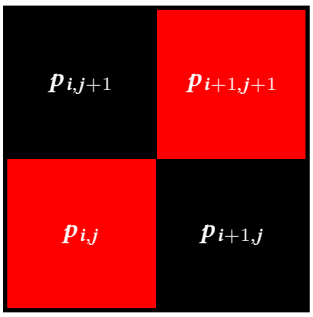


Figure 2.2: Example checkerboard pattern used for red/black splitting.

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 β 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 ϵ 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 β precalculation increased speed by 1.4x, and the removal of ϵ 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.

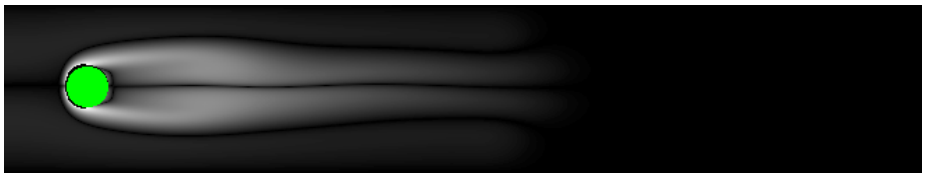
This project aims to develop new visualization techniques for two-dimensional simulations that are more intuitive than the current offerings, that can be extended to three dimensions easily. Using high-speed rendering APIs like Vulkan[72] will allow these visualizations to be made even more complex while 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 (ζ), the strength of vortical (a.k.a. rotational) motion at each point in the grid; and Stream Function (ψ), 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.

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

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



(a) Vorticity ζ

(b) Stream Function ψ

(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 ζ. 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 ≥ 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

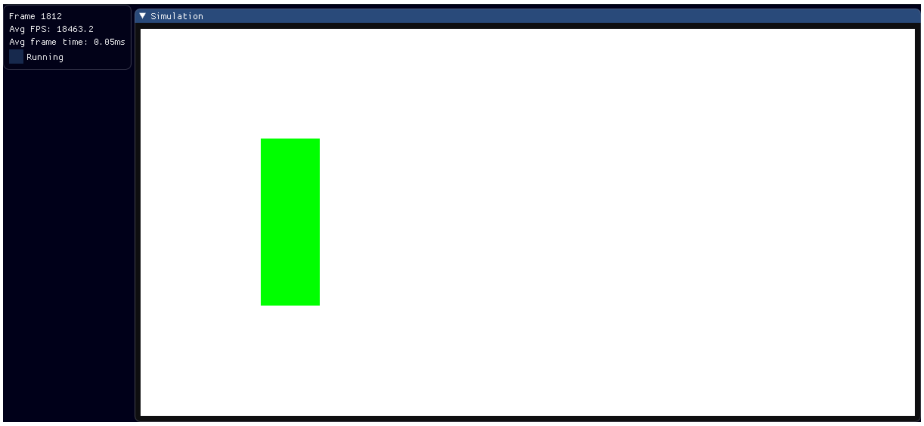
5.2 Generating Inputs

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



(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.

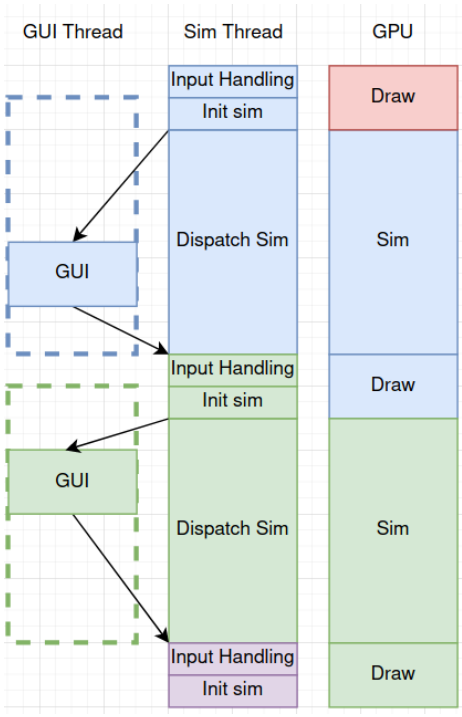


Figure 5.4: Thread utilization diagram.

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 δt 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 $1/60 = 16.67\,\mathrm{ms}$.

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.

```cpp
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.

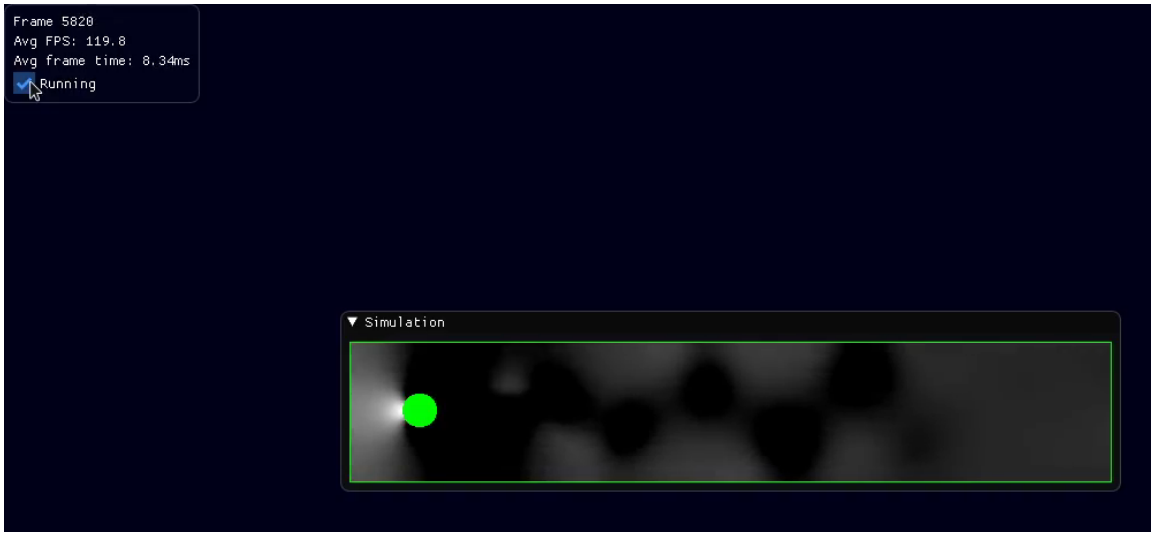


Figure 6.4: An example of the real-time visualization running on the ACA input.

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.

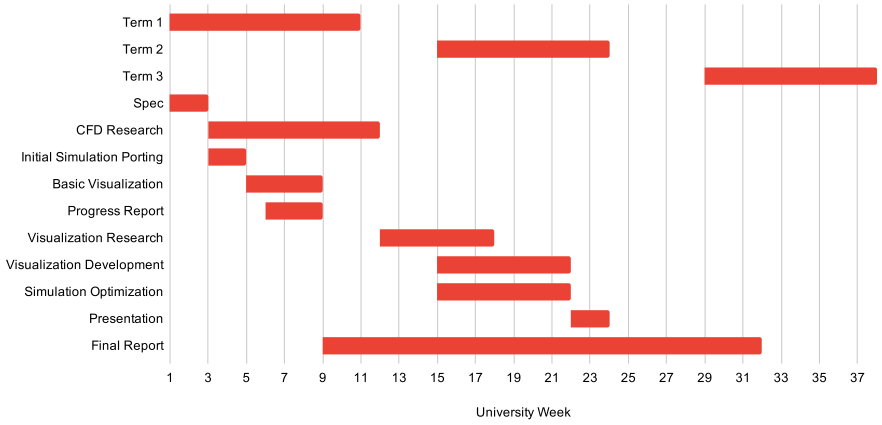


Figure 7.1: Project Schedule as a Gantt Chart

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