Computer Science NEA: Greenpower Fluid Dynamics Simulator

Sam Glover - Candidate 5156 at centre 16437

25th March 2024

Contents

I Analysis Stage 5

1 Project Proposal 6
1.1 Brief Outline . . . 6
1.2 Languages and Frameworks . . . 6
1.3 Project Walkthrough . . . 6
1.3.1 File Creation . . . 6
1.3.2 Configuring the Simulation . . . 7
1.3.3 Running the Simulation . . . 7
1.3.4 Drawing Obstacles . . . 7

2 Identification of User 8
2.1 Principal User . . . 8
2.2 Intended Wider Users . . . 8

3 Background 9
3.1 The Greenpower Organisation . . . 9
3.2 The CRGS Greenpower Team . . . 9

4 Research 10
4.1 User Interface . . . 10
4.1.1 Layouts of UI . . . 10
4.1.2 UI design Patterns . . . 12
4.2 Types of Visualisation . . . 12
4.3 Definition of Obstacles . . . 14
4.3.1 Constraint-Defined Obstacle Creation . . . 14
4.3.2 Drawing Obstacles in the Program . . . 14
4.4 Mathematics and Backend . . . 14
4.4.1 Process for CFD Simulation . . . 14
4.4.2 Stefan Bartels Paper . . . 15
4.4.3 Numerical Simulation in Fluid Dynamics Book . . . 15
4.4.4 Parallel SOR Algorithm . . . 15

5 Interview 16
5.1 A Note on Notation . . . 16
5.2 Transcript . . . 16
5.3 Objectives . . . 20

II Design Stage 21

6 Overall Project Structure 22
6.1 Overall Project Structure . . . 22
6.1.1 File Maker . . . 22
6.1.2 User Interface . . . 22

6.1.3 Visualisation . . . 22
6.1.4 Backend . . . 22
6.2 Project Structure Diagram . . . 22

7 User Interface 24
7.1 User Interface Structure . . . 24
7.2 UserControls . . . 24
7.3 Management of Parameters Throughout the Program . . . 24
7.4 Management of Units Throughout the Program . . . 24
7.5 Data Binding . . . 25
7.5.1 Converters . . . 25
7.6 Other Classes . . . 26
7.6.1 Commands . . . 26
7.6.2 Queues . . . 26
7.7 Class Diagram . . . 26

8 Definition of Obstacles 28
8.1 Reading In of File . . . 28
8.1.1 Constraint-Defined File Creation . . . 28
8.1.2 Object-Oriented Structure . . . 28
8.2 Click-and-Drag Node System . . . 33
8.2.1 Definition of Control Points . . . 33
8.2.2 Use of Polar Coordinates . . . 33

9 Flow Visualisation 34
9.1 Visualisation Methods in CFD . . . 34
9.1.1 Visualising Function Output . . . 34
9.1.2 Particle Tracing and Streaklines . . . 34
9.1.3 Streamlines . . . 35
9.2 Visualisation Design . . . 36
9.2.1 Colour Gradient . . . 36
9.2.2 Contour Drawing . . . 36

10 Discretisation and Numerical Solution 38
10.1 Introduction . . . 38
10.2 Discretising Space . . . 38
10.2.1 One-Dimensional Example . . . 38
10.2.2 Extending to the 2-Dimensional Case . . . 38
10.2.3 Staggered Grid Method . . . 39
10.2.4 The Navier-Stokes Equations . . . 39
10.3 Discretising Time . . . 39
10.4 Numerical Solution Algorithm . . . 40
10.4.1 Time-Stepping Loop . . . 40
10.4.2 Discrete Momentum Equations . . . 40
10.4.3 Solving the Pressure Equation . . . 41
10.4.4 Timestep Restriction . . . 41
10.5 Dealing with Obstacles . . . 41
10.5.1 Boundary Conditions . . . 41
10.5.2 Obstacles . . . 42
10.5.3 Implementation . . . 42
10.6 Calculation of Drag and Drag Coefficient . . . 43
10.6.1 An Introduction to Drag Forces . . . 43
10.6.2 Defining Equations . . . 43
10.6.3 Discretisation of Pressure Drag . . . 44

10.6.4 Discretisation of Viscous Drag 44
10.6.5 Discretisation of Wall Shear Stress 44

11 Parallelisation and Use of a GPU 45
11.1 The Merits of Parallelisation 45
11.2 GPU Architecture 45
11.2.1 GPU Terminology 45
11.2.2 Limitations and Considerations for GPU Programming 45
11.2.3 Thread Synchronisation 46
11.3 GPU Algorithms 46
11.3.1 Obstacle-Aware Algorithms 46
11.3.2 Reduction Kernels 47
11.3.3 Red-Black SOR 49
11.3.4 Drag Calculation 49

12 Backend Structure 50
12.1 CPU Backend 50
12.2 GPU Backend 50
12.3 Backend Library 50

13 Backend-Frontend Interface 54
13.1 Named Pipes 54
13.2 Communication Protocol 54
13.3 Serialisation 54
13.3.1 Primitive Data Types 54
13.3.2 Field Data 55
13.3.3 Obstacle Data 55

III Technical Solution 56

14 Examples of High-Level Skills in NEAFluidDynamics 57
14.1 Model 57
14.1.1 CPUBackend Staggered Grid (Complex Scientific/Mathematical model) 57
14.1.2 UserInterface OOP (Complex OOP and Dynamic Generation of Objects) 57
14.1.3 Backend OOP Model 57
14.2 Data Structures 58
14.2.1 GPUBackend Pitched Arrays (Complex array handling) 58
14.2.2 UserInterface Queues (Queue Operations) 58
14.2.3 FileMakerCLI Stacks (Stack Operations) 58
14.3 Algorithms 58

IV Testing 60

15 Test plan 61
15.1 Objective-Based Tests 61
15.2 User Input Tests 62
15.3 Use Case Test 62
15.4 Testing Video 63

V Evaluation **64**

16 Accordance With Objectives **65**

16.1 Backend / Computation Objectives 65
16.2 User Interface Objectives 65
16.3 Visualisation/Obstacle Defining Objectives 66

17 Feedback From User **67**

17.1 Interview 67
17.2 Conclusions 69

18 Areas For Improvement **70**

18.1 Backend 70
18.1.1 OpenCL 70
18.1.2 Adaptive Grid 70
18.1.3 Optimisation of Timestepping Loop 70
18.1.4 Drag Coefficient 71
18.1.5 Turbulence Modelling 71
18.2 User Interface 71
18.2.1 Overall Styling / Theme 71
18.2.2 Tooltips 71
18.3 Visualisation 71
18.3.1 Automatic Contour Tolerance 71
18.3.2 Automatic Min/Max Sliders 71
18.4 Obstacle Defining 72
18.4.1 Click-And-Drag Node System 72
18.4.2 Parsing of Common File Types 72
18.4.3 File Maker GUI 72

A Glossary of Terms **74**

B Explanation of Mathematical Symbols **76**

C Pipe Protocol Specification **78**

C.1 Requirements and Definitions 78
C.2 General Structure of Control Bytes 78
C.3 Precise Specification 79
C.4 Sample Usage 81

D Source code **83**

D.1 Project code structure 83
D.2 Source code 85

Part I
Analysis Stage

Chapter 1

Project Proposal

Sam Glover - Interactive 2D Computational Fluid Dynamics (CFD) simulator with GUI

1.1 Brief Outline

The program I want to develop is a Computational Fluid Dynamics (CFD) simulation program, that can simulate the airflow over obstacles. The user will first select a file (made with a second executable that allows drawing of obstacles via mathematical constraints) or select to draw the obstacles in the program. After changing relevant parameters, the user will start the simulation. When this happens, an executable to handle all of the processing (the backend) will be launched, and this will communicate with the user interface and send output data. CFD programs are well-suited to computation on a GPU, and therefore the backend will be able to run on a GPU to maximise performance. To visualise the data, a visualisation system will also run inside the UI. This will show the data output by the backend (for example, fluid velocity at each point) using a user-friendly method such as a colour gradient.

1.2 Languages and Frameworks

I plan to use C++ with CUDA (NVidia's GPU compute framework) for the backend simulation; C# and WPF (Microsoft's Windows UI framework) for the user interface; and OpenGL (a graphics library) for the visualisation.

1.3 Project Walkthrough

This section explains how each part of the project is interacted with by the user, and explains a typical use case for the entire project.

1.3.1 File Creation

Optionally, the user can create an obstacle file using the file creator. This is a CLI program that allows the user to input mathematical inequalities that can then be converted to a shape which is stored in the binary file format used by the simulation program. Constraints are mathematical inequalities in x and y which define regions in the x-y plane, and can then be converted to an obstacle. For example, the set of constraints in equation 1.1 encode a square. As the intended users of the project are budding engineers, this is suitable as they are likely to be familiar with the practice of defining a region using inequalities.

$$x \geq 90, x \leq 110, y \geq 45, y \geq 55 \tag{1.1}$$

The type of file that will be used is a binary file, which means that the data is in a *raw* format that needs little processing. The format is as follows: the size of the simulation domain is transmitted as metadata in the first 8 bytes (4 for x length then 4 for y length). Thereafter, each bit is used to represent whether that cell is obstacle (0) or fluid (1), in a column-first scan from bottom-left to top-right.

1.3.2 Configuring the Simulation

The user then launches the main simulation program, and is greeted with a start screen. This contains parameters the user can edit, an option to select the file they just made or draw their own obstacle in the following screen, and options to select the units in the program. A further screen exists for other parameters that are not commonly changed (advanced parameters).

1.3.3 Running the Simulation

The user then presses simulate on the initial screen which starts the simulation running. The user then sees a different screen, which contains the flow visualisation. The user also has options to configure parameters (like fluid speed) while the simulation is running, and to change the look of the visualisation. At this point, the user also has the option to close the program or go back to the start screen.

1.3.4 Drawing Obstacles

Also, if the user has not loaded a file and instead elected to draw their own obstacle, the user can interact with the click-and-drag node system to draw their obstacle. The user moves around control points to create the obstacle, and presses finish when they are done. The obstacle is then converted to the same binary format that the files use, and sent to the backend.

Chapter 2

Identification of User

2.1 Principal User

The principal user shall be Ben Baxandall. He is a year 13 student and a member of the Greenpower team at CRGS, and has been since the start of year 12. He is interested in the idea of a CFD program to allow the team to improve the aerodynamics of the car. He is a member of the design engineering sub-team, and therefore wants the CFD simulator to help with evaluating his designs' aerodynamic performance.

2.2 Intended Wider Users

Mine and Ben's intentions are that the program will be able to be used by all the members of the CRGS Greenpower team, or at least the members of the design sub-team, and will need to be intuitive to use for those with little knowledge of the system or indeed CFD systems at all. I will therefore need to design the UI with this in mind.

As part of prototyping the system, I may consult with the entire Greenpower team on possible prototype UIs, to ensure that the whole team's feedback is taken into account in the final design. I will do this through feedback forums wherein I will meet with the team and show them a current prototype for the user interface.

Chapter 3

Background

3.1 The Greenpower Organisation

Greenpower is a charitable organisation that organises races for schools to compete with their own electric cars (see figure 3.1). The motor and batteries are specified by Greenpower, so key differentiators between cars are weight and aerodynamic performance. Weight can be simply measured, however aerodynamic performance is much harder to measure. Therefore, a method of simulation of car designs, especially in real-time, greatly aids the design of a successful car.

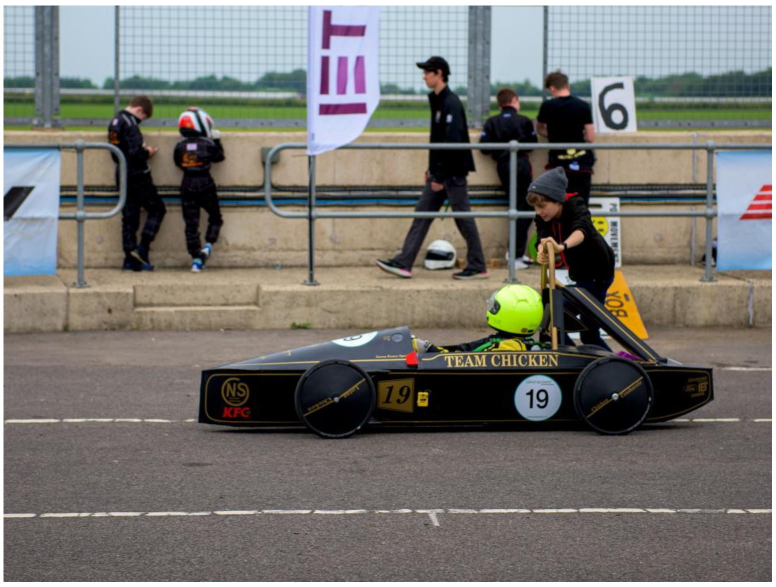


Figure 3.1: An example of a Greenpower car

3.2 The CRGS Greenpower Team

The CRGS Greenpower team, consisting of sixth-form students, design, build and race the car. The team competes in races in the summer, and makes changes to the design in the winter. Therefore, the timescale of this project is appropriate to the needs of the Greenpower team. The team currently has no way of gaining insight into the aerodynamic performance of their cars, since both professional software and taking the car to a real windtunnel are prohibitively expensive.

Chapter 4

Research

4.1 User Interface

4.1.1 Layouts of UI

As part of my research, I looked at multiple different UI designs from similar programs, so that these could be submitted as candidate UI layouts to the user. The images in the following pages show the results of this research. I tried to get a variety of different UI styles, made for programs as similar as possible to my project, to allow the user the best chance of selecting a UI design that works well. Of course, this will be taken further during the design stage and will undergo continuous feedback and improvement.

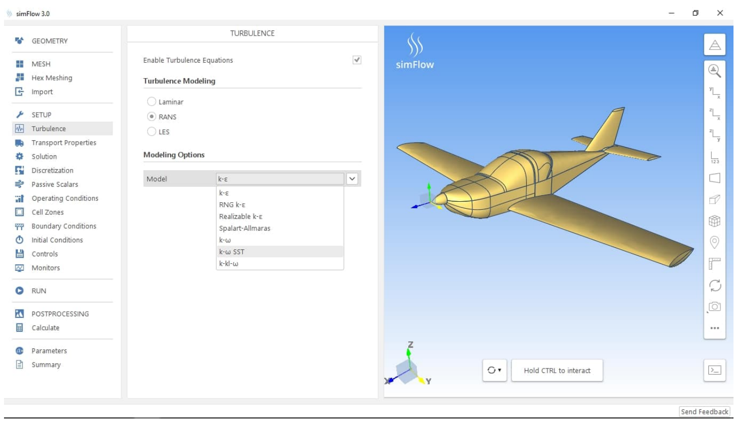


Figure 4.1: Side panel with secondary menus popping out to the side

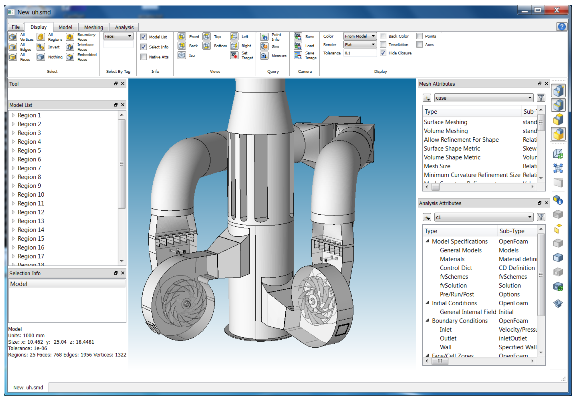
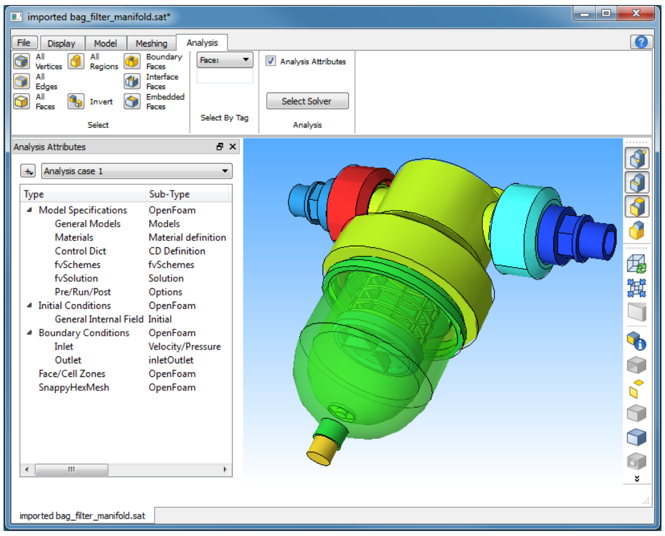
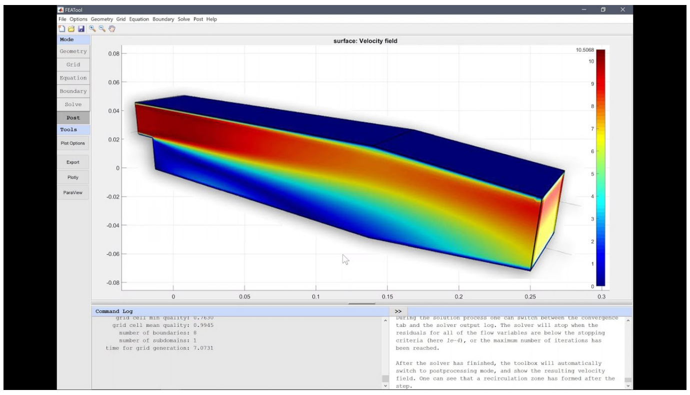


Figure 4.2: Ribbon with secondary menus as side panels



Type	Sub-Type
Model Specifications	OpenFoam
General Models	Models
Materials	Material definition
Control Dict	CD Definition
fvSchemes	fvSchemes
fvSolution	Solution
Pre/Run/Post	Options
Initial Conditions	OpenFoam
General Internal Field	Initial
Boundary Conditions	OpenFoam
Inlet	Velocity/Pressure
Outlet	inletOutlet
Face/Cell Zones	OpenFoam
SnappyHexMesh	OpenFoam

Figure 4.3: Ribbon with secondary menus as side popouts



```
  grid cell min quality: 0.7630
 grid cell mean quality: 0.9945
   number of boundaries: 8
   number of subdomains: 1
time for grid generation: 7.0731
```

During the solution process one can switch between the convergence tab and the solver output log. The solver will stop when the residuals for all of the flow variables are below the stopping criteria (here 1e-4), or the maximum number of iterations has been reached.

After the solver has finished, the toolbox will automatically switch to postprocessing mode, and show the resulting velocity field. One can see that a recirculation zone has formed after the step.

Figure 4.4: Side ribbon with secondary menus as bottom panels

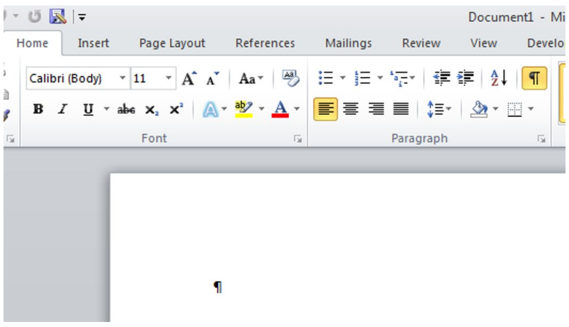


Figure 4.5: MS Office layout: top ribbon only

4.1.2 UI design Patterns

The UI is to use WPF, Microsoft's Windows app framework. This uses `XAML` (a markup language that is similar to XML) to define what the UI looks like, and uses C# to provide the functionality of the UI. After research into using WPF, I found that the most common design pattern (and the one that is best suited to the framework) is Model-View-ViewModel (MVVM). Figure 4.6 explains what each of these are and how they work together. The key advantage of MVVM is that each section can

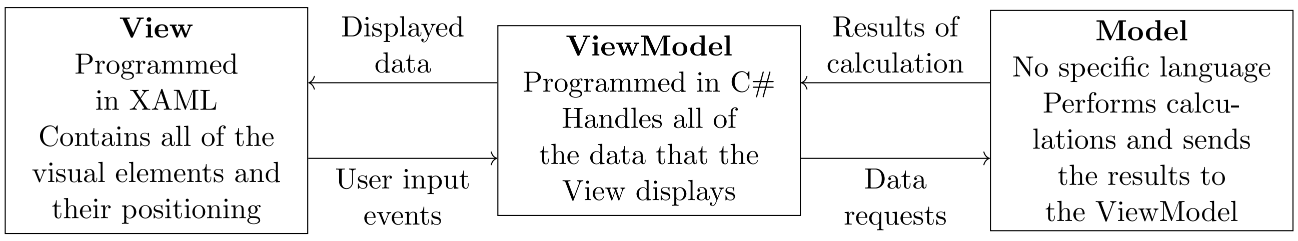

View
Programmed in XAML
Contains all of the visual elements and their positioning
Displayed data
User input events
ViewModel
Programmed in C#
Handles all of the data that the View displays
Results of calculation
Data requests
Model
No specific language
Performs calcu- lations and sends the results to the ViewModel

Figure 4.6: MVVM diagram

be decoupled and tested individually. For example, the View can be decoupled from the ViewModel and given static test data to make prototypes. It also encourages programming to interface rather than implementation (a pillar of good OOP programming) since each part depends on the exposed interfaces of the others.

4.2 Types of Visualisation

Similar to the UI layouts, I researched multiple different CFD visualisation methods to be able to give images to the user. A key consideration for the visualisation is the identifiability of patterns in the flow. This allows for identification, by the user, of areas where flow is turbulent (which would include drag) and where it is laminar, and therefore provides insight into where on the simulated object needs to be improved. The images on the following pages are the results of my research and will be submitted to the user.

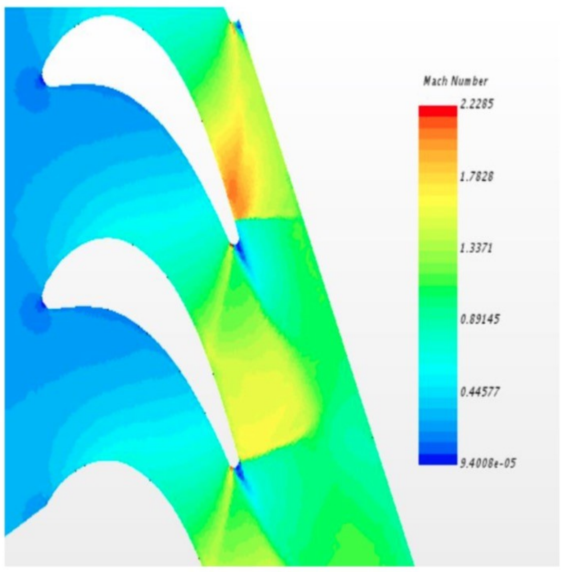


Figure 4.7: Colour only

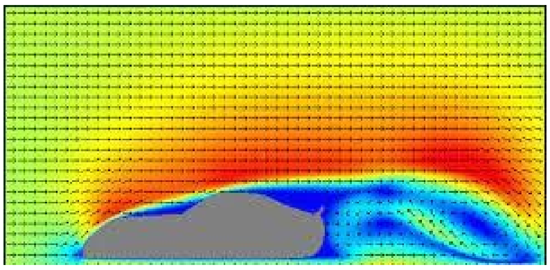


Figure 4.8: Colour with arrows

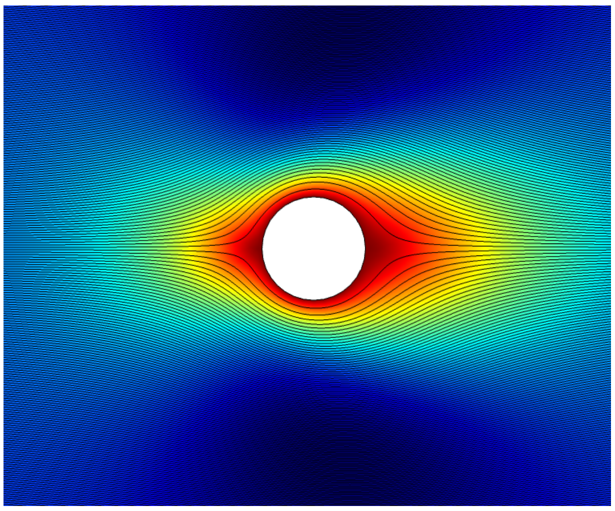


Figure 4.9: Colour and flow lines

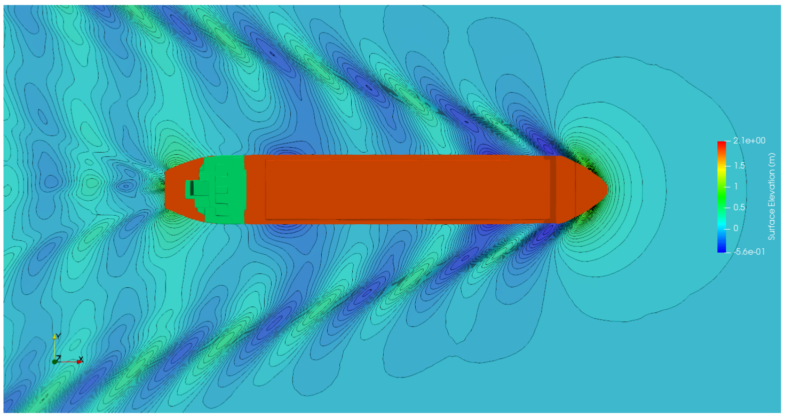


Figure 4.10: Colour and contour lines

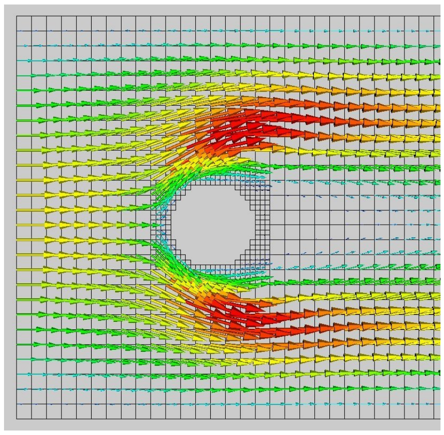


Figure 4.11: Coloured arrows

4.3 Definition of Obstacles

In the program, definition of obstacles will work in 2 main ways. Either the user will create a file (with a separate program), based on mathematical constraints (inequalities), or the user will choose to draw their own obstacle in the program.

4.3.1 Constraint-Defined Obstacle Creation

To create a file to be simulated in the program, the user will interact with a separate program that reads in mathematical inequalities and produces a binary file. This will be suitable because the users of the program are engineers and will understand how to use mathematical inequalities.

I therefore researched how to parse in mathematical expressions into a program. I researched the shunting-yard algorithm, which can convert a normal mathematical expression (such as $5 + 4$) into a postfix expression, where the operator is after the numbers (or variables) that it operates on. The expression $5 + 4$ then becomes 54+. This is easier for a computer to parse, and will have to form part of this program.

4.3.2 Drawing Obstacles in the Program

The other method is by not loading a file and instead drawing the obstacle in the program. I therefore researched methods to draw a shape, to be used inside the main program. Drawing the outline of the shape using the mouse would be possible, but it would be difficult for the program to figure out the *inside* and *outside* of the obstacle. Another alternative in research was splines, as documented in [3]. The precise algorithm for computing splines is further documented in Section 8.2

4.4 Mathematics and Backend

The initial idea for this project was that GPU acceleration would be used, to allow for efficient simulation in real-time (as was a requirement for the project). I knew that the Navier-Stokes equations would be the underlying mathematical equations for the simulation, since these are the key equations governing fluid dynamics. I therefore searched for papers and previous computational solutions for Navier-Stokes simulation with a GPU.

4.4.1 Process for CFD Simulation

The basic process for CFD is as follows:

1. **Loading of configurations and objects:** At this point parameters are defined, such as fluid velocity and behaviour, and objects are loaded in from a file. Further development of this section needs well-defined user requirements for how objects should be loaded and what parameters should be changed.

2. **Discretisation:** Discretisation, in this case, refers to the breaking down of the continuous problem (the number of particles to be simulated can be treated as infinite, therefore the 2D area to simulate can be said to be continuous) to a discrete one. This is done by assigning a grid to the continuous area and treating the small area as having the same inflow and outflow velocities, and pressure. Of course, as the size of these areas is decreased (and therefore the number of them increases), the model becomes more accurate to real flows.

3. **Solution of the discrete problem:** In this stage, the differential equations, which are now re-formed to represent the discrete grid, can be solved by a numerical method. This numerical method shall be researched, and discussed extensively in part II (p22) Design. I plan to handle this with GPU computation.

4. **Representation of flow:** This part is the most subjective and is also very dependent on user input, since there are multiple ways to represent the flow. Further research is in chapter 4.2 (p12).

4.4.2 Stefan Bartels Paper

One of the first papers that I found when researching the subject was a paper by Stefan Bartels, Chair of Scientific Computing at Munich University [4]. This paper describes an implementation of CFD using OpenCL (an alternative to CUDA). This paper introduced me to the considerations when using a GPU, and also referenced the book Numerical Simulation in Fluid Dynamics by Griebel, Dornseifer and Neunoeffer [1]. The paper by Bartels is more modern, and I use it as an example of a modern CFD program on modern hardware, in contrast to [1] which was published in 1998 and therefore does not discuss current hardware.

4.4.3 Numerical Simulation in Fluid Dynamics Book

This book was a key element of my research into how the backend computation works. It goes into great detail about numerical simulation on computers, an emerging field at the time of publication, and the algorithms and mathematics behind this. Greater detail is in Part II (p22) Design. The book also discusses the difference between laminar and turbulent flow, and the difficulty in dealing with turbulent flow.

4.4.4 Parallel SOR Algorithm

In [1], the SOR computation is the most important part of the numerical simulation and takes the most time to compute. Therefore, this would be a key section to optimise in the GPU backend. [2] discusses the Red-Black SOR implementation for parallel processors: this defines cells as either ”red” or ”black” and calculates the ”red” cells before the ”black” cells, in order to solve the issue of cell dependencies.

Chapter 5

Interview

5.1 A Note on Notation

In the following section, square brackets, [], will be used to reference something that was not said, but is added for clarity (for example, I may have gestured towards an element of an image in the interview and used the word *this*, but may put [side panel] in the transcript for clarity). Rounded brackets and italics, *()*, will be used for showing actions, such as showing Ben an image. Different colours are also used - blue is used for me (the interviewer) and BrickRed is used for Ben (the interviewee). Black is used for summaries or identification of objectives that I have put in after the interview ended.

5.2 Transcript

(Start of interview. Ben was notified that this would be recorded and used for an NEA project before recording started.) Sam: What sorts of parameters would you like to be able to control prior to the simulation? So, parameters would be quantities like fluid speed, how the fluid interacts with objects and the edges of the simulation, or fluid viscosity which is the stickiness of a fluid.

Ben: I think for Greenpower, I mean I was speaking to Will about the front of the car, it would be helpful to be able to edit the friction that the material experiences. Will was taking about slip friction, I think it was like the drag, I think the friction that the material would experience would be helpful. I don't know if that's possible?

Sam: Yeah that's definitely doable.

The program will be able to calculate drag on the obstacle.

Ben: Fluid speed would obviously be good, because I think the car reaches up to around 40 mph, so it would be good to vary that. I think everything you've said would be helpful. I mean, considering that next year there will be new year 12s who don't know all that much from the start, so not too advanced.

Fluid speed and material friction shall be editable parameters.

Sam: Ok fine. With those parameters, how would you like to change them before the simulation begins? Possible options could be a setup or config file, a popup before the simulation begins or command-line parameters. Which sounds most useful?

Ben: I think a popup with slider values, and if it could be a window that you can move to the side. I think that would be good. Something like a config file might be too complicated.

The parameters should be configurable via a popup before the simulation begins.

Sam: Obstacles: so these are things like the model of the car; how should obstacles be passed to the program?

Ben: Well we would want to load in models, but I think it would be helpful to be able to draw them. If you were iterating models, to be able to compare two different quickly-drawn models would be helpful.

The program should be able to read in a file and allow real-time drawing during the simulation.

Sam: Ok, how would you like the results of the simulation to be presented?

Ben: As in, how would they be displayed?

Sam: What do you want displayed, and how?

Ben: Well this is like the other files you sent me with the prototypes, isn't it?

Sam: Yeah, we'll come onto that in a sec.

Ben: Ok, well I think like areas of high impact or friction would be nice. Would you be able to have like a pinpoint, or like an area where you can get an average of all the forces experienced?

Sam: Yeah, so we can do drag coefficient, which is the overall drag on the obstacle. Does that sound like something that's useful?

Ben: Yeah that would be good - to be able to see an overall view.

Drag coefficient should be calculated for each obstacle submitted to the program.

Sam: Would you like to be able to move and edit the obstacle during the simulation, and if so, how do you envision this would be done?

Ben: I'm thinking if you had nodes, that you could drag to reshape the obstacle. Being able to erase, like a pen, where you could erase and draw.

obstacles will be able to be edited and reshaped using a click-and-drag-node system. I envision this to be somewhat complex as mathematical equations will need to be used to interpolate between the control points. Further research will be needed into different methods to achieve this in the design section.

Sam: Which parameters would you like to be able to edit in real-time (during the simulation)?

Ben: Speed would be good to change, with the possibility of using the data from the datalogger. I know that the datalogger outputs to an Excel spreadsheet, so if you can get those speeds and output it to another table in the excel spreadsheet.

Sam: Ok yeah, so it does like a speed versus time, drag coefficient versus time.

Ben: Yep that would be good. I mean, that's really complicated I think but that would be pretty helpful.

Sam: Yeah that's definitely doable.

Possible extension project: compute drag coefficient over time given datalogger input.

Sam: So now are some examples of visualisations. So first is the flow, so the actual thing inside the GUI - we're not talking about the layout of the GUI we're talking about the flow visualisation here. I've sent these to you already but I'll click through them. *(Shows figure4.7 (p12))* What're your thoughts on this one with just colour?

Ben: Just colour, ok.

Sam: Like this, where you've got the colours that show you the speed and things.

Ben: I mean it's not particularly clear, because these colours aren't particularly easy to differentiate. I think it would be better, I know it's a picture, but it would be easier if you could hover your mouse over certain areas and it would come up with a value instead of having to look between [the visualisation and the key]. I quite like the idea of having the colours though.

Sam: Colours, ok, good. What about this one? *(Shows figure 4.8 (p13))* This is colours with arrows showing where the fluid flows. What are your thoughts on this one? So we've got the colours again, but what are your thoughts on the arrows? Are they a good addition or actually make it more complicated to see?

Ben: I think the arrows make it look more complicated. When there's an arrow between sections [of colour], it's a little bit hard to understand. I think it would be a bit more, well, aesthetic if it was just the colour.

Sam: Ok. What about something like this *(shows figure 4.9 (p13))* where you've got the colours but also these lines which are showing you the direction of the fluid flow? Does that look good?

Ben: Yeah, that one looks a lot cleaner than the previous one.

Sam: I mean obviously they're all simulating different obstacles because they're just off the internet.

Ben: Yeah, I'm not thinking about the shape - I think if it was arrows this wouldn't look as good, I think it's better with just a plain line without the arrow. I think maybe it's because the arrow head takes up space - it makes it a bit more convoluted. But yeah, I like that one.

Sam: Yeah, and hopefully when the simulation's running in real-time and you're seeing all the fluids and things it will be clear which way it's flowing and you won't need the arrows as much. The other thing, do you understand what a contour line is?

Ben: Yeah I know it from Geography, on maps.

Sam: Yeah, so this one *(shows figure 4.10 (p13))* is using lines, but contour lines rather than flow lines so it's showing you, this is a boundary of approximately the same velocity or pressure or whatever this is measuring. What are your thoughts on this versus the previous one with the lines?

Ben: I'm not so sure - it wouldn't be as easy to interact with would it?

Sam: Yeah, well these lines would move when it's simulating, but obviously this is static and I'm worried this might look a bit weird when it's actually simulating.

Ben: Yeah, I think the other one is a lot cleaner, I feel like if I was going to take actual data then [figure 4.9 (p13)] would be a lot clearer.

Sam: Ok last one *(shows figure 4.11 (p13))* - here the arrows are actually coloured...

Ben: *(disgusted ooh)*

Sam: Yeah, from what you've said about arrows previously...

Ben: Yeah ooh

Sam: Yeah it's pretty shocking isn't it.

Ben: Yeah I don't particularly like that one!

Sam: I mean, that [figure 4.9 (p13)] was my favourite as well so that's very informative thank you.

Ben: Yeah, I like that one [figure 4.9 (p13)], that one's good.

Flow visualisation should be modelled on figure 4.9 (p13), with flow lines to differentiate between laminar and turbulent flow, and also to show interruptions in flow that could cause drag as the air flows over the drawn objects.

Sam: Ok, so same again but now we're doing UI layouts. *(Shows figure 4.1 (p10))* So assume this [pointing to the 3D model in figure 4.1 (p10)] is our simulated thing, we're not looking too much at this, I'm more getting you to look at the panels and things. So, this is where you've got a side panel already, and when you click one of the buttons it comes out as another side panel.

Ben: I think it would be good to be able to adjust the size of these panels.

Sam: Yeah, I would have thought you'd have a thing where you'd hover over and could change the size - like a reactive layout.

Ben: Yeah that would be good. As minimal as possible so I think quite a lot of dropdowns would be helpful. I quite like that.

Sam: Ok, how about this one? *(shows figure 4.2 (p10))* Where you've got a panel at the top and then secondary panels at the side. Do you think that's a bit cluttered or do you like it?

Ben: Yeah I think it's a little bit overcomplicated I think. What we'd be looking for for Greenpower is more like the one above. There'll be year 12s looking at that and, you know. I prefer a simpler kind of layout probably.

Sam: Alright. So this is a similar kind of thing *(shows figure 4.3 (p11))*. We've also got some things when you click on these icons the menus will come up. So it's quite similar to the one above - what do you think?

Ben: I think versus the one just before, with this one it doesn't look as convoluted as there aren't two panels in each side. Well, the one on the right is just a small panel but the one before had actual panels on either side. I think I prefer this, and having the top bar might make it a bit more familiar for people because you've got the File top-left, things like that. I think that would be helpful for people getting into it.

Sam: Yeah. What about this one? *(shows figure 4.4 (p11))*

Ben: Is that command-line?

Sam: Yeah it is, but don't worry too much about the content of the panels, just more where they are.

Ben: Ok, sure.

Sam: So you've got the simulation model is right at the front, then a slim menu down the side and more bulkier menus down the bottom. What are your thoughts on this?

Ben: I think I quite like that actually - it makes the thing you're actually interested in the biggest part which is obviously a lot more helpful, but I think when it comes to changing the settings it might be a bit more complicated because you'll have a small window. I think that one or the first one [figure 4.1 (p10)] is probably my favourite.

Sam: Ok, and then this is literally just taken from Microsoft Word *(shows figure 4.5 (p11))* but what do you think of the, like, just a panel at the top in sort of Microsoft Office products?

Ben: Only the panel on the top?

Sam: Just the top panel, and all the menus are up there.

Ben: I mean, because it's quite thin, I think you wouldn't fit too many buttons.

Sam: Yeah, and if you've got sliders and things that are bigger…

Ben: Yeah you'd only fit like 4 sliders up there. I think it's quite simple, but you need more on the side I think.

Sam: Yeah. So out of these 5, which would you say are your favourites?

Ben: I'd say the first one. [Figure 4.1 (p10)]

Sam: Ok, what would you say is good about it - what can I replicate?

Ben: I think you've got a large bit, over half, is taken up by the simulation, but you've also got plenty of space for the other settings that you need. I quite like the logos. [referring to the icons on the side panel] That makes it a bit easier - the other ones don't really have logos, they've just got words. Logos kind of help with a bit of flow. I quite like the dropdown, and I think it's a lot more simple.

Sam: What can you think of that you'd improve about it? You said about perhaps making this [referring to the space for the simulation] bigger...

Ben: Yeah, maybe slightly bigger. But, let me think, maybe...

Sam: Do you want to have a look at this one [figure 4.4 (p11)] that was your second favourite. What does this do that's good?

Ben: I think the simulation part is the biggest part. I think I quite like a big simulation and simple boxes on the left-hand side clearly split it up. I don't really know what else to improve about that one.

The UI should mostly be modelled on figure 4.1 (p10), with the large flow visualisation section as seen in 4.4 (p11).

Sam: Yeah, that very informative - that's fine. Great. So just a couple more questions, what sort of hardware should the simulation be able to run on?

Ben: I mean, I'd like to be able to use GPU acceleration.

Sam: Right that's great. Also, would you like me to use CUDA or OpenCL?

Ben: I have a NVidia GPU at home, but I'd like to be able to run it at school.

The simulation should be able to run on NVidia GPUs (via CUDA), and be able to run in a CPU-only mode to support school computers.

Sam: Ok. What are your thoughts on the balance between time to execute versus the accuracy of it - so, which is more important, do you want it to take longer and be more accurate or do you want it to be quick but be less accurate.

Ben: Probably being quick, because you're just iterating. The main thing of this is perfecting what you're making, so I think if you can do it quickly which is what iterating really is.

Sam: Do you want the option - to be able to change how long it takes?

Ben: Probably not, we don't need it super accurate, like professional levels of accurate, it's kind of just a comparison [between designs]

Sam: So what counts as quick then? How long would you expect from when you press go to it being a simulation?

Ben: Maybe a maximum of 10 seconds. It would probably be a bit slower on the school computers but yeah maybe 10 seconds on like a decent computer.

The simulation should be able to stabilise within 10 seconds on a computer with a sufficiently powerful GPU.

Sam: Fine. We've sort of already discussed this - what summary quantities would you like to be calculated, so some thoughts that I came up with were overall drag, we discussed that you wanted that, or stress points. Would that be useful to be able to see where is under the most stress from the aerodynamics?

Ben: Yeah I think so, because then we can, if we're fixing on a new front of our car, we want to be able to know where is under stress. Yeah, that would be helpful too. But being given like an overall value you can compare to everything else would be good.

Summary calculated statistics will be drag coefficient and, as an extension project, stress points will be able to be calculated.

Sam: So some things that are specific to the Greenpower team - what do you think are some issues or considerations specific to the program's use with the Greenpower team? You've said about it being easy to use...

Ben: I think we've only used one [CFD program] and it was on Eammon's iPad, but speed [to execute] as we only have 2 hours in the afternoon. Most of the time we want to be able to be tinkering with the [car] and not doing software bits I guess.

Sam: Alright, well that's been very useful thank you Ben - that's all.

(End of interview)

5.3 Objectives

Key objectives identified for the project are as follows:

1. The program will be able to calculate drag coefficient, accurate to at least 1 significant figure.
2. Editable parameters shall include fluid speed (from 0 to 40mph) and obstacle material friction. Fluid speed will be able to be varied during the running of the simulation.
3. A popup before the simulation begins shall be used to change parameters (fluid speed and material friction).
4. The program shall be able to read in files for obstacles. These will be in the format of an image with bit depth 1, where 1 represents fluid and 0 represents obstacle.
5. The program shall allow real-time obstacle editing based on a click-and-drag-node system.
6. Units (such as length, speed, time, density and viscosity) used throughout the program shall be dynamic, and editable through a settings panel.
7. Flow visualisation shall be based on figure 4.9 (p13), using flow lines to differentiate between laminar and turbulent flow and a colour gradient from blue to pink to represent field value.
8. The UI shall have panels on the left using icons to differentiate between menus, based on figure 4.1 (p10).
9. The flow visualisation shall take up at least half of the UI by width by default.
10. The UI shall be able to be resized from full-screen to taking up only a portion of the screen without clipping (where parts of the UI are obscured).
11. The UI shall use Object-Oriented programming practices and the MVVM design pattern.
12. The backend shall use the Navier-Stokes equations to form a model of fluid flow.
13. The simulation should be able to run on NVidia GPUs.
14. The backend must make use of parallel computation and multithreading.
15. The simulation should stabilise within 10 seconds on a sufficiently powerful GPU (where sufficiently powerful means a GPU more powerful than a GTX 1060).
16. The simulation will be able to run in a CPU-only mode, to facilitate use of the school computers. This may not stabilise within the 10-second time limit.

Part II

Design Stage

Chapter 6

Overall Project Structure

6.1 Overall Project Structure

6.1.1 File Maker

The File Maker (see Section 8.1.1 (p28)) is an optional first part that the user interacts with. A CLI program takes mathematical constraints (as discussed in the project proposal), converts them to a format that the program can parse, and creates the file.

6.1.2 User Interface

The User Interface (see Section 7 (p24)) is the part of the program the user interacts with. It contains all the sliders for changing parameters, an interface for loading a file, and contains the part of the project which shows the simulation data (the visualisation).

6.1.3 Visualisation

The Visualisation (see Section 9 (p34)) refers to the part of the project that turns the data output by the program into user-friendly colours and flow lines. It uses the graphics library OpenGL to change the colour of the individual pixels on the part of the screen where it resides, and to draw the stream lines.

6.1.4 Backend

The Backend is the part of the project that performs all of the computations (see Section 10 (p38)). It is commanded by the user interface via the backend-frontend interface (see 13 (p54)), and performs calculations on the CPU or, if the harware supports it, the GPU to maximise performance.

6.2 Project Structure Diagram

Figure 6.1 shows how each part of the project link together diagrammatically.

[1]The File Maker is run as a separate program to the rest, indicated using a dashed arrow rather than solid. The file it creates is saved into a storage location by the user and then loaded by the main program.

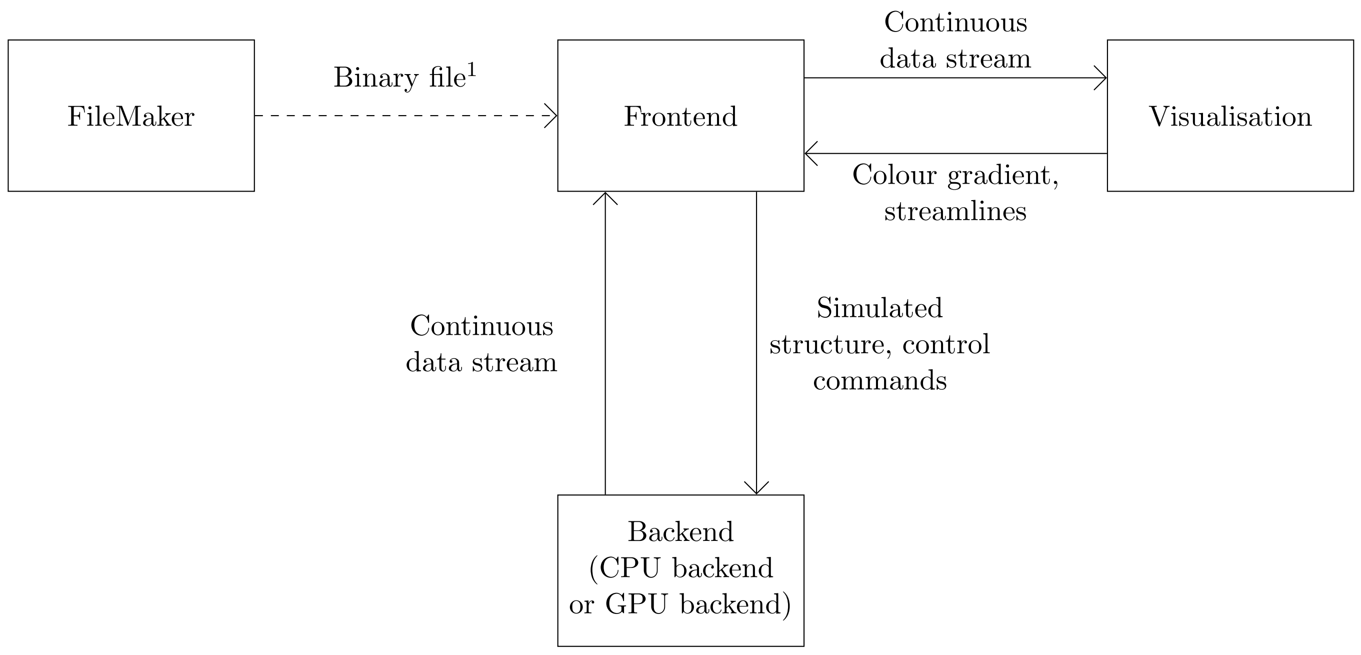

FileMaker
Binary file[1]
Frontend
Continuous data stream
Visualisation
Colour gradient, streamlines
Continuous data stream
Simulated structure, control commands
Backend (CPU backend or GPU backend)

Figure 6.1: Project structure diagram

Chapter 7

User Interface

7.1 User Interface Structure

The user interface, made using WPF, consists of 3 main screens. The user is initially greeted with the Config Screen, which contains some options for parameters including those that cannot be changed during the simulation. From this screen the user can navigate to a popup screen for advanced parameters, in accordance with objective 3. The main screen is the Simulation Screen, which loads the visualisation, starts the backend, and allows obstacles to be defined. Figure 7.2 (p26) represents this structure diagrammatically. Figure 7.1 shows prototypes of these three screens that were developed at a meeting with the Greenpower team.

7.2 UserControls

WPF provides `UserControls`, which are UI elements defined as one object to be used in multiple places throughout the program. Common groups of UI elements, like a slider with text box, are implemented as UserControls and then instantiated in many places in the UI. `UserControls` to be defined are: `ResizableCentredTextBox`, a text box with text parameter that centres and resizes itself to fill the parent container using a WPF `ViewBox`, and `SliderWithValue`, a slider with attached text box, as well as labels indicating minimum and maximum values of the slider.

7.3 Management of Parameters Throughout the Program

Parameters in the program are to be managed by a single class `ParameterHolder`. The `ParameterHolder` should contain all of the parameters that the user may specify. Dependency injection is used throughout the program to allow the same instance of `ParameterHolder` to be passed to the view models by the `App` class, so that all view models' changes affect the UI's state. Each view model will then make accessible a property, that the corresponding view will bind to and use in a slider or other method of input.

7.4 Management of Units Throughout the Program

In accordance with objective 6, units throughout the program must be dynamic. Similar to the `ParameterHolder` class, a `UnitHolder` class is defined to hold the current units in use in the program. It holds the current units for length, speed, time, density and viscosity. The units are defined via an inheritance tree from an abstract base class `Unit`, to abstract classes like `LengthUnit`, `SpeedUnit`, to the concrete child classes such as `Metre` and `Second`. A new unit class is instantiated and stored whenever the user selects it.

A `UserControl`, `UnitConversionPanel`, is defined which contains options to change units and can be used in multiple places in the program. Dropdown menus are used exclusively here, to change units individually or to change the unit system (e.g. metric or imperial) entirely.

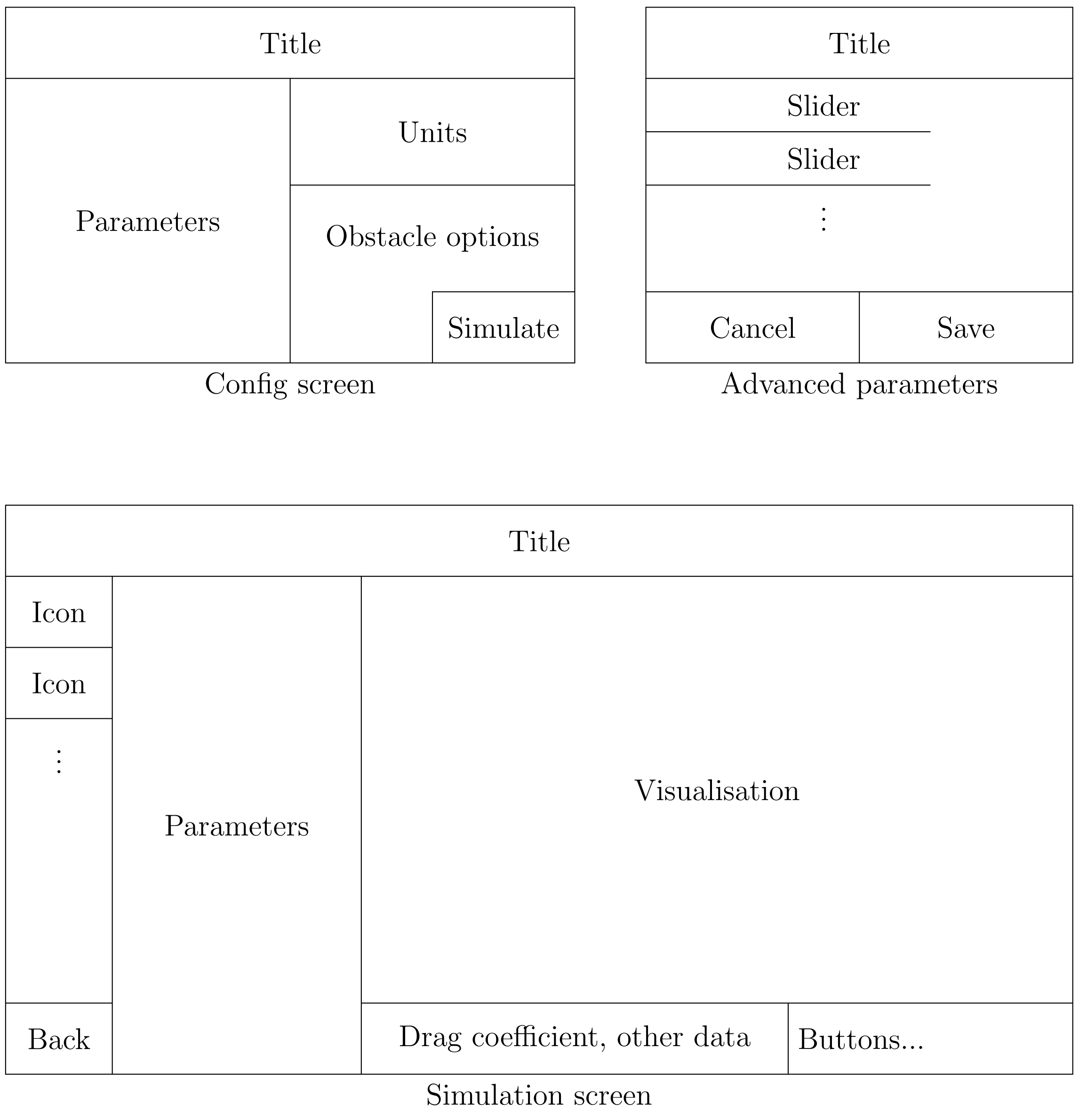


Figure 7.1: User interface prototype screens

7.5 Data Binding

Data binding is the primary method to transfer data between views and view models. The view model exposes a public property, and the view uses the XAML binding syntax to set variables relating to the user interface. For example, consider a checkbox for showing streamlines or not. This could be bound to a boolean property in the view model, which would set the corresponding property in the parameter holder and affect whether streamlines are shown.

7.5.1 Converters

A data binding can optionally use a converter, to allow properties of different data types to be bound together. Converters are classes that implement the `IValueConverter` interface. For example, conversion between polar and rectangular coordinates is handled by a converter class `RectangularToPolar`.

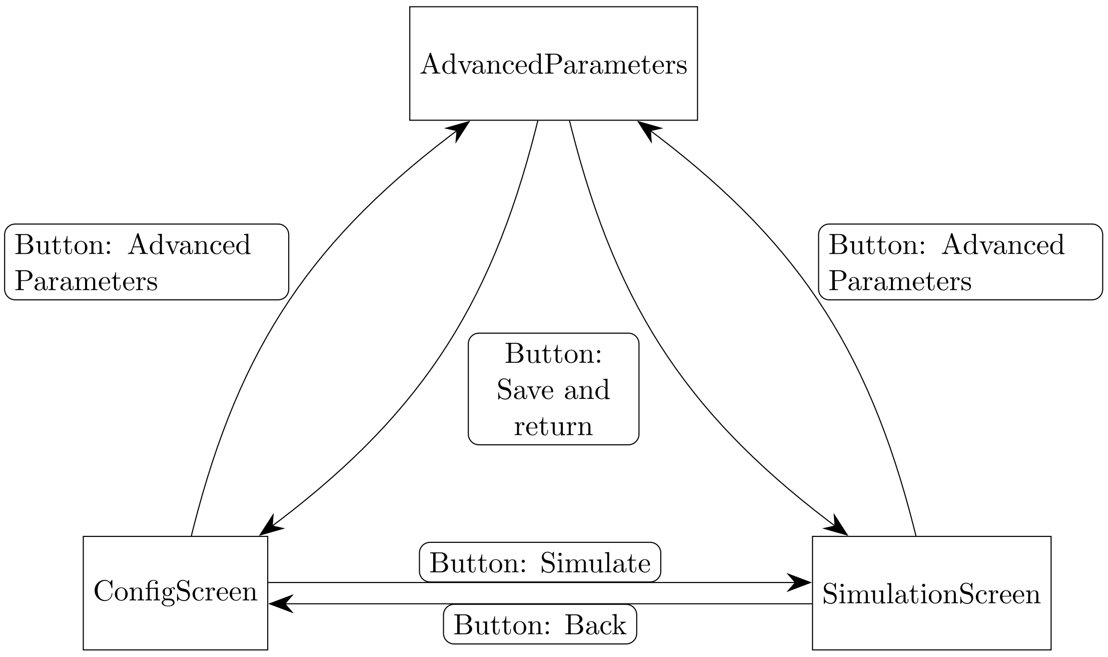


Figure 7.2: User interface structure diagram

7.6 Other Classes

7.6.1 Commands

In accordance with the MVVM design pattern prescribed by objective 11, click events are relayed to the ViewModel to perform the relevant task. In WPF, this is realised with `Commands`. These are simply classes that implement the `ICommand` interface. Commands are reusable, since the view model makes an instance of the command available to the view but does not own it, and can form part of inheritance hierarchies as normal classes. In the UI, classes that require data from the parent view inherit from the abstract `VMCommandBase`, which contains logic to handle a parent view model reference. When the view model instantiates any such class, it simply passes a reference to itself in the constructor. Commands that also require access to the parameter holder inherit from the abstract `ParameterCommandBase`, which itself inherits from `VMCommandBase`. This contains logic for passing the `ParameterHolder` to the command so it can modify it. Commands all exist inside a container class `Commands`, which is composed of all of the commands for the program.

7.6.2 Queues

The user interface has multiple use cases for different types of queues. For a moving average, data points are queued and dequeued to maintain a window of values to average, and parameters to send are added to a queue before they are sent. An abstract `Queue` base class is implemented, which contains methods such as enqueueing and dequeuing. Two classes, `CircularQueue` and `ResizableLinearQueue`, override these methods to provide their own implementation.

7.7 Class Diagram

The below diagram represents all of the classes to be defined, and how they interact with each other. There are many classes, and so classes have been compressed to just their identifier.

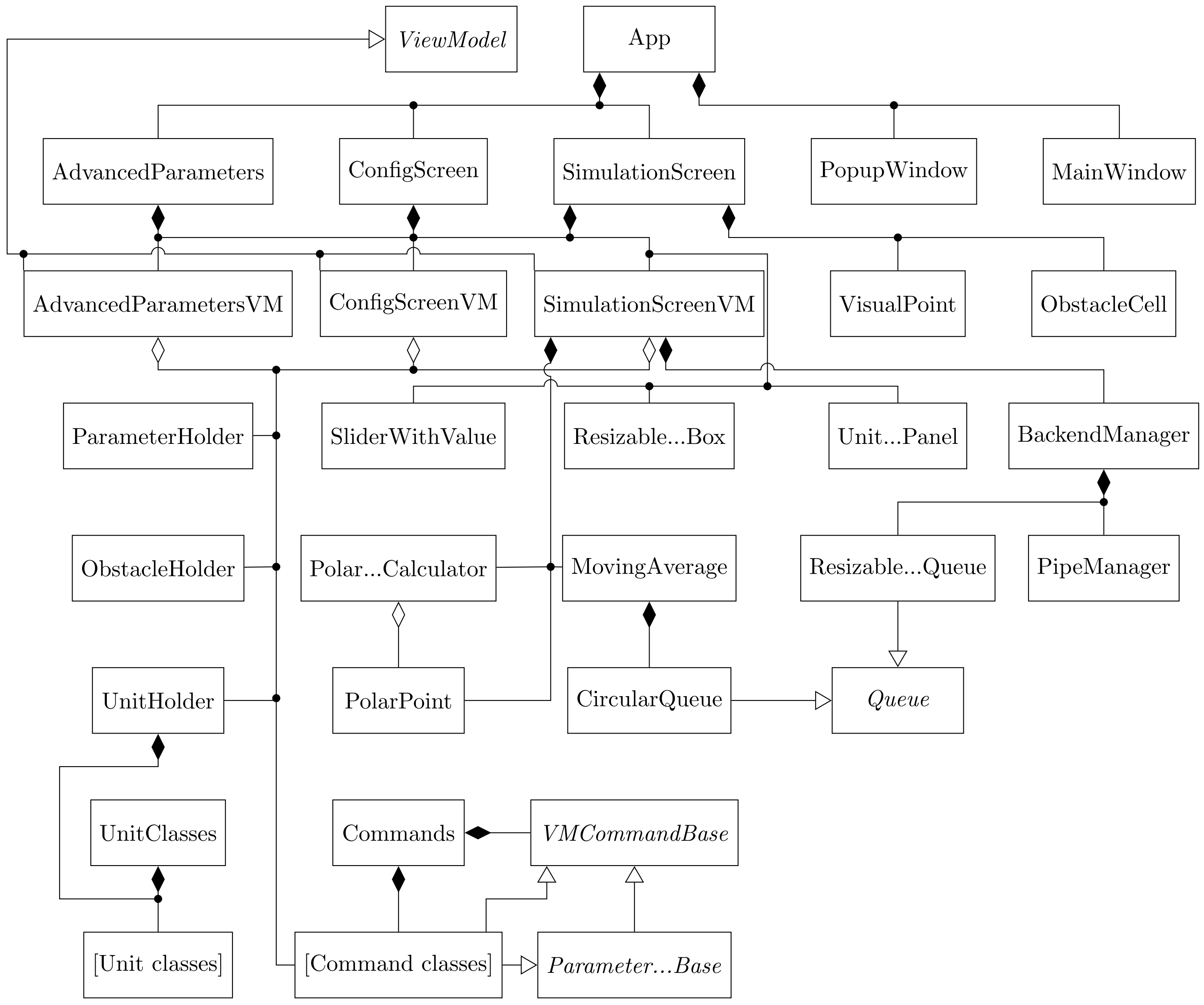


Figure 7.3: Frontend class diagram

Chapter 8

Definition of Obstacles

The definition of obstacles will be possible in 2 ways. First is reading in a file and using that as the simulation object. Second is using the click-and-drag node system referenced in objective 5.

8.1 Reading In of File

In accordance with objective 4, the program must be able to read in a file. The file type used by the system will be a binary file as this is what is used by the system internally. The size in the x and y directions of the simulation domain are given in the first 8 bytes, and the bits thereafter represent fluid (1) or obstacle (0). This is the same format as the compressed obstacle data as described in section 13.3.3 (p55), and it will be read in using an adapted version of Algorithm 13.1 (p55). It will also contain the dimensions of the simulation domain to use (iMax and jMax) to allow the data to be interpreted correctly. In the UI, an `OpenFileDialog` will be used to select the obstacle file. The `OpenFileDialog` uses the default windows dialog for file opening, so will be easy to use (as users will be familiar with this dialog).

Once the user has selected a file (or elected not to use a file and to instead use the click-and-drag node system), this data is set in a class `ObstacleHolder`, which is passed to all the screens. When SimulationScreen is instantiated, therefore, it can either read the obstacles from the file and send this to the backend, or allow the user to define their own obstacle.

8.1.1 Constraint-Defined File Creation

As a binary file is quite unintuitive to work with, a command-line interface program is to be created to enable the use of constraints to define obstacles. Application of constraints is a simple task, accomplished by looping through all of the integer points in the obstacle domain and setting them to fluid or obstacle depending on whether the point satisfies all of the constraints. The difficulty arises from parsing user input into a format that can be easily evaluated by the program. This is accomplished by first converting the constraints into postfix notation (see definition (p75)), using a tokeniser (see Algorithm 8.1 (p29)) and the Shunting Yard algorithm described in Algorithm 8.2 (p30), then evaluating the postfix expression repeatedly using a stack-based evaluation algorithm (see Algorithm 8.3 (p31)).

8.1.2 Object-Oriented Structure

In order to use OOP effectively, static classes are preferred to namespace-level methods. This allows methods to be grouped under a common functionality, if not making up an object. A `Constraint` class is defined, that uses the C# `Func` structure, and from this inherits an `RPNConstraint` class which overrides methods to allow it to work with reverse-polish strings rather than `Func`s. Figure 8.1 (p32) is a class diagram that represents this structure.

```
Input  : A string input.
Output: An array of strings tokens.
while not at the end of the string do
    Set the current character currentChar
    if currentChar is a space then
        Skip to the next iteration
    end
    if currentChar is a digit then
        if lastChar is a digit or decimal point then
            Append currentChar to the last string in tokens
        else
            Add currentChar as a separate string to tokens
        end
    else if currentChar is a decimal point then
        if lastChar is a digit then
            Append currentChar to the last string in tokens
        else
            /* A decimal point followed by numbers implies a 0 before the point.
            */
            Add "0." as a separate string to tokens
        end
    else if currentChar is x or y then
        if lastChar is neither an operator nor a left bracket then
            Add a multiplication sign to tokens
            /* Turns e.g. $2x$ into $2 \times x$                              */
        end
        Add currentChar to tokens
    else if currentChar is an operator or bracket then
        Add currentChar to tokens
    else if currentChar is the start of a mathematical function then
        /* Mathematical functions could be sin or cos, for example.           */
        if lastChar is neither an operator nor a left bracket then
            Add a multiplication sign to tokens
        end
        Add the function to tokens
    else
        /* input was in an invalid format.                                    */
        Raise an exception.
    end
    Set lastChar to currentChar
end
```

Algorithm 8.1: Tokenisation algorithm

```
Input  : An tokenised string array tokens.
Output: A postfix representation output.
Initialise a stack operatorStack and a list output
while there are tokens to be read do
    Set the current token currentToken
    if currentToken is a number then
        Add currentToken to output
    else if currentToken is a function then
        Push currentToken onto operatorStack
    else if currentToken is $x$ or $y$ then
        Add currentToken to output
    else if currentToken is an operator then
        if operatorStack is empty then
            Push currentToken onto operatorStack
        else
            while there exists a higher-precedence operator at the top of the stack, or a
              same-precendence operator at the top of the stack and currentToken is a
              left-associative operator do
                Pop an item from the top of operatorStack and add it to output
            end
            Push currentToken onto operatorStack
        end
    else if currentToken is a left bracket then
        Push currentToken onto operatorStack
    else if currentToken is a right bracket then
        while the item at the top of the stack is not a left bracket do
            Pop an item from the top of operatorStack and add it to output
        end
        Pop the left bracket from operatorStack and discard it
        if There is a function at the top of operatorStack then
            Pop it from the top of operatorStack and add it to output
        end
    else
        /* The expression is not a valid infix expression                    */
        Raise an exception
    end
end
while operatorStack is not empty do
    Pop an item from the top of operatorStack and add it to output
end
```

Algorithm 8.2: Shunting Yard algorithm

```
Input  : A tokenised postfix expression input, along with values for any variables.
Output: The result of the calculation encoded by input.

while there are tokens to be read do
    Set token to be the current token in input;
    if token is an operator or function then
        if token requires 2 operands then
            Pop an item from evaluationStack and store it in operand2;
            Pop another item from evaluationStack and store it in operand1;
        else
            /* token requires one operand                                    */
            Pop an item from evaluationStack and store it in operand1;
        end
        Perform the operation encoded by token on operand1 and, if applicable, operand2, and
          push the result onto evaluationStack;
    else
        /* token is a number or variable                                     */
        if token is a number then
            Push token onto evaluationStack;
        else
            /* token is a variable                                           */
            Push the value of token onto evaluationStack;
        end
    end
    Pop the result of the calculation from evaluationStack;
end
```

Algorithm 8.3: Stack-based postfix evaluation algorithm

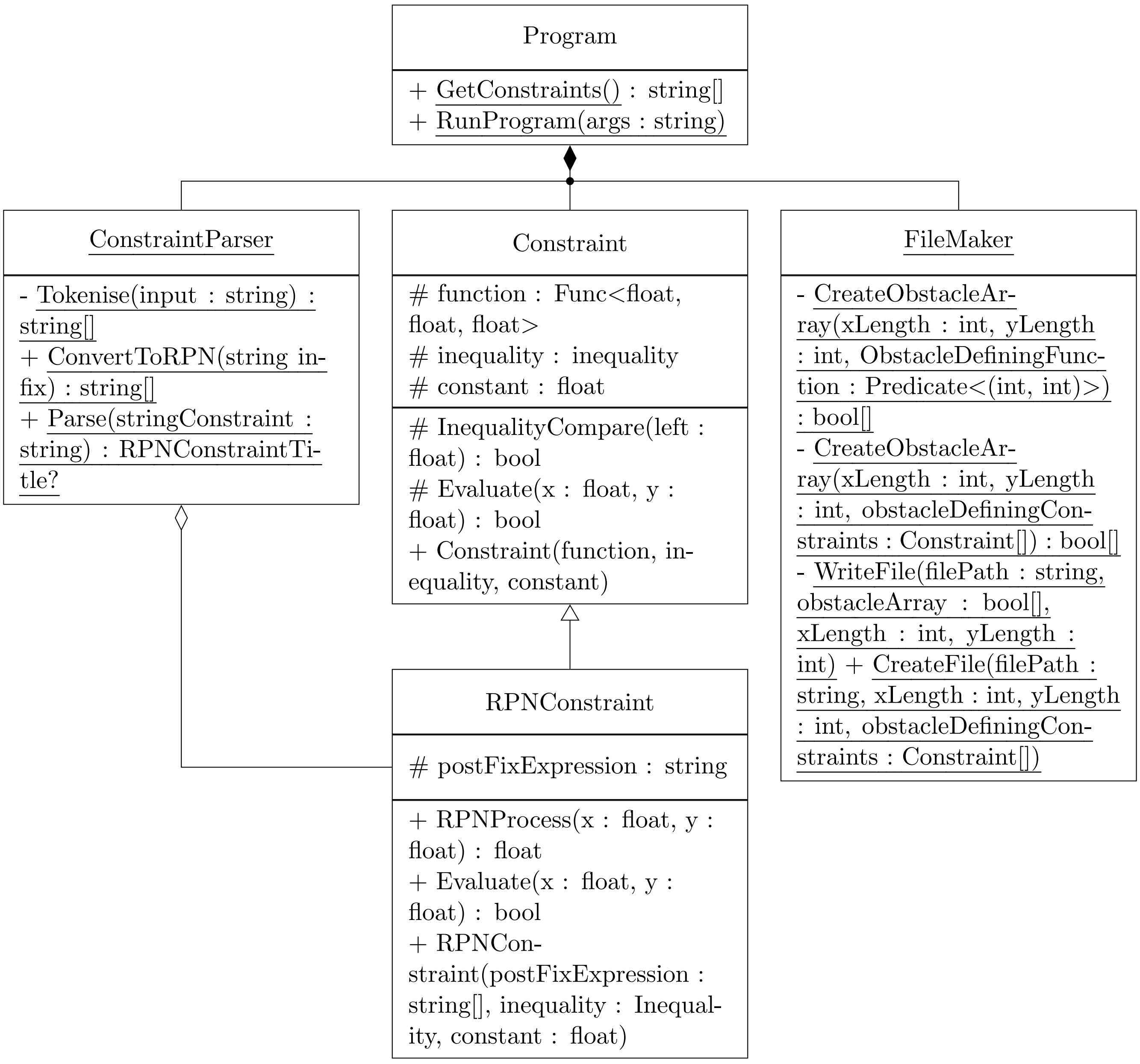

Program
+ GetConstraints() : string[]
+ RunProgram(args : string)
ConstraintParser
- Tokenise(input : string) : string[]
+ ConvertToRPN(string infix) : string[]
+ Parse(stringConstraint : string) : RPNConstraintTitle?
Constraint
# function : Func<float, float, float>
# inequality : inequality
# constant : float
# InequalityCompare(left : float) : bool
# Evaluate(x : float, y : float) : bool
+ Constraint(function, inequality, constant)
FileMaker
- CreateObstacleArray(xLength : int, yLength : int, ObstacleDefiningFunction : Predicate<(int, int)>) : bool[]
- CreateObstacleArray(xLength : int, yLength : int, obstacleDefiningConstraints : Constraint[]) : bool[]
- WriteFile(filePath : string, obstacleArray : bool[], xLength : int, yLength : int) + CreateFile(filePath : string, xLength : int, yLength : int, obstacleDefiningConstraints : Constraint[])
RPNConstraint
# postFixExpression : string
+ RPNProcess(x : float, y : float) : float
+ Evaluate(x : float, y : float) : bool
+ RPNConstraint(postFixExpression : string[], inequality : Inequality, constant : float)

Figure 8.1: FileMaker class diagram

8.2 Click-and-Drag Node System

Users will be able to draw their own obstacles using a system based on cubic splines (see definition (p74)), as described in [3]. Cubic polynomials are most suited to this task as they are the simplest type of polynomial which can form a point of inflection (where the curvature of the function changes from concave to convex). Splines are created, in this implementation, through the creation of multiple different cubic functions such that one cubic function exists between any two control points. Polar coordinates (see definition (p75)) are also used, so that the final polynomial can wrap around to the first control point and the entire spline can have a defined centre. There are therefore as many cubics as there are control points. Each polynomial between control points is then subject to 4 constrains. For the nth spline between the nth and $(n + 1)$th control points:

- It must pass through the nth control point.
- It must pass through the $(n + 1)$th control point.
- Its gradient at the $(n + 1)$th control point must match the gradient of the $(n + 1)$th cubic.
- Its curvature (second derivative) at the $(n+1)$th control point must match the curvature of the $(n + 1)$th cubic.

Therefore, for each of the individual cubic functions, there exist 4 constraints. As a result, the 4 coefficients of the cubic may be determined using the 4 equations that result from the constraints. In the implementation, this forms a linear system of equations equal to 4 times the number of cubics. When the number of control points is small (in the region of 1-10), this is able to be solved via a method of LU decomposition. The constraints are converted to equations using the coordinates of the user-defined control points. These equations are in terms of the coefficients of each cubic. These equations are then placed into a square matrix. Through inversion of this matrix via LU decomposition, a distinct solution is able to be found for the coefficients of the different cubics. A class `PolarSplineCalculator` performs these calculations to obtain the resulting output function.

8.2.1 Definition of Control Points

Control points must be able to be user-defined. Therefore control points are instances of a `VisualPoint` class that is able to be moved via a click-and-drag system, created with a left click and deleted with a right click. The position of these points may then be transferred to the spline calculator whenever the points change. As a bonus, the centre point of the obstacle is also a `VisualPoint`, but removes the left- and right-click functionalities.

8.2.2 Use of Polar Coordinates

As mentioned above, the spline calculation uses polar coordinates. However, the canvas on which the visualisation and obstacles are drawn uses rectangular coordinates. Therefore, points must be converted to polar before they can be used in calculation. A converter class is therefore defined which takes a rectangular point and the obstacle centre as input to return a polar point whose pole is at the obstacle centre.

Chapter 9

Flow Visualisation

9.1 Visualisation Methods in CFD

9.1.1 Visualising Function Output

Commonly, a function may take a 2D coordinate as its input and give a single number as its output. The data output by the backends will be in this format. For these functions, there exist some standard visualisation methods. One example is the *graph* of the function, which is drawn by lifting each point on the (x, y) plane such that its height is the output of the function. Figure 9.1 demonstrates this for an arbitrary function.

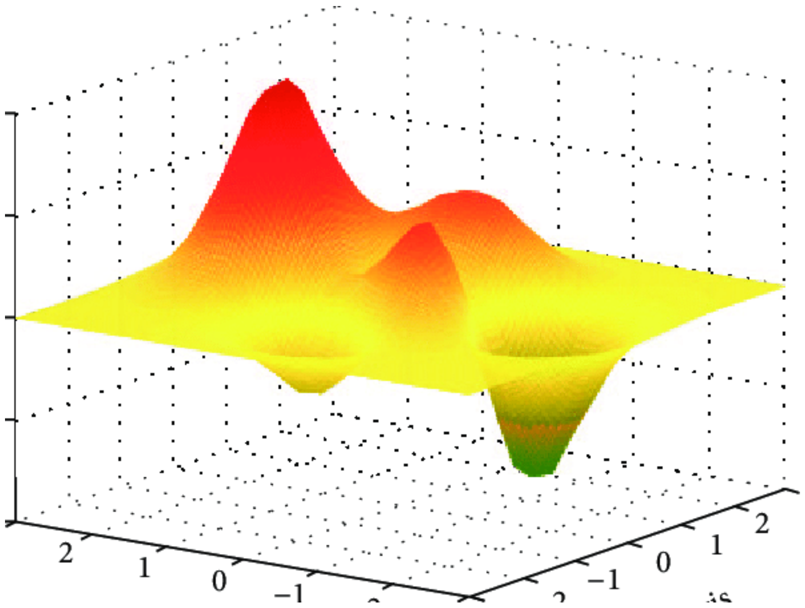


Figure 9.1: Graph visualisation of a 2D function, including colouring

Another technique is the use of *level sets*, which are the points at which a function takes a specific value. For continuous functions, plotting these level sets gives contour lines, as shown in figure 9.2.

The final technique to discuss is the use of colouring a 2D space according to a spectrum. Greyscale can be used, although more often a different spectrum (such as the red-yellow-green in figure 9.1) is used that is more appealing to the eye and thus easier to understand.

In accordance with objective 7, the visualisation will use the colouring technique to visualise the value of the pressure at each grid point.

9.1.2 Particle Tracing and Streaklines

Often, it is desirable to analyse flow patterns in terms of the movement of individual particles. There are 2 main methods for doing this:

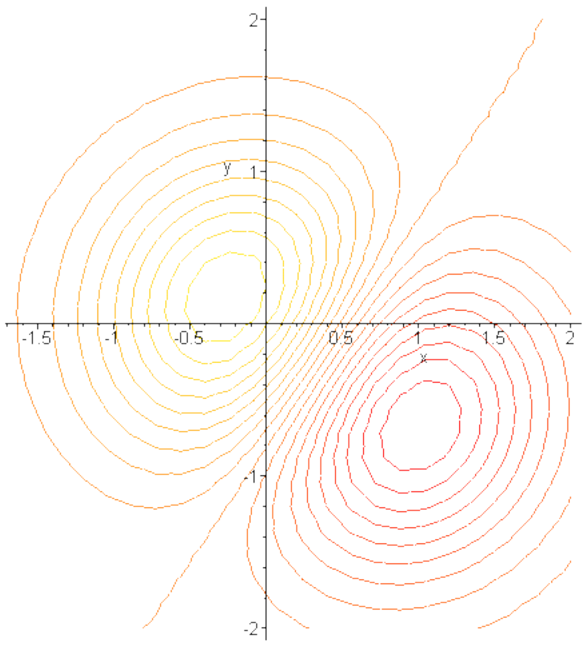


Figure 9.2: Contour plot of a 2D function, including colouring

1. **Pathlines:** The positions of one particle through successive timesteps are shown.
2. **Streaklines:** Particles are injected into the flow at a fixed location in short, regular time intervals, and the positions of the particles belonging to the same injection are shown.

For this to work, we require the ability to precisely define horizontal and vertical components of velocity at every point on the grid (we need to turn our discrete grid back into a continuous domain). To do this, we use interpolation.

Consider a point in the discrete grid (x, y). We must first find the cell in which (x, y) lies. These can be determined by using floor operations, to give the coordinates of the upper-right corner of the containing cell, (i, j). We may then use a *bilinear interpolation* (linear interpolation in 2 dimensions) to find an approximation of the values of the velocity components at (x, y). Again, as the size of our grid cells decrease, the approximation will approach the true value.

The streaklines and particle lines can then be calculated with monitoring the motion of a set of particle through the fluid.

9.1.3 Streamlines

Streamlines are defined as a line whose tangent is parallel to the velocity vector for all of its points at one time. A less precise definition could be that streamlines show the movement of a particle with no mass (and therefore no momentum) as it moves through the field. This method of visualisation will be the principal method used as required by objective 7.

The stream function, $\psi(x, y)$ is therefore introduced and defined by the following equation:

$$\frac{\partial \psi}{\partial y} = u \tag{9.1}$$

A discrete version may be found by using a backward difference method:

$$\left[\frac{\partial \psi}{\partial y}\right]_{i,j} = \frac{\psi_{i,j} - \psi_{i,j-1}}{\delta y} \tag{9.2}$$

And, setting $\psi = 0$ for all bottom cells (where $j = 0$), a simple iterative formula can be used.

$$\psi_{i,j} = \psi_{i,j-1} + u_{i,j}\delta x \tag{9.3}$$

The relation between the stream function and streamlines is that streamlines are the contour lines of the stream function. The drawing of such lines is described in Algorithm 9.1 (p37).

9.2 Visualisation Design

9.2.1 Colour Gradient

For the colour gradient, a simple blue-purple colour gradient was chosen. The colour gradient can be seen in figure 9.3. Due to its simplicity, this colour gradient can be represented by a single colour vector.

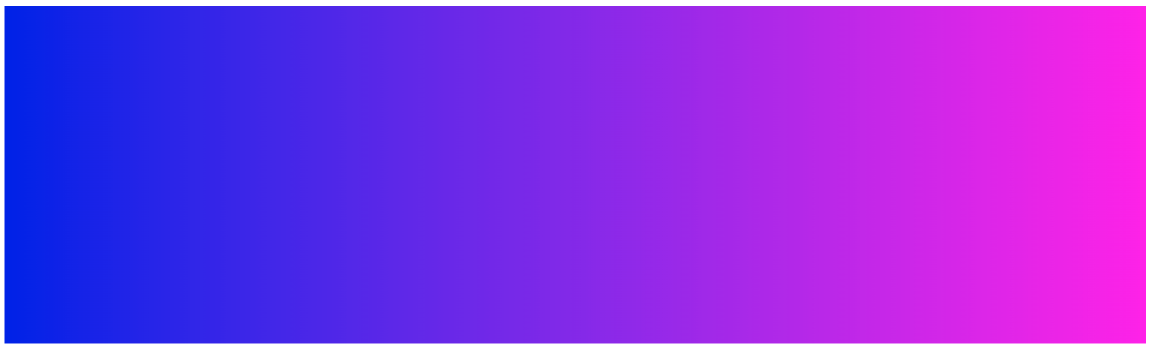


Figure 9.3: The colour scale to use for the visualisation

If the relative value of a field (between 0 and 1) is x, then the RGB colour vector is $(256x, 40, 248)$.

9.2.2 Contour Drawing

Drawing of the streamlines is a more complex task because it requires contour lines of the stream function. Algorithm 9.1 (p37) describes the algorithm for calculation of contour lines of a function, with given spacing and tolerance.

```
Input  : A 2D array streamFunction to draw the contour of and its width and height, the
         contour spacing multiplier spacingMultiplier and tolerance contourTolerance, and the
         value to append to the indices list to start the next line nextLineValue.
Output: An array indices of the indices of each contour line.
levelSets ← new 2D array of integers
for $i \leftarrow 0$ to height do
    streamFunctionValue ← streamFunction[$i$]
    if streamFunctionValue $= 0$ then
        continue
    end
    if streamFunctionValue within contourTolerance of a multiple of spacingMultiplier then
        levelSet ← Round(streamFunctionValue / spacingMultiplier)
        while levelSet $\geq$ length of levelSets do
            Add a new row to levelSets
        end
        Add $i$ to levelSets[levelSet]
    end
end
indices ← new 2D array of integers
for $i \leftarrow 1$ to length of levelSets do
    if length of levelSets[$i$] $= 0$ then
        continue
    end
    currentHeight ← levelSets[$i$][0]
    targetValue ← $i \times$ spacingMultiplier
    for $j \leftarrow 1$ to width $- 1$ do
        streamFunctionValue ← streamFunction[$j$][currentHeight]
        if streamFunctionValue within contourTolerance of targetValue then
            Add index of streamFunctionValue to levelSets[$i$]
        else if streamFunctionValue > targetValue then
            while currentHeight $> 0$ and targetValue − streamFunctionValue <
             contourTolerance do
                currentHeight ← currentHeight $-1$
                streamFunctionValue ← streamFunction[$j$][currentHeight]
            end
            if streamFunctionValue within contourTolerance of targetValue then
                Add index of streamFunctionValue to levelSets[$i$]
            end
        else
            while currentHeight < height $-1$ and streamFunctionValue ¡ targetValue do
                currentHeight ← currentHeight $+1$
                streamFunctionValue ← streamFunction[$j$][currentHeight]
            end
            if streamFunctionValue within contourTolerance of targetValue then
                Add index of streamFunctionValue to levelSets[$i$]
            end
        end
    end
    Add the contents of levelSets[$i$] to indices
    Add nextLineValue to indices
end
```

Algorithm 9.1: Algorithm to draw contour lines for a field.

Chapter 10

Discretisation and Numerical Solution

10.1 Introduction

The following 2 sections describe how to convert the Navier-Stokes equations (the principal model for fluid flow) into algorithms that can be run in parallel on a multi-core processor. The current section describes how to arrive at sequential algorithms for the fluid flow model, and the following section describes how these sequential algorithms may be adapted for parallel processing.

In numerical solutions, discretisation is the process of translating a continuous domain (a domain in which functions can be evaluated at every point) to a discrete one (in which functions can only be evaluated at specific, evenly-spaced values). This allows the differential equations that describe the flow precisely to be converted to algebraic equations that approximate the flow, but can be solved.

10.2 Discretising Space

10.2.1 One-Dimensional Example

To explain the basic idea behind the discretisation algorithm, we consider the 1-dimensional case. The domain to be consider can take any value in the range from 0 to a. This can be written as $0 \leq x \leq a$. If we choose a number of points to approximate this domain, i_{max} points[1], then the difference between the points, δx, is as follows.

$$\delta x = \frac{a}{i_{max}} \tag{10.1}$$

This is known as the *finite difference method*, as the difference between the discrete points at which variables are evaluated is a known, finite, constant.

Therefore, the derivative between each of the points can be calculated. The derivative, from first principles, is defined as the difference in the output of a function, $f(x) = y$, divided by the corresponding change in input to a function. This, notationally, can be written as $\frac{dy}{dx}$, where dy and dx represent the small change in the output and the small change in the input, respectively. In the continuous case, one considers what happens to this derivative as dy and dx get close to 0, called considering the *limit as dx tends to 0.* However, in the discretisation we do not need to consider what happens when dx tends to 0 - we instead consider the difference between each of the finite points we define. Of course, as our number of points, i_{max}, increases and therefore difference, δx, decreases, our approximation becomes more accurate.

The specific mathematics involved in this discretisation is more complex than the simplification above, and is outside the scope of this document. They are more fully described in [1, Chapter 3].

10.2.2 Extending to the 2-Dimensional Case

The 2-dimensional case has a domain from 0 to a in the x direction, and from 0 to b in the y direction. We still have our i_{max} divisions in the x direction giving a spacing of δx, but now include j_{max} divisions

[1] In computer science terms, i can be thought of as the index of the relevant point in space, so i_{max} is the upper bound for i - the maximum number of points.

in the y direction with a spacing of:

$$\delta y = \frac{b}{j_{max}} \tag{10.2}$$

10.2.3 Staggered Grid Method

The discretisation of space that will be used in the program is called a staggered grid. This means that the different grids used for the relevant variables (pressure and velocity) are not the same. Instead, the continuous 2D space is divided into cells, each of size $\delta x \times \delta y$ (see figure 10.1). Then velocity is evaluated at the centre of the boundaries of the cells (which can be thought of as the flow in and out of the cell) and pressure is evaluated at the centre of each cell. This gives rise to what may be thought of as 3 separate grids (one for each of pressure, P, vertical velocity, v and horizontal velocity, u).

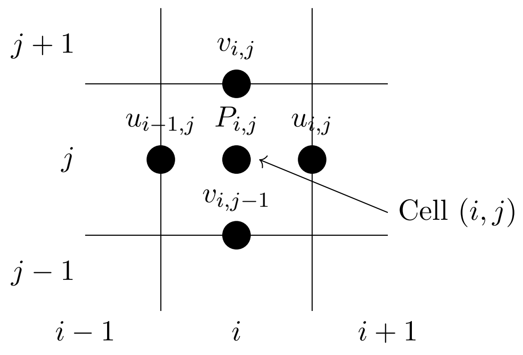


Figure 10.1: Staggered grid discretisation method

10.2.4 The Navier-Stokes Equations

This section contains lots of mathematical symbols. For explanation, see Appendix B (p76) - Definition of mathematical symbols. The Navier-Stokes equations, frequently alluded to above, are now properly defined below in their 2-dimensional forms.

Momentum equations, arising from conservation of momentum:

$$\frac{\partial u}{\partial t} + \frac{\partial p}{\partial x} = \frac{1}{Re}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) - \frac{\partial (u^2)}{\partial x} - \frac{\partial (uv)}{\partial y} + g_x \tag{10.3}$$

$$\frac{\partial v}{\partial t} + \frac{\partial p}{\partial y} = \frac{1}{Re}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) - \frac{\partial (uv)}{\partial x} - \frac{\partial (v^2)}{\partial y} + g_y \tag{10.4}$$

Continuity equation, arising from conservation of mass:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \tag{10.5}$$

10.3 Discretising Time

Similar to the discretisation in space, the discretisation in time works upon a finite difference method. We subdivide the time interval, $[0, t_{end}]$ into a number of timesteps and are able to create a finite difference, δt. Again, this allows for derivatives to be evaluated across the timestep rather than in a continuous case. In my implementation, the t_{end} does not exist, since the simulation will run until the user halts it. The timestep is also calculated using a timestep control function, based on the magnitudes of the velocities and the step sizes, δx and δy.

10.4 Numerical Solution Algorithm

10.4.1 Time-Stepping Loop

With initial values for u, v and $t = 0$, time is incremented by δt until halted. The time discretisation of the momentum equations yields the following:

$$u^{(n+1)} = u^{(n)} + \delta t \left[\frac{1}{Re} \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) - \frac{\partial (u^2)}{\partial x} - \frac{\partial (uv)}{\partial y} + g_x - \frac{\partial p}{\partial x} \right] \tag{10.6}$$

$$v^{(n+1)} = v^{(n)} + \delta t \left[\frac{1}{Re} \left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right) - \frac{\partial (uv)}{\partial x} - \frac{\partial (v^2)}{\partial y} + g_y - \frac{\partial p}{\partial y} \right] \tag{10.7}$$

We then introduce the abbreviations F and G, including all the spatial derivatives of velocity as follows:

$$F = u^{(n)} + \delta t \left[\frac{1}{Re} \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) - \frac{\partial (u^2)}{\partial x} - \frac{\partial (uv)}{\partial y} + g_x \right] \tag{10.8}$$

$$G = v^{(n)} + \delta t \left[\frac{1}{Re} \left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right) - \frac{\partial (uv)}{\partial x} - \frac{\partial (v^2)}{\partial y} + g_y \right] \tag{10.9}$$

To obtain the following, after time discretisation:

$$u^{(n+1)} = F^{(n)} - \delta t \frac{\partial p^{(n+1)}}{\partial x} \tag{10.10}$$

$$v^{(n+1)} = G^{(n)} - \delta t \frac{\partial p^{(n+1)}}{\partial y} \tag{10.11}$$

Through rearranging a substitution with 10.5, a more useful pressure equation can be found in the form of a Poisson equation. The explanation of this is out of the scope of this document, but the derivation is shown in [1].

A result of these equations is that the pressure field must be calculated before the corresponding velocity fields. A brief algorithm is given below.

For the $(n + 1)$st time step we must:

1. Compute $F^{(n)}$ and $G^{(n)}$ according to 10.8 and 10.9 from the velocities $u^{(n)}$ and $v^{(n)}$
2. Solve the equation for pressure to give $p^{(n+1)}$
3. Compute the new velocity field $u^{(n+1)}$ and $v^{(n+1)}$ using 10.10 and 10.11 with the $p^{(n+1)}$ values computed previously.

10.4.2 Discrete Momentum Equations

The discretisation of the momentum equations in space is the final step to allow for computation of them. Discretisations are as follows:

$$u_{i,j}^{(n+1)} = F_{i,j}^{(n)} - \frac{\delta t}{\delta x} \left(p_{i+1,j}^{(n+1)} - p_{i,j}^{(n+1)} \right) \tag{10.12}$$

$$v_{i,j}^{(n+1)} = G_{i,j}^{(n)} - \frac{\delta t}{\delta y} \left(p_{i,j+1}^{(n+1)} - p_{i,j}^{(n+1)} \right) \tag{10.13}$$

$$F_{i,j} = u_{i,j} + \delta t \left[\frac{1}{Re} \left(\left[\frac{\partial^2 u}{\partial x^2} \right]_{i,j} + \left[\frac{\partial^2 u}{\partial y^2} \right]_{i,j} \right) - \left[\frac{\partial (u^2)}{\partial x} \right]_{i,j} - \left[\frac{\partial (uv)}{\partial y} \right]_{i,j} + g_x \right] \tag{10.14}$$

$$G_{i,j} = v_{i,j} + \delta t \left[\frac{1}{Re} \left(\left[\frac{\partial^2 v}{\partial x^2} \right]_{i,j} + \left[\frac{\partial^2 v}{\partial y^2} \right]_{i,j} \right) - \left[\frac{\partial (uv)}{\partial x} \right]_{i,j} - \left[\frac{\partial (v^2)}{\partial y} \right]_{i,j} + g_y \right] \tag{10.15}$$

Here we see, especially in the definitions for F and G, that the derivatives are now defined for single cells rather than in the continuous plane.

10.4.3 Solving the Pressure Equation

Arguably the most difficult to understand and most computationally intense part of solving the Navier-Stokes equations, in this implementation, is dealing with the discrete pressure equations. One ends up with a linear system of equations with a number of equations and unknowns equal to the number of cells in the discretisation. This is far too large for normal analytical methods, and thus is solved by iteration. The algorithm used is known as successive over-relaxation (SOR) and is more fully described in [1]. A summary of the algorithm is below:

1. Copy pressure values for each non-fluid cell from the closest fluid cell
2. Perform a SOR iteration on the entire domain (this consists of a calculation for each cell based on its neighbours)
3. Calculate the residual (difference between the current pressure domain and the required solution)
4. If the residual is below a certain value (residual tolerance), stop
5. If the number of pressure iterations is equal to a maximum value, stop
6. Return to step 1

10.4.4 Timestep Restriction

The size of the timestep, δt, must be restricted to avoid nonphysical predictions. 3 restrictions are employed: no particle may travel further than the width of 1 cell in a timestep; no particle may travel further than the height of 1 cell in a timestep; and another, more complex restriction that relates the turbulence of the flow and the harmonic mean (a type of average) of the two timesteps. Combination of 3 conditions yields the following:

$$\delta t = \tau \min \left(\frac{Re}{2} \left(\frac{1}{\delta x^2} + \frac{1}{\delta y^2} \right)^{-1}, \frac{\delta x}{|u_{max}|}, \frac{\delta y}{|v_{max}|} \right) \tag{10.16}$$

10.5 Dealing with Obstacles

10.5.1 Boundary Conditions

With the discretisation used, often we find that cells depend on the values in their neighbours. For example, the pressure in cell (i, j) depends also on the pressure in the 4 cells $(i-1, j)$, $(i+1, j)$, $(i, j-1)$ and $(i, j+1)$. For most cells, the cells around it are cells in their own right with valid pressure values. However, we must consider the cells on the boundary for which at least one neighbour is not a valid cell.

We therefore consider the different quantities that a cell has: velocities in the x and y directions, F, G, pressure and, for the iterative solution of the pressure, the right-hand side of the pressure equation (RHS).

The easiest quantities for which to prescribe boundary conditions are the velocities. Along the 4 different edges, we can define different conditions to allow fluid to flow from left to right. Along the left edge, the horizontal velocities are set to some constant (an input velocity) where the vertical velocities are set to 0. This is an inflow condition. Along the right edge, velocities are copied from the cells one to the left to allow mass to flow out at this boundary - this is an outflow condition.

Along the top and bottom, we prescribe a free-slip condition where the vertical velocities are 0 and the horizontal velocities are copied from the cells on the inside of the boundary. This has the effect of modelling a frictionless wall.

For the values of F and G, we set them equal to the velocities at the boundary cells, which has the effect of allowing us to copy the pressures from the inside to the boundary cells (very useful during the SOR iterations to solve for the pressure). RHS does not need to be defined on the boundary cells, as the pressure algorithm only requires the RHS value for non-boundary cells.

10.5.2 Obstacles

When the user defines obstacles, these must be *embedded* into the simulation domain. We therefore use a similar algorithm to that for the boundary cells - pressure is copied from the closest fluid cell, F and G are copied from velocities, and RHS does not need to be defined. However, the velocities themselves are slightly more complex to define.

Using the velocities, we must simulate friction occurring between the surface of the object and the fluid. This is achieved via a partial-slip boundary condition.

The easiest way to simulate friction at a boundary is via a *no-slip* condition, where the velocity of the fluid relative to the boundary is equal to 0. This is a fair assumption, but does not allow for changes in the material friction, as is required as part of objective 2. The no-slip condition assumes first that there is no flow of fluid in the direction perpendicular to the boundary, so we set that velocity component to 0. It also assumes that the velocity parallel to the boundary is 0, however the boundary lies on the edge of 2 cells, as shown in figure 10.2.

In the discretisation, vertical velocities are defined at the midpoint of an edge, so v_o and v_f are defined, but v_b is not. However, it is v_b that must equal 0. Therefore, we use the average of v_o and v_f to define v_b:

$$v_b = \frac{v_f + v_o}{2} \tag{10.17}$$

Therefore, setting $v_b = 0$ requires that $v_o = -v_f$. If we want to be able to change the friction of the material, we may model this with a new parameter $\chi, 0 \leq \chi \leq 1$, such that $v_b = \chi v_f$. Using 10.17, it follows that $v_o = (2\chi - 1)v_f$. This gives a completely frictionless free-slip boundary condition when $\chi = 1$, and a no-slip (maximum friction) boundary condition when $\chi = 0$.

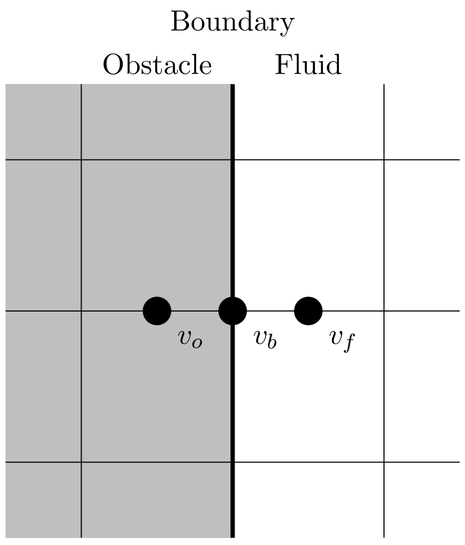


Figure 10.2: Position of discrete velocities on an object boundary

One final caveat is the existence of so-called *corner* cells. These are cells which are in the boundary region, but share 2 adjacent edges with fluid. For these, discretisation proceeds as normal except that the pressure value in this cell is defined as the average of the pressure in the 2 fluid cells it borders.

10.5.3 Implementation

The distinction between a fluid cell and a boundary cell may no longer be made simply by using their position. Instead, we use 2D array of bytes termed a flags array. Each bit of these bytes stores whether the cell itself and the cells in the cardinal directions are fluid cells (when the bit is set to 1) or obstacle cells (when the bit is set to 0). From bit 4 to bit 0, the cells referenced are: centre (the cell itself), north, east, south, and west (see Figure 10.3). For example, the cell with a bit representation 00011110 is a fluid cell (bit 4 is a 1) with a boundary on the western edge (this bit is a 0); similarly, a cell with bit representation 00001100 is a corner obstacle cell with boundaries on the northern and eastern edges.

			Self	North	East	South	West

Figure 10.3: Layout of bits in each byte of flags array

10.6 Calculation of Drag and Drag Coefficient

10.6.1 An Introduction to Drag Forces

A crucial element of a CFD simulation is the ability to calculate the drag on a structure. Drag, in the context of fluid dynamics, comes in 2 forms: pressure drag and viscous drag. Pressure drag, also called form drag or shape drag, is the drag caused by pressure differentials either side of the shape (a buildup of high-pressure at the front of the obstacle, and an area of low pressure at the back of the obstacle), and is loosely proportional to the frontal area of the body. Viscous drag, also called friction drag or skin drag, is the effect of frictional forces as the fluid flows around the object and interacts with the surface of the object. The proportion of each drag force that makes up the whole is based solely on areas: an object with a large frontal area but short length, like coin falling face-down, will have its drag caused mostly by pressure drag as its large frontal area displaces a larger volume of fluid; an object with a streamlined frontal area but with longer length, like an aeroplane wing, will have most of its drag caused by surface drag as the object slices through the fluid but is slowed by friction along its length.

10.6.2 Defining Equations

The important equations for this section are below.

$$c_d = \frac{2F_d}{\rho v_{in}^2 A_{\boldsymbol{i}}} \tag{10.18}$$

$$F_d = F_p + F_\mu \tag{10.19}$$

$$F_p = \int_S (p - p_0)(\hat{\boldsymbol{n}} \cdot \hat{\boldsymbol{\imath}})\, dS \tag{10.20}$$

$$F_\mu = \int_S (\hat{\boldsymbol{\tau}} \cdot \hat{\boldsymbol{\imath}})\tau_w\, dS \tag{10.21}$$

$$\tau_w = \mu \frac{\partial \boldsymbol{V}}{\partial d} \tag{10.22}$$

For definitions of symbols, refer to Appendix B (p76) - Explanation of Mathematical Symbols.

10.18 describes the relationship between the drag on a structure and the drag coefficient. Drag coefficient is an object property that allows for faster calculation of the structure drag at different velocities and fluid pressures. It is also an important comparative tool to describe the relative aerodynamic efficiency of an object.

10.19 simply describes the summative relationship between pressure drag and viscous drag with total drag.

10.20 is the characteristic equation for pressure drag - the integral over the surface of the object of pressure differences, in the direction of fluid flow.

10.21 is the characteristic equation for viscous drag - the integral over the surface of the object of shear stresses, perpendicular to the direction of fluid flow.

10.22 is the defining equation for wall shear stress - the viscosity multiplied by the change in fluid velocity with respect to the direction of the boundary. This may be more informally thought of as how much the air around the object is slowed down by frictional forces. $\boldsymbol{V}$ is the velocity field, incorporating both the u and v components.

10.6.3 Discretisation of Pressure Drag

In both the equations for pressure and viscous drag, the dot product is used to get the component of the relevant vector perpendicular or parallel to the overall fluid flow.

In the grid, pressure exists in the centre of cells. The surface to integrate over should therefore be the fluid cells that are next to the boundary layer. In the discretisation of 10.20, dS is one of the step sizes δx or δy, or $\frac{\delta x+\delta y}{2}$ for corner cells (the average step size). Also, $\hat{\boldsymbol{n}}$ is defined as the unit vector perpendicular to the line from the centres of the cells either side of it. For most cells, this is the direction normal to the cell's own boundary, but for corner cells this is the direction at $45°$ to the boundaries. Also, p_0 should be taken to be the average of the pressure at each of the 4 corners of the simulation domain, where the fluid flow is least impacted by the obstacle.

10.6.4 Discretisation of Viscous Drag

The discretisation in terms of wall shear stress is relatively simple, it is simply the sum of all the cells' wall shear stresses, multiplied by the dot product of the shear direction and fluid direction, and multiplied by the dS as described above. In fact, 10.21 could be rewritten as $\int_S (\boldsymbol{T_w} \cdot \hat{\boldsymbol{i}}) dS$, where $\boldsymbol{T_w}$ is the wall shear stress as a vector. However, wall shear stress is defined in the upper-right corner of a cell in the staggered grid method, so care must be taken to ensure that the correct cells are included for the surface integral. These would be the obstacle cells on the boundary for top and right edges, but the fluid cells on the boundary for bottom and left edges.

10.6.5 Discretisation of Wall Shear Stress

The discretisation of 10.22 is problematic because it requires directions that are not parallel to the coordinate axes. A discrete derivative shall be used here, with the start point at the top right of the boundary cell and the end point at the next cell in the direction of the normal to the surface $\hat{\boldsymbol{n}}$. For cells with one boundary edge, this is a simple case of calculating the velocity vector $\boldsymbol{V}$ from its components, and performing a discrete derivative calculation with the step size in the direction of the normal vector (also calculated from the stepsize components δx and δy). For cells with 2 boundary edges, the normal vector is not parallel to one of the axes, so interpolation is required. The value of $\boldsymbol{V}$ must be interpolated from the surrounding components, so that the derivative may be calculated using one step size. From this, the calculated derivative is multiplied by the viscosity μ, to give wall shear stress.

Chapter 11

Parallelisation and Use of a GPU

11.1 The Merits of Parallelisation

Parallelisation (performing multiple calculations at the same time) can vastly reduce the execution time of a program. This algorithm, in particular, lends itself well to parallelisation as operations on cells can each be performed concurrently by one thread of a GPU.

11.2 GPU Architecture

The way that GPUs perform calculations is very different even to the way that a multi-core CPU might perform calculations. A brief explanation is below.

11.2.1 GPU Terminology

- A thread is the smallest unit of execution on a GPU. It executes one instruction at a time.
- A warp is a collection of threads (often 32 threads) that share the same program counter and thus execute the same instruction.
- A thread block is a group of threads that share the same memory, and may be synchronised to process data cooperatively. See 11.2.3 (p46): Thread synchronisation.
- A grid is a group of thread blocks that all execute the same kernel.
- A kernel is a function submitted to be executed in parallel on the GPU. The submission of a kernel to the GPU is called a kernel launch.
- A Streaming Multiprocessor (SM) is a collection of threads controlled by the same unit, that can execute multiple blocks at the same time.
- Shared memory is fast memory close to each thread of execution, accessible by each thread in a block. Its proximity to the GPU cores and access times is comparable to CPU cache.
- Global memory is slower memory that is accessible by any thread on the GPU.

11.2.2 Limitations and Considerations for GPU Programming

A key difference between CPU programming and GPU programming is how the hardware deals with branching. I define branching here to be any code that causes threads to execute different instructions based on a condition. Thus, selection statements and condition-controlled loops are all examples of branching. Branching means that each thread in a warp may not execute the same instruction at the same time, and this is called warp divergence. When threads in a warp diverge, the warp as a whole must execute each branch serially, which leads to many inactive threads and computational inefficiency.

To avoid such inefficiencies, branching should be kept to a minimum. My kernels make good use of boolean multiplication - multiplying a boolean (cast from true or false to 1 or 0) by certain parts of a formula to eliminate it, which means more instructions than necessary are executed by each thread, but the threads may maintain parallel execution. If branching is unavoidable, the code should then be structured to ensure that each warp executes one branch. If every other element of a list is to be processed differently, for example, then warps should process all even or all odd indices of the list to avoid divergence.

Another key area for optimisation is memory access. Memory transfer between the CPU and GPU is slow, and should be kept to a minimum and/or completed asynchronously. Furthermore, threads access memory most efficiently when it is in their own block's shared memory, rather than GPU-wide global memory. Especially for kernels that access the same memory location multiple times, it is often faster to copy from global to shared memory at the start of a kernel, and back at the end, so that during execution threads only access data from fast shared memory. The reduction kernel algorithm 11.2 (p48)

11.2.3 Thread Synchronisation

Threads must often work together to perform a task. For efficiency, a thread block should be able to execute independently of other thread blocks so that only one kernel launch is needed. Kernel launches are costly in terms of execution time. Therefore, any execution dependencies should be contained in a single thread block. For example, a thread block may load values from global memory into shared memory and then perform calculations on the values in shared memory and then return results to global memory. This is a very common pattern in GPU programs, since operations on shared memory is much faster than accessing global memory multiple times. Traditionally, thread synchronisation was possibly by a single call to `__syncthreads` which would synchronise all threads before continuing. With the advent of CUDA Cooperative Groups, an abstract class `thread_group` was defined which can represent a thread block, warp, or tile (variable-size group of 2, 4, 8, 16, 32, 64 or 128 threads). This class also defines synchronisation methods. This common interface allows for a kernel to be run on variable size tiles based on the size of the input. In the reduction kernel algorithm 11.2 (p48) implementation, the part of the algorithm that performs a reduction uses the abstract `thread_group` to allow for flexibility in the usage of the GPU hardware.

11.3 GPU Algorithms

11.3.1 Obstacle-Aware Algorithms

Though obstacle handling for CPU functions may be completed via `if` statements, the same task is more complex in the context of warps. To ensure that all warps can execute the same instructions, algorithms should use boolean multiplication to avoid if statements. Algorithm 11.1 provides an example of how boolean multiplication can be used to only execute an instruction if the containing cell has both the self and east bits set (as may be used for computation of horizontal velocity).

```
Input  : The cell's flag flag, and a computation to perform computation.
Output: The performed computation stored in result, if the cell is a valid cell. Otherwise 0.
/* Extract the SELF and EAST bits of the flag by ANDing with a mask and
   shifting so the bit is in the 1s position.                          */
selfBit ← flag & 0b00010000 >> 4;
eastBit ← flag & 0b00000100 >> 2;
result ← computation() × selfBit × eastBit;
```

Algorithm 11.1: Sample obstacle-aware algorithm

11.3.2 Reduction Kernels

Reduction kernels take an array of values and performs an operation on pairs of values until only one value is returned. The two reduction values implemented in the project are sum and maximum. Algorithm 11.2 (p48) is a reduction algorithm for computing the sum of the array, but the changes to make this algorithm perform another operation is a trivial matter of changing the operation in both index threshold loops. The algorithm described operates on a 2D array. It works in 2 stages. In the first, a kernel is launched with a number of blocks equal to the number of columns, and enough threads per block to span the height of a column. These blocks then perform a parallel reduction on each column, to create a partial sum to be stored back to global memory. The second stage requires a single block of enough threads to span the width of a row. It takes all of the partial sums in global memory and performs a reduction to get the final value across the entire array. Figure 11.1 shows the process with an array of width and height 8.

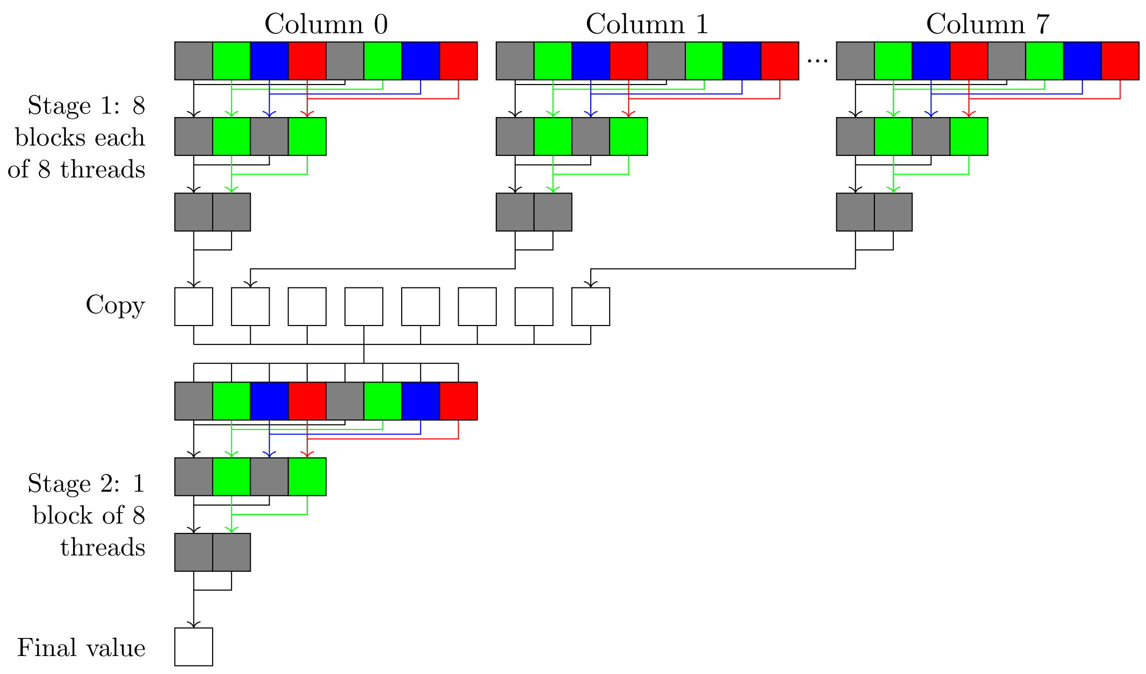

Column 0
Column 1
...
Column 7
Stage 1: 8 blocks each of 8 threads
Copy
Stage 2: 1 block of 8 threads
Final value

Figure 11.1: 2-stage reduction algorithm

```
Input  : A 2D array field and its dimensions xLength and yLength, with pitch pitch
Output: The sum of all of the input elements
Function FieldMax
    partialSums ← array of length xLength
    Spawn xLength × pitch threads, organised into xLength blocks.
    Assign each block an element in partialSums
    Each thread executes
        if thread rank > yLength then
            Load value from field to shared memory
        else
            Load a 0 into shared memory
        end
        Synchronise threads
        indexThreshold ← block size
        while indexThreshold > 0 do
            if thread rank < indexThreshold then
                /* Perform reduction                                    */
                Add together the value in shared memory at thread rank to the value in shared
                  memory at thread rank + indexThreshold
                Store this value in shared memory at thread rank
            end
            Synchronise threads
            indexThreshold ← indexThreshold /2
        end
        if thread rank = 0 then
            Store value at index 0 of shared memory in partialSums
        end
    end
    Spawn xLength threads in 1 blockEach thread executes
        if thread rank > yLength then
            Load value from field to shared memory
        else
            Load a 0 into shared memory
        end
        Synchronise threads
        indexThreshold ← block size
        while indexThreshold > 0 do
            if thread rank < indexThreshold then
                /* Perform reduction                                    */
                Add together the value in shared memory at thread rank to the value in shared
                  memory at thread rank + indexThreshold
                Store this value in shared memory at thread rank
            end
            Synchronise threads
            indexThreshold ← indexThreshold /2
        end
        if thread rank = 0 then
            Return the value at index 0 of shared memory as the final value
        end
    end
end
```

Algorithm 11.2: Reduction kernel algorithm

11.3.3 Red-Black SOR

The SOR algorithm for the pressure is a sequential algorithm, i.e., each cell depends on the values of the cells around it. In the CPU backend, parallelisation is not used and thus the cells may be evaluated from bottom left to top right. In the GPU backend, however, this is not possible. [2] describes the "red-black" approach, wherein cells are coloured "red" or "black" based on whether the sum of their coordinates are odd or even. Cells with even coordinate sums are marked "black" and evaluated first, then "red" cells can be evaluated using the up-to-date values of its neighbouring black cells. This technique requires more iterations to converge than the serial technique used in the CPU backend, but massive speed increases can be gained from evaluating all the red cells in parallel and then all the black cells in parallel. In both implementations of the SOR technique, a residual norm is calculated in order to determine whether the calculation may be stopped. The residual norm requires a summation reduction kernel, and therefore requires more processing overall than the actual SOR iteration. Therefore, in my implementation I only calculate the residual norm every 10 iterations to avoid this lengthy process every iteration.

11.3.4 Drag Calculation

Due to the way that, for parallelisation, the number of threads must be known in advance, the drag calculation portion of the drag calculation will be implemented as a class. This is to enable processing tasks like finding the obstacle coordinates to be calculated only once, and saved as a private variable to the class. The procedural approach taken by the other calculation routines, while faster than an object-oriented approach in terms of processing speed, is not appropriate here because the tasks like finding the obstacle boundary cells would have to be processed every timestep to yield the same result, and would actually be slower overall than the object-oriented approach.

A new class `DragCalculator` shall be defined, with structures for the boundary cells relevant to pressure drag or viscous drag. It exposes a method, `ProcessObstacle`, which is called whenever the simulation obstacle changes. This fills the boundary cell structures so that these are readily available for each timestep.

Chapter 12

Backend Structure

The backend, which runs all of the computations described in the previous two chapters, is split into 2 sections based on hardware. The CPU backend will run on any computer and only makes use of the CPU. It has little parallelisation, although calculation of pressure SOR cycles is split into sections to make use of multi-threading. The GPU backend only runs on computers with an NVidia GPU, and makes much more use of parallelisation and the parallel processing the GPU offers. Elements that are common to both backends are in a library file BackendLib, so that both backends can access the same methods and classes without having to build them twice.

12.1 CPU Backend

The CPU backend is programmed mostly procedurally, although has a central class `CPUSolver` that provides a common interface for pipe management. The timestep method calls multiple functions to update the field values continuously using the equations in the previous chapters. Figure 12.1 (p51) represents this diagrammatically.

12.2 GPU Backend

The GPU backend is programmed as a mix of GPU kernels and CPU functions, depending on whether the function can be parallelised or not. Again a central class exists to interface with the pipe, `GPUBackend`, and the timestep method calls all of these functions. A commonly used pattern is a CPU function that acts as a wrapper for multiple GPU kernels, especially when outer boundaries are handled differently for a field. Figure 12.2 (p52) represents the structure.

12.3 Backend Library

The CPU and GPU backends must interact with the pipe in the same way, and to avoid repetition a library file is to be created which contains all of these methods. The backend library also contains an abstract class `Solver`, which contains abstract methods for getting data and starting computation. This exposes a common interface that the CPU and GPU backends, though wildly different in terms of hardware implementation, must both adhere to so that the data can be accessed in the same manner to be sent to the frontend. It also contains a class `BackendCoordinator`, which coordinates with the frontend via the pipe and calls the methods defined in `Solver`. Figure 12.3 (p53) shows the classes that exist in the backend. Each class is compiled in a different assembly.

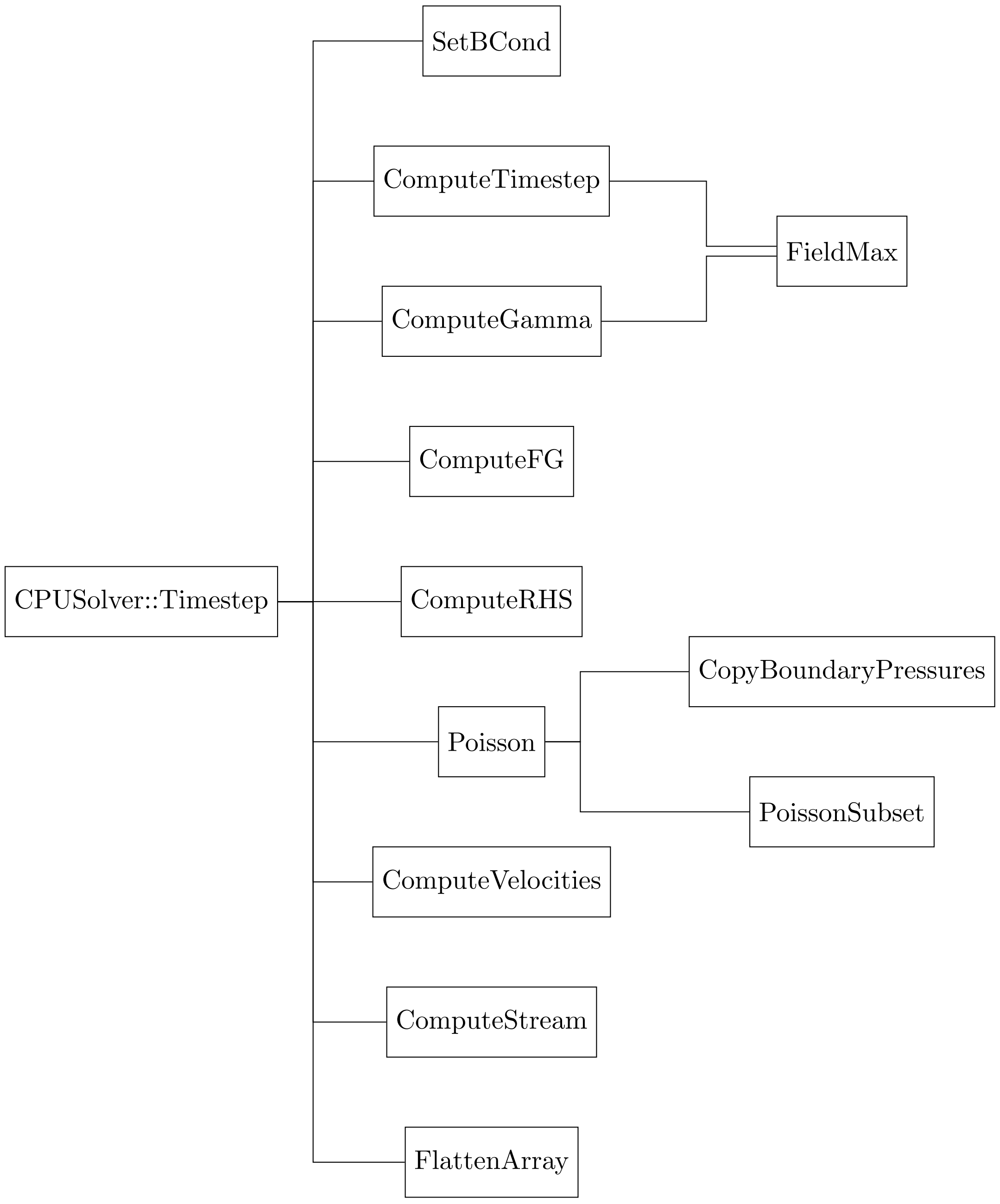


Figure 12.1: CPU backend hierarchy chart

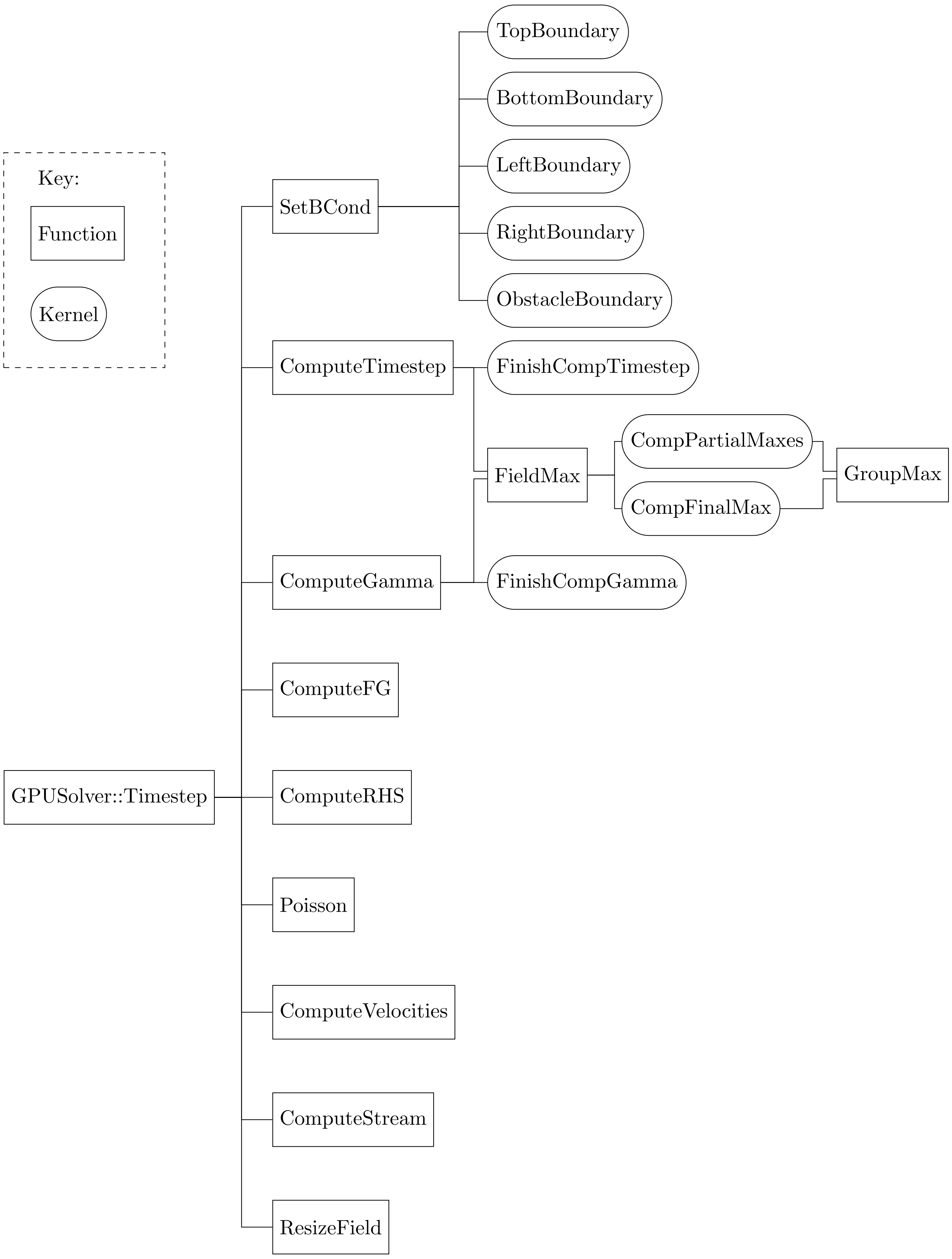


Figure 12.2: GPU backend hierarchy chart

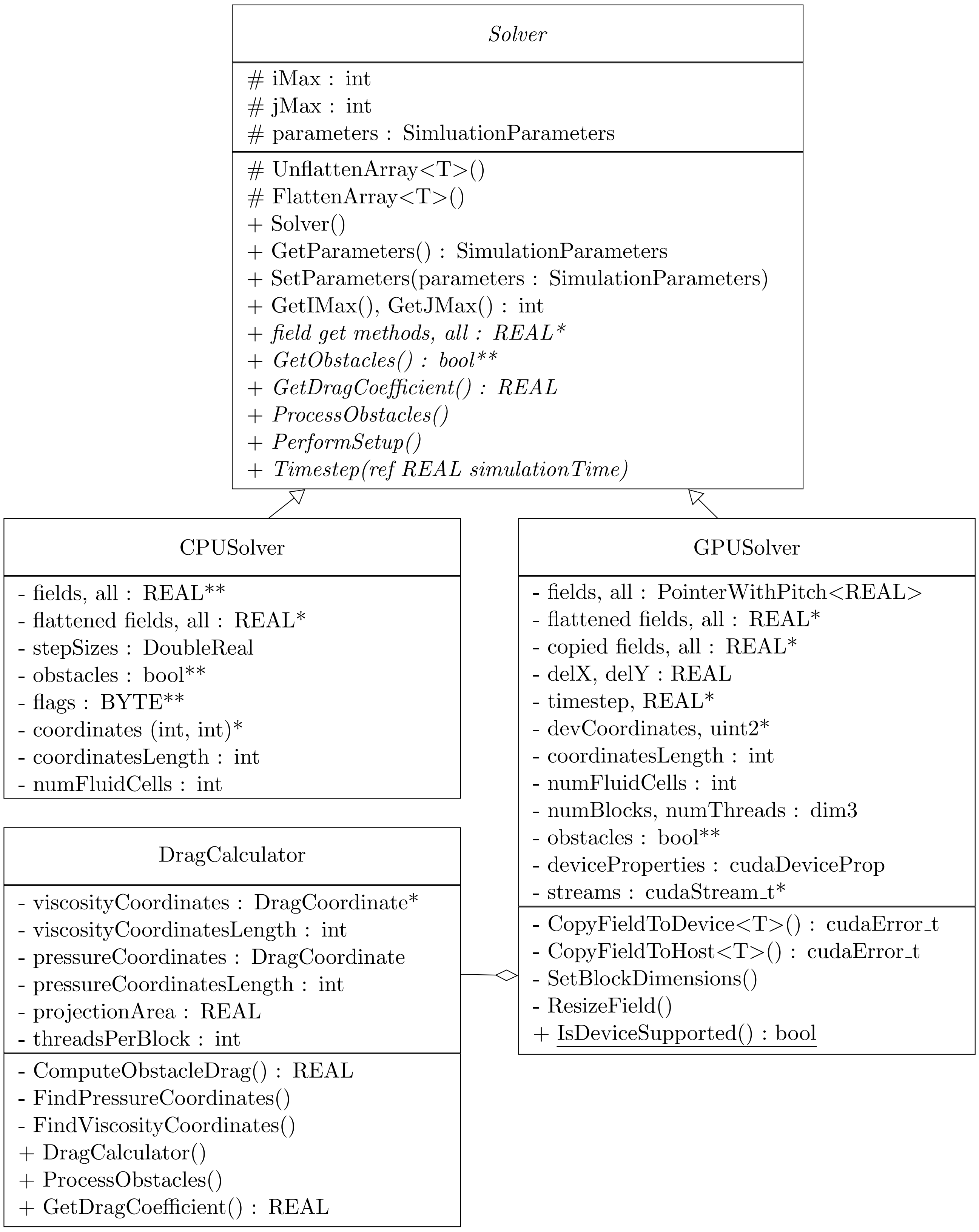

Solver
# iMax : int
# jMax : int
# parameters : SimluationParameters
# UnflattenArray<T>()
# FlattenArray<T>()
+ Solver()
+ GetParameters() : SimulationParameters
+ SetParameters(parameters : SimulationParameters)
+ GetIMax(), GetJMax() : int
+ *field get methods, all : REAL**
+ *GetObstacles() : bool***
+ *GetDragCoefficient() : REAL*
+ *ProcessObstacles()*
+ *PerformSetup()*
+ *Timestep(ref REAL simulationTime)*
CPUSolver
- fields, all : REAL**
- flattened fields, all : REAL*
- stepSizes : DoubleReal
- obstacles : bool**
- flags : BYTE**
- coordinates (int, int)*
- coordinatesLength : int
- numFluidCells : int
GPUSolver
- fields, all : PointerWithPitch<REAL>
- flattened fields, all : REAL*
- copied fields, all : REAL*
- delX, delY : REAL
- timestep, REAL*
- devCoordinates, uint2*
- coordinatesLength : int
- numFluidCells : int
- numBlocks, numThreads : dim3
- obstacles : bool**
- deviceProperties : cudaDeviceProp
- streams : cudaStream_t*
- CopyFieldToDevice<T>() : cudaError_t
- CopyFieldToHost<T>() : cudaError_t
- SetBlockDimensions()
- ResizeField()
+ IsDeviceSupported() : bool
DragCalculator
- viscosityCoordinates : DragCoordinate*
- viscosityCoordinatesLength : int
- pressureCoordinates : DragCoordinate
- pressureCoordinatesLength : int
- projectionArea : REAL
- threadsPerBlock : int
- ComputeObstacleDrag() : REAL
- FindPressureCoordinates()
- FindViscosityCoordinates()
+ DragCalculator()
+ ProcessObstacles()
+ GetDragCoefficient() : REAL

Figure 12.3: Backend class diagram

Chapter 13

Backend-Frontend Interface

As seen in Figure 6.1 (p23), the backend and frontend need a way to communicate. In the implementation, a Named Pipe is used to send data between the two executables.

13.1 Named Pipes

In Windows terminology, a named pipe is a Windows communication method to allow for Inter-Process Communication (IPC). The way that C++ and C# handle this is different, but the underlying concept is that the pipe can be written to and read from as you would a file or other device.

13.2 Communication Protocol

To allow for efficient and error-free transmission, a protocol must be defined that both processes adhere to and understand. A full documentation of this protocol can be found in Appendix C (p78) - Pipe Protocol. The types of data that will need to be transferred through the pipe may be split into 2 categories - *control commands* and *data*. Data is simple to define - all that is needed is a sequence of bytes (after all, all data can be simply converted to a sequence of bytes). Control commands however, if they are to be efficient (i.e., not simple text commands that would take much longer to send and process), may be more complex and therefore require careful thought. A key concept for the protocol is the idea of a *data unit*. This is defined as the size, in cells, of the simulation domain ($i_{max} \times j_{max}$), as transmissions of data will be in multiples of this unit. Thus, fixed-length data will have its length given in data units, and continuous data will have markers after each data unit number of bytes. The full protocol specification is in Appendix C (p78) - Pipe Protocol Specification.

Control commands are split into 4 types - STATUS, REQUEST, MARKER and ERROR, each of which have unique 2-bit patterns at the start of their byte. The 6 remaining bits define what message is being sent, or provide rudimentary parameters.

13.3 Serialisation

Serialisation defines deconstructing a data type into a format used for transmission, and in a way that allows for reconstruction after transmission. In this case, this means converting different data types into bytes to be sent along the pipe, and to be reconstructed at the other end.

During serialisation, in almost all cases where more than one byte is transmitted a buffer is used to hold the data before transmission. In the general case of serialisation, the data must be converted into bytes (this is described in more detail below), added to the buffer, then the entire buffer is transmitted at once.

13.3.1 Primitive Data Types

The primitive data types - single `int` and `float` variables - are fairly easy to transmit. `int` values are by far the easiest - the int is truncated to a byte, then right-shifted by 8 bits and this is repeated until all 4 bytes of the int have been stored. The right-shift operation (as opposed to some sort of division) is the most efficient for this purpose, because specific hardware optimisations exist for logical shift operations for `float` data, the 4 bytes of the IEEE single-precision floating point representation[1] are transmitted contiguously.

13.3.2 Field Data

Field data is slightly harder to transmit, due to the requirement to flatten the 2D array before transmission. In the C++ code, extensive use of the `memcpy` function is made: each column of the 2D array is stored contiguously in memory, and is copied from the 2D field array to a 1D buffer, which is subsequently sent.

13.3.3 Obstacle Data

Obstacle data, represented as a 2D array of booleans, is able to be optimised since booleans only need 1 bit of data to be stored. A compression algorithm on the frontend, and a decompression algorithm on the backend, is used along with array flattening to allow for 8x fewer bytes to be transmitted.

```
Input  : A boolean array obstacles.
Output: A byte array buffer to be sent down the pipe.
index ← 0
for $i$ ← 0 to length of obstacles do
    /* Shift each boolean onto the byte                    */
    shiftedObstacles ← obstacles [$i$] << $i$ MOD 8
    buffer[index] ← buffer[index] |shiftedObstacles
    if $i$ MOD $8 = 7$ then
        index ← index + 1
    end
end
```

Algorithm 13.1: Boolean data compression algorithm

[1]This is simply a commonly used binary representation for floating-point storing of real numbers. C# uses this internally, and the C++ code as compiled with Microsoft's compiler MSVC does also.

Part III

Technical Solution

Chapter 14

Examples of High-Level Skills in NEAFluidDynamics

14.1 Model

14.1.1 CPUBackend Staggered Grid (Complex Scientific/Mathematical model)

The backend computation works upon a staggered grid system: quantities such as pressure, velocity and stream function are all discretised into individual cells, in which all quantities are assumed to take a single value. However, rather than taking all of the quantities to lie at the centre of cells, some quantities such as the velocity lie at the edge of cells, and the stream function at the top right of the cells. More detail can be found in chapter 10 (p38), but the use of this complex model requires programming complex array handling, especially for boundary conditions. A good example of this is at Boundary.cpp (p109).

14.1.2 UserInterface OOP (Complex OOP and Dynamic Generation of Objects)

The user interface for the project makes use of WPF, which lends itself well to an object-oriented design. Each of the different windows of the application inherit from `UserControl` so that the master object for the application (`App`) can switch these windows around by only knowing whether the current UserControl is a full-screen window or a popup window. In fact, due to the way that the window switching is handled, in order to follow the principles of RAII, the type is passed to `App` and then `App` instantiates it using `Activator.CreateInstance`. This qualifies for dynamic generation of objects because the `UserControls` are instantiated when the user presses buttons to navigate through the application. See App.xaml.cs (p195).

Though lots of the inheritance is from WPF classes, such as UserControl, similar classes are often grouped under a single parent. `ViewModel` defines common methods and fields for ViewModels in the MVVM pattern, including the object that holds simulation parameters and an implementation of the WPF interface `INotifyPropertyChanged`. The commands all implement the WPF interface `ICommand`, but through inheritance from the abstract `CommandBase`. Commands that are defined to have a parent ViewModel, and must access an instance of it, inherit from `VMCommandBase`. This is an abstract, generic class that allows for dependency injection of a view model into the command class. See Commands.cs (p220).

14.1.3 Backend OOP Model

In order to ensure consistency in the interfaces of both backends, an abstract `Solver` class was defined so that the other items in `BackendLib` can access the CPU or GPU backend via this common interface. `Solver` (see Solver.cpp (p105)) defines methods for retrieving the pressure, velocity and stream function fields, as well as methods to process obstacles and perform setup. Therefore, BackendCoordinator.cpp (p85) can call the correct methods and retrieve the correct data without knowing whether the solver is a `CPUSolver` or `GPUSolver`.

14.2 Data Structures

Multiple complex data structures are required for the project. Simpler data structures are implemented as `struct`s, such as `SimulationParameters` in Definitions.h (p91), but more complex data structures are detailed below.

14.2.1 GPUBackend Pitched Arrays (Complex array handling)

When programming on a GPU, memory accesses are often a key area of optimisation since both transfers between global and shared memory and GPU and main memory are slow when compared to shared memory accesses and general computation. For 2-dimensional arrays, this problem is further exacerbated because memory is accessed in groups called banks. Multiple threads can access one memory bank independently of other threads, provided they do not try to access the same bank. A logical option, therefore, is to match the length of a column with the length of a bank, so that each column starts at the start of a bank and avoids a bank conflict (where multiple threads try to access memory from the same bank). The CUDA API has functions to manage this, `cudaMallocPitched` for example, so the size of a bank does not have to be known at compile time. However, accessing memory in these arrays is less well-documented. In Definitions.cuh (p160), macros are defined for accessing of pitched memory by using pointer casts to describe memory locations in bytes or in elements.

14.2.2 UserInterface Queues (Queue Operations)

The user interface has multiple applications where queues are required. Therefore, a hierarchy of queue classes was created (see Queue.cs (p250), ResizableLinearQueue.cs (p251) and CircularQueue.cs (p219)) to serve these applications.

One application was in moving averages (see MovingAverage.cs (p228)), which requires a constant-length queue implemented as a circular queue. Each item in the moving average must be enqueued when it is added to the sum, so that the correct item is removed from the sum when dequeue is called. A circular queue implementation is appropriate for this use case because the maximum size of the queue is constant, and a circular queue ensures that no more memory than is needed is required.

Another application is the queue for user interface parameters. Every time a parameter is changed, an event handler in BackendManager.cs (p208) adds the parameter to a queue. At the end of each timestep, the frontend sends any parameters in this queue to the backend by successively dequeuing until the queue is empty. For this use case, the queue is implemented as a resizable linear queue, since the size of the queue is constantly changing. This was chosen because a resizable linear queue is able to be very memory efficient when the size of the queue is changing, at the expense of a small amount of processing overhead on some operations.

14.2.3 FileMakerCLI Stacks (Stack Operations)

The file maker command-line interface requires parsing of mathematical expressions as text. As documented in Section 8.1.1 (p28), this requires many stack operations (to convert an infix expression to postfix, and then to evaluate the postfix). Therefore, a class `ResizableStack` was created, that exposes the standard `Pop`, `Push` and `Peek`, as well as an `IsEmpty` property, to allow stack operations. This abstracts from the underlying resizing method, based on C#'s built-in `Array.Resize`, that uses more or less memory space based on the number of items on the stack. This represents complex stack operations, both in the use of the stack and the underlying implementation. See ResizableStack.cs (p145)

14.3 Algorithms

Algorithms in the project, where sufficiently complex, are given explicitly in the design section. A brief list is below:

- Obstacle serialisation: for transfer across the pipe, the frontend compresses its obstacle array (an array of `bool`s) from 1 datum per byte to 8 data per byte, using efficient bit manipulations such as logical shifts and bitwise OR. A similar algorithm, but reversed, is used to decompress the obstacle data once the backend receives it. See Algorithm 13.1 (p55), PipeManager.cpp (p97) and PipeManager.cs (p238).

- Multi-threaded pressure SOR solver: the CPUBackend's pressure equation solver uses multi-threading to make the greatest use of the CPU. It determines the parallelisation capability of the CPU, then calculates and runs the most efficient possible breakdown of the domain. Calculation of the pressure residual is completed by the individual threads, and must be summed. I implement this with a simple recursive summation algorithm. See Computation.cpp (p112)
- Management of boundary conditions (`SetBoundaryConditions` and `CopyBoundaryPressures`): boundary condition application is a complex task and is implemented efficiently using a flag array. The flag array is described in more detail in Chapter 10 (p38). It allows for efficient bit manipulations to take place, especially in the application of pressure boundary conditions.
- Rendering of streamlines: For the visualisation, a complex algorithm is required to compute the contour lines of the stream function. First, lists are instantiated to contain each level set (the indices of each streamline), then these level sets must be filled with all the indices whose stream function value are in the level set. See Algorithm 9.1 (p37) and GLHelper.cs (p306).
- Tokenisation of user input: In FileMakerCLI, user input must be converted into individual units called tokens. These are not quite as simple as a single character, as functions like *sin* and numbers like 100.4 are individual tokens. See Algorithm 8.1 (p29), and ConstraintParser.cs (p139).
- Conversion of infix to postfix (Shunting Yard Algorithm): In order to allow for evaluation of an expression, the infix input must be converted to postfix, using the well-known Shunting Yard Algorithm. See Algorithm 8.2 (p30) and ConstraintParser.cs (p139).
- Evaluation of postfix expression: To evaluate the postfix expression, a stack-based evaluation algorithm (Algorithm 8.3 (p31)) was implemented. See RPNConstraint.cs (p147).

Part IV

Testing

Chapter 15

Test plan

15.1 Objective-Based Tests

Drag coefficients are taken from Wikipedia page on drag coefficients, and are for Reynolds number 10^4 to 10^6. See right hand column of `https://en.wikipedia.org/wiki/Drag_coefficient#/media/File:Hoerner_fluid_dynamic_drag_coefficients.svg`

Test	Description	How to reproduce	Data	Objective
1	Testing creating a circular obstacle with the file creator.	Load the file creator CLI and input the data.	iMax: 500, jMax: 250, constraint: $(x - 250)^2 + (y - 125)^2 < 625$.	4
2	Testing the drag coefficient of a circular obstacle.	Set the obstacle and allow the drag coefficient to stabilise.	Circle created in above test, Reynolds number 10^5. Drag coefficient: 1.2.	1
3	Testing creating a rectangular obstacle with the file creator.	Load the file creator CLI and input the data.	iMax: 500, jMax: 250, constraints: $x \geq 220, x \leq 280, y \geq 135, y \geq 165$.	4
4	Testing the drag coefficient of a rectangular obstacle.	Set the obstacle and allow the drag coefficient to stabilise	Rectangle created in above test, Reynolds number 10^5. Drag coefficient: 2.1.	1
5	Showing the effect of editing fluid speed during the simulation.	Start a simulation with default parameters; modify the fluid speed.	N/A	2
6	Showing the pre-simulation popups for parameters and advanced parameters.	Start the program and change parameters, and navigate to advanced parameters.	N/A	3
7	Showing the dynamic units throughout the program.	Change a unit in the config screen, note sliders changing. Start a simulation, note that the new units are used.	N/A	6
8	Showcasing the visualisation and flow lines, and ensuring the visualisation is big enough.	Start a simulation with default parameters, let it stabilise and see flow lines and that the visualisation takes up more than half of the screen.	N/A	7, 9

9	Showcasing panels differentiated by icons on the simulation screen.	Start a simulation with default parameters, click the side panel buttons to open menus.	N/A	8
10	Showcasing resizeble, dynamic UI	Open the program, go out of full screen and resize the UI, start a simulation, do the same again	N/A	10
11	Showing running of the simulation on a GPU	Use a command to show the current GPU, then show the simulation running and use task manager to show the executable using the GPU.	Powershell command to get the name of the current GPU: `(Get-WmiObject Win32_VideoController).Description`	13
12	Showing the speed at which the simulation stabilises	Set the default obstacle and start a timer. Time how long the simulation takes to stabilise.	N/A	15
13	Showing the CPU-only mode	Use a device without an NVidia GPU and show the simulation using the CPU instead.	Same powershell command as above.	16

15.2 User Input Tests

Test	Object tested	Example input data	Input type
14a	Fluid speed slider	2	Normal
14b	Fluid speed slider	40	Boundary
14c	Fluid speed slider	41	Invalid
14d	Fluid speed slider	2a	Erroneous
15a	Visualisation max slider	10	Normal
15b	Visualisation max slider	18 (velocity mode), 100000 (pressure mode)	Boundary
15c	Visualisation max slider	20 (velocity mode), 110000 (pressure mode)	Invalid
15d	Visualisation max slider	2a	Erroneous
16a	FileMaker iMax / jMax input	100	Normal
16b	FileMaker iMax / jMax input	1	Boundary
16c	FileMaker iMax / jMax input	-5	Invalid
16d	FileMaker iMax / jMax input	2a	Erroneous
17a	FileMaker constraint input	$3\sin(2x^2 - 5)^3 - 5tan(y)tan(.3y) + 100 <= 20$	Normal
17b	FileMaker constraint input	$5san(x) > 3$	Erroneous
17c	FileMaker constraint input	$5sin(x > 3$	Erroneous
17d	FileMaker constraint input	$a\cos(2x) > 3$	Erroneous

15.3 Use Case Test

Test 18 is a more user-focussed test that walks through a standard use case for the program, and also tests objective 5. The test shows the difference between a square front surface and a curved front surface for the car.

First the program is run and obstacles from file is deselected. Reynolds number is increased to around 10^5 and surface friction is set to 0.5 to represent a semi-smooth surface. Then, an obstacle is defined using the click-and-drag node system, in the shape of a Greenpower car. In this first iteration, the front of the car is a rounded shape. The simulation is then allowed to stabilise. The drag coefficient is noted, and then the drawn obstacle is changed to have a squared-off front. The drag coefficient is again noted, and the difference would inform whether the user would put a flat or rounded shape for the front of the car. The simulation should show that the rounded shape is more aerodynamically efficient.

15.4 Testing Video

The testing video is uploaded as an unlisted video on YouTube. A QR code is below for ease, but the link is at `https://www.youtube.com/watch?v=UkojeVlYWSQ`

Part V

Evaluation

Chapter 16

Accordance With Objectives

As is further discussed below, all of the objectives set for the program were met. However, some objectives were met in a way that compromises intuitiveness or ease of use.

16.1 Backend / Computation Objectives

Objective 1 requires that the program can calculate drag coefficient to 1 significant figure. Though the program is successful in calculating this, 1 significant figure is quite lenient for a drag coefficient calculator, and may not provide a good insight into the aerodynamic properties of the obstacle. The scaling is also not quite right for large obstacles that take up more of the simulation domain.

Objective 12 required that the simulation be based on the Navier-Stokes equations. This is clearly met as it formed a key part of the design of all of the algorithms for the backend (See Section 10 (p38)).

Objective 13 requires that the simulation is able to run on NVidia GPUs, which was simple to comply with as I was using CUDA (an NVidia platform).

Objective 14 requires that the backend use multithreading as part of their calculation. The CPU backend uses some multithreading (namely in the pressure iterations), but the GPU backend uses multithreading at every opportunity to take advantage of the GPU hardware.

Objective 15 requires that the simulation stabilises within 10 seconds. A relatively stable state can be achieved within 10 seconds, so the objective was fulfilled, but the time taken for drag coefficient to become valid is much longer. The program also takes longer to stabilise if there are more simulation cells. Further methods to improve simulation speed are discussed in Section 18.1.3 (p70).

Objective 16 requires a CPU-only mode to facilitate running the program on school computers, or more generally computers without a NVidia graphics card. As well as forming part of the project, this was where I was able to figure out how to convert the mathematical formulae given in [1] into code. However, it is not very well-optimised, and multithreading capability is limited to the pressure SOR calculations.

16.2 User Interface Objectives

Objective 2 requires that fluid speed be editable during the simulation. In fact, I was able to implement far more editable parameters. This is achieved through a UI class, `ParameterHolder`, containing all the information about the program parameters. Every time a parameter changes, its new value is added to a queue that, after every timestep of the backend, is checked by the frontend and any new parameters sent.

Objective 3 requires a popup before the simulation exists to set initial parameters. This was successfully implemented using the `ConfigScreen`, which is the start screen for the application and contains sliders to change values.

Objective 6 requires that units throughout the program are dynamic and configurable. Selecting units was achieved through a `UserControl` `UnitConversionPanel`, with dropdown menus to select units. Data binding was used in all sliders to bind to a class `UnitHolder`, which contained the current units state of the program via dependency injection.

Objective 8 requires side panels on the simulation screen, selected by clicking icons. This is all handled in the view layer, in SimulationScreen.xaml (p291) / SimulationScreen.xaml.cs (p295)

Objective 9 constrains the layout of the simulation screen to make the visualisation take up at least half of the screen. Simply setting grid sizing in the `SimulationScreen` code is simple enough to comply with this objective, and keeps the UI dynamic.

Objective 10 requires a dynamic UI, which is achieved by using WPF's `Grid` extensively as this scales its constituent elements to fit the screen size. Maximum row/column values were used to ensure that title pages did not get too big. Text sizing for headings was also dynamic, making use of a `UserControl` `ResizableCentredTextBox` to scale the text up and down depending on the size of the rendered control.

Objective 11 requires that the UI uses OOP and MVVM. OOP is clearly used (see, for example,the UI class diagram (p27)), and MVVM is used to a good extent by making a clear difference between the views and view models.

16.3 Visualisation/Obstacle Defining Objectives

Objective 4 was successfully achieved by using binary files and reading these in to send to the backend in the same way that user-drawn obstacles are sent to the backend. A problem was encountered where the size of the simulation domain must be set at startup for the backend, so that it can allocate memory before any further communication happens. Therefore, this data had to be sent as the backend executable was run, in the form of a command-line parameter that the program is launched with. A class `ObstacleHolder` exists to contain this data, and pass it between screens through dependency injection. In the binary file, the simulation domain sizing was transmitted in the first 8 bytes as metadata.

Objective 5 defines a click-and-drag node system. This was implemented using cubic splines, which have the disadvantage that every point on the spline changes when one point moves, which makes it difficult to make small changes to one part of the obstacle. Other methods to implement a click-and-drag node system are discussed in section 18.4.1 (p72).

Objective 7 defines what the flow visualisation should look like: colour gradient from blue to pink to show areas of high velocity or pressure, and streamlines. OpenGL was used to set the colour of each vertex (which corresponds to one backend cell), and draw streamlines on top of it based on contours of a stream function.

Chapter 17

Feedback From User

The user, Ben Baxandall, was again interviewed in order to gain feedback about the project. The transcript is written here in the same format as above, with the interviewer (me) in blue, and the interviewee (Ben) in red.

17.1 Interview

(Start of interview. Ben was notified that this would be recorded and used for an NEA project before recording started.) Sam: *(Looking at the two executable files)* So there's the main user interface, the click-and-drag node system is done, and I've also got a thing where you input mathematical inequalities to make an obstacle, but I don't think we'll focus on that very much. User Interface is the one you'll probably want.

Ben: *(Clicks on User Interface executable) Ok.*

Sam: *(Program now opens to Config Screen) Yeah so this is the project, so there's lots of parameters and things, hopefully it all makes sense.*

Ben: *(Looking at the sliders) So do I just move these?*

Sam: Just go for it, yeah.

Ben: *(Moving the sliders around and noting the text box changing)* I like that, yeah. *(Navigating around the UI)* Do I select an obstacle?

Sam: You can select a file. That's for when you make one with constraints. So if you uncheck the box you can make you own.

Ben: Ok. *(Presses the Simulate button, and enters the simulation.)* Oh so is this the default shape?

Sam: Yeah, and then if you want to change the obstacle, press [Edit simulation obstacles].

Ben: *(Entering obstacle editing mode)* Oh that's cool *(Moving control points around)*, that's really cool.

Sam: Left click is to add a control point, right click to get rid of it.

Ben: *(Starting to edit the obstacle)* Can I rotate the shape?

Sam: No, sadly not.

Rotating the obstacle could be added.

Sam: *(Ben continues to try out the click-and-drag node system)* So you could try to make a car shape.

Ben: I'm trying to remember what shape the front was. I think we've got a rounded front. *(Starts to make a car shape)* I guess that's the rough shape. *(Presses Finish editing, simulation starts around the drawn car)* Ooh I like that.

Ben: *(Navigating to units side panel)* So where are these measurements coming from, like where does it happen?

Sam: So if we change speed units *(changes speed units to miles per hour)* and then go to the other parameters panel...

Ben: Oh I see - it's miles per hour now. So what do [the fluid speed and material friction sliders] do?

Sam: So material friction is like, if you set this all the way [to 1], that means the fluid is slipping past [the obstacle] with no friction, but if you set it all the way down like it is by default then there's a lot of friction going on. *(Changing the fluid speed slider)* This makes the fluid go faster, and slower fluid.

Ben: *(Noticing streamlines disappearing)* What's happening with the lines?

Sam: With the streamlines, the problem I've found is that when I have a higher velocity, you have to put the tolerance up. The tolerance is like how well the streamlines fit the flow - if I put it all the way up then it's a bit blocky, but if I put this too low then they disappear, so you've got to find a balance.

Ben: It would be nice if that could be set automatically, if that's possible.

Sam: Yeah that's a good idea.

Contour tolerance could be set automatically based on the lowest value that still shows all the streamlines.

Ben: *(Ben continues to play around with the simulation, and tries to edit the obstacle again. He finds that it is difficult to make changes in one place without affecting the rest of the obstacle)* Ooh, whoops. Yeah the click-and-drag node system is, well it's fine and it works, but it's a bit tricky to use.

The click-and-drag node system could be reviewed and improved, or could use a different method to draw splines.

Ben: What's the file maker like? Could I try that?

Sam: Yeah, so just close [the user interface program], and open the file maker.

Ben: *(Opening the file maker)* Oh no it's command line. I thought it was like an app.

Sam: Oh yeah, sadly not.

The file maker could have been implemented as a GUI.

Ben: So how many cells, like 100?

Sam: Yeah 100 by 100 is pretty standard.

Ben: File path, so is that the name?

Sam: Yeah, so just like obstacle or something.

Ben: *(Entering domain size and file path, and now greeted with the constraints message)* Oh so is this the inequality thing? I'll just put something. *(Ben enters a constraint, and presses q to finish)* So is that done?

Sam: Yeah, that's made the file. So then, we can go into the user interface and load it.

Ben: *Obstacle loads and covers half the screen* Oh I don't know what I've drawn!

Sam: Yeah, do you want to use a ready-made one?

Ben: Yeah good idea. *(Ben closes the program.)*

Sam: So I've got a circle or rectangle, which do you want?

Ben: Rectangle I reckon. *(Ben selects rectangle file and starts simulation)* Oh that's nice. *(Sees edit obstacle is greyed out)* Oh you can't edit it. Yeah that makes sense.

Ben: *(Entering visualisation settings)* What do [the visualisation min and max sliders] do?

Sam: So the maximum and minimum, that's like the biggest and smallest values.

Ben: *(Selecting pressure plot)* Oh where did it go?

Sam: Ah, I find that the pressure is generally quite a narrow range, so it's tricky to find that.

Ben: Could the min and max be automatic to? Like based on what the numbers are?

Sam: Oh ok yeah that's possible.

Min and max sliders could be set automatically based on the min and max value of the field being visualised.

Sam: Velocity plots generally give you a better view, because if we turn the max down...

Ben: Oh yeah that's good. Is this areas of higher velocity - the pink?

Sam: Yeah that's right. You can see the big low-velocity tail coming off of it.

Ben: Yeah. *(Moving the Number of streamlines slider)* Ooh nice.

Ben: *(Going back to the units panel)* Ooh what does the [unit system] do?

Sam: So you can choose the unit system, so like normal metric, or imperial.

Ben: Oh ok that's cool, like presets.

Sam: Yeah exactly.

Ben: Yeah that's helpful.

Ben: So is there anything else you want me to look at, or any questions?

Sam: Well that's it for the program, but there are some questions. General thoughts - what do you like and what do you think can be improved?

Ben: I think it's really advanced, like almost professional level. I think for Greenpower, I think it would need to be a little bit simpler. I think having the advanced parameters separate is good, but yeah I think it's really cool. I think the [file maker] could be simplified I guess, but that would just be a lot more effort.

Sam: Yeah, I think the main use case is the click-and-drag node system, which does have some issues I guess. Also, one other thing, how involved do you feel you've been in the project? Do you feel I've taken on your feedback?

Ben: I think so yeah, because you asked me about the UI and it looks like what I said. It's what, in my eyes, looked good I guess.

Sam: Ok perfect, thanks very much Ben.

17.2 Conclusions

Some key areas for improvement identified as a result of the interview were:

- **File Maker:** The file maker, as a CLI program, was deemed not very intuitive. It would have been better, given more time, to implement a GUI to improve ease of use.
- **Click-and-Drag Node System:** The click-and-drag node system was able to be used effectively by Ben, but he found it difficult to draw an obstacle and then make smaller changes. A different spline drawing method could have been used here.
- **General Ease of Use:** Ben commented on the fact that the program may be slightly too complex for the Greenpower team to get used to using the program. Methods could be implemented to further improve ease of use.

Also, the following were identified as possible extra features:

- **Automatic Contour Tolerance:** The contour tolerance could be set automatically based on the minimum tolerance that draws all the streamlines.
- **Automatic Min and Max Sliders:** The visualisation min and max sliders could be set automatically based on the min and max of the field being viewed.

All of the above are fully discussed in the following chapter.

Chapter 18

Areas For Improvement

The program in its current state meets all of the objectives set out for it, and functions well as a CFD simulator. However, there of course exist some issues or possible changes that could have been addressed if I were to revisit the project. Ideas given by Ben in the feedback interview (p67) are also given here.

18.1 Backend

18.1.1 OpenCL

NVidia CUDA is only able to be run on NVidia GPUs. This restricts the possible users as they have to have specific hardware. To enable GPU computation on different hardware, OpenCL (an alternative to CUDA that is supported on all hardware) could have been used. For the project, I did not choose this as it is generally accepted that there is a steeper learning curve for OpenCL than CUDA but, with more time to learn the technology, OpenCL would have made a better alternative.

18.1.2 Adaptive Grid

Currently, the size of the grid is fixed and is defined before the backend even starts executing. In the project, this was done so that the amount of memory that the backend needed to allocate was known at runtime and could be allocated for the whole of the backend's lifetime. However, the problem with this is that areas of simple laminar flow (far away from the obstacle) require the same amount of processing (and take the same amount of time) as complicated turbulent flow close to the obstacle. This is a poor allocation of resources, and therefore adaptive grid methods have been developed. Adaptive grids reduce the step sizes of the grid around complex flow, and increases it around simple flow. This means that an increase in simulation accuracy can be realised without changing the number of grid cells or the time taken to simulate.

18.1.3 Optimisation of Timestepping Loop

The backend is able to be optimised far further than its current state, given more time to profile and optimise. Furthermore, on each iteration, pressure iterations are terminated due to reaching a maximum number of iterations rather than the residual norm going below the limit. This means that the pressure iteration is terminated each time not because it is accurate, but because it runs for too long. This leads to slower simulations. To resolve this, I could use a relative tolerance, as described in [1, Chapter 3], or employ other methods to make the pressure iteration more accurate. Another key area that would speed up the simulation is more thorough optimisation through profiling. Some optimisation was carried out on the GPU backend, such as only computing the current residual every 10 iterations (because it requires a reduction kernel on the GPU), but further optimisation would speed up the simulation further.

The GPU backend could make use of CUDA graphs to speed up the timestepping loop. A CUDA graph has the kernels that are to be run, and the dependencies of each kernel and one another.

Therefore, the kernels can all be submitted at once and less time is spent synchronising the GPU as this is handled by the graph dependencies.

18.1.4 Drag Coefficient

The discretisation method for drag coefficient was not guided by any specific document (unlike the discretisation for other sections, which was in whole or part guided by [1]) and was derived by me. Therefore, there is a likelihood that there are oversights in the discretisation or the subsequent implementation in code that have led to inaccuracies in drag calculation.

18.1.5 Turbulence Modelling

The simulation is unable to deal with very turbulent flows in real time, as the number of cells has to be so large that the simulation cannot run efficiently, and there exists a trade-off between speed and accuracy. Speed was prioritised (as was the general feeling from the interview with Ben), but other methods exist to model turbulence without a large number of grid cells. The k-ϵ turbulence model, as described in [1] could have been used, which would have increased the accuracy of the simulation for turbulent flows. As the obstacles to be simulated were fairly streamlined, it was decided that this was not important.

18.2 User Interface

18.2.1 Overall Styling / Theme

The current user interface does not include much styling apart from the visualisation, and a consistent colour scheme could have been added to make the user interface look nicer.

18.2.2 Tooltips

Use of tooltips or a first use guide could have been implemented which would have improved ease of use (as described in the interview). Tooltips for sliders and buttons would make clearer what the slider does, and what it will affect when changed. For example, the pressure residual tolerance could be annotated to explain that increasing it will make the simulation run faster but possibly be less accurate.

18.3 Visualisation

18.3.1 Automatic Contour Tolerance

At the moment, contour lines are based on a contour tolerance (which is the greatest difference from the ideal value of the contour line from the points it is drawn between). Loosely, this defines how closely the contour line fits the data. This is a bit of a balance, however, since a contour tolerance that is too low will lead to lines not being drawn, and one that is too high will lead to inaccurate contours. Therefore, if contour tolerance could be selected automatically such that it is as low as possible without losing lines, this would remove this unintuitive parameter.

18.3.2 Automatic Min/Max Sliders

Ben, in the interview, identified that setting the visualisation min and max automatically would make the program easier to work with. This would require finding the min and max value of the field being visualised. As this is quite a slow process in comparison to the rest of the computations, this could be handled using multithreading so that the colouring calculations and the min/max calculations occur concurrently. This could also only be run every 10 iterations, similar to the residual calculation for the GPU backend, to decrease the effect of this on the overall speed of the program.

18.4 Obstacle Defining

18.4.1 Click-And-Drag Node System

The click-and-drag node system, in its current state, is difficult to work with as small changes to one control point can affect the entire structure. Also, it is possible to make the structure unphysical with large loops that result from the polar coordinates. Instead, other spline drawing methods, such as the Catmul-Rom splines mentioned in [3]. Rather than the curvature of the whole structure, these splines use the directions of the surrounding control points which makes local changes less likely to affect the whole structure.

18.4.2 Parsing of Common File Types

The binary file (and its creation using constraints) is difficult to integrate into an actual workflow for the Greenpower team. Rather, it would have been preferable to accept common file types from design programs used by the Greenpower team to allow them to use existing designs without having to redraw them in the program, or define them with constraints.

18.4.3 File Maker GUI

With more time, a GUI could have been implemented on top of the CLI program that currently provides the interface for the file maker. This could make the file maker simpler to work with (which was identified in the interview), and could include a preview of what the obstacle will look like when drawn.

Bibliography

[1] Michael Griebel, Thomas Dornseifer, and Tilman Neunhoeffer. *Numerical Simulation in Fluid Dynamics*. Society for Industrial and Applied Mathematics, 1998. DOI: `10.1137/1.9780898719703`. eprint: `https://epubs.siam.org/doi/pdf/10.1137/1.9780898719703`. URL: `https://epubs.siam.org/doi/abs/10.1137/1.9780898719703`.

[2] Chaoyang Zhang et al. “Parallel SOR Iterative Algorithms and Performance Evaluation on a Linux Cluster”. In: (2005). URL: `https://apps.dtic.mil/sti/pdfs/ADA449212.pdf`. (accessed 15.11.2023).

[3] Donald H. House. “Computer Graphics: Spline Curves”. In: (2014). URL: `https://people.computing.clemson.edu/~dhouse/courses/405/notes/splines.pdf`. (accessed 20.11.2023).

[4] Stefan Bartels. *Interactive Simulations with Navier-Stokes Equations on many-core Architectures*. URL: `https://www5.in.tum.de/pub/bartels_idp_14.pdf`. (accessed 20.06.2023).

Appendix A

Glossary of Terms

CFD Computational Fluid Dynamics: the study of simulating fluid flow using computational power to solve equations iteratively.

Continuous Continuous, in the context of maths, refers to the fact that a function, equation or problem can be solved or evaluated for any point that exists inside a given boundary.

Cubic spline A spline (see definition below) that is defined by a cubic polynomial - a polynomial in the form $y = ax^3+bx^2+cx+d$, where the constants a, b, c and d determine the shape of the polynomial.

Derivative An expression that defines the *rate of change* of a function. This expression can then be evaluated at a specific point to find the rate of change at that point

Differential equation An equation which relates the rate of change of a parameter. For example, a differential equation could define the velocity of an object (its rate of change of displacement) with respect to time. Differential equations are solved via integration, which is trivial for the velocity example here but non-trivial for the equations underpinning CFD.

Discretise The process of translating a continuous problem into one defined for a finite set of points (discrete domain)

Domain (of a function) The set of values for which a function can be evaluated; the set of inputs to a function.

GPU The Graphical Processing Unit: a part of the computer that handles the processing needed to draw graphics. In recent years, these have become ever more advanced, such that their ability to perform many computations in parallel has seen their use in non-graphical applications such as CFD.

Laminar flow Flow of fluid which is smooth, with uniform pressures and velocities. Antonym of turbulent in this context.

Navier-Stokes equations The underlying equations governing fluid dynamics. These are a set of 2 differential equations that arise from conservation of mass and conservation of energy.

Numerical method A method of solving an equation by repeatedly performing a mathematical operation to get closer to a solution each time. Often used for equations that cannot be solved any other way.

Partial derivative A derivative for a function that has a domain of more than one dimension. The derivative is evaluated with respect to only one variable (dimension), with the other variable treated as constant. Written using the ∂ notation.

Polar coordinates A coordinate system that specifies an angle and distance from an initial point, in contrast to the more common rectangular coordinates which specify 2 different lengths in perpendicular directions.

Postfix notation In contrast to the standard mathematical notation in which operators sit between their operands, such as $2+7$, in postfix notation the operator sits after its operands, such as 27+. This eliminates the need for parenthesis and the order of operations, and is therefore easier for computers to parse.

RAII Stands for Resource Acquisition Is Initialisation: the principle that a resource should be wholly owned by an object. This includes instantiation, use, and deallocation and deletion. The principle aims to avoid memory leaks by ensuring that the creation and deletion of a resource are handled by the same object, to avoid control flows that allocate memory but fail to free it.

Spline A curve that fits a set of points. The points are named control points and are often configurable by the user.

Appendix B

Explanation of Mathematical Symbols

Below is an explanation of mathematical symbols, in the order that they appear: δx: The small step in the x direction, used when discretising.

δy: As above, in the y direction.

u, v: The vectors for velocity in the x and y directions, respectively.

v_{in}: Fluid inflow speed

p: The 2D array of pressures

p_0: The value of pressure infinitely far away from the obstacle - a baseline pressure value.

Re: The Reynold's number - a ratio of inertial forces to external forces. Related to fluid viscosity and speed.

g_x, g_y: The external forces acting on the fluid in the x and y directions.

$\frac{dy}{dx}$: The function that computes the change in a function, y, at a given input x.

$\frac{\partial u}{\partial t}$: A partial derivative. As above, but the ∂ symbol shows that the function u has more than one input, and the change calculated is only with respect to one of the inputs, t.

n: An arbitrary iteration variable, incremented in the time-stepping loop.

$u^{(n)}$, $u^{(n+1)}$, $v^{(n)}$, etc.: The value of the given variable at a given point in space, n, or the following timestep, $n + 1$.

$u_{i,j}$,$v_{i+1,j}$, $p_{i,j+1}$, etc.: The value of the given variable at a given point in space, with coordinates, for example, (i, j).

δt: The size of the timesteps.

τ: A safety factor for the timestep, between 0 and 1. Closer to 0, the simulation is more accurate but will take longer. The opposite is true for close to 1.

χ: A parameter defining the amount of fluid slip at a boundary. $\chi = 0$ gives completely no-slip, $\chi = 1$ gives completely free-slip (no friction).

F_d: The total aerodynamic drag force on an object.

F_p: The drag force on an object due to pressure differential.

F_μ: The drag force on an object due to friction along its surface.

c_d: The drag coefficient for an object.

ρ: The density of a fluid.

$A_{\boldsymbol{i}}$: The area of an object as projected in the same horizontal direction as the fluid. Due to the simulation being 2D, this is a 1D quantity.

$\int_S$: The integral over the surface of an obstacle.

$\hat{\boldsymbol{\tau}}$: The unit vector in the direction of shear stresses acting at a point. Tangential to the obstacle surface.

$\hat{\boldsymbol{n}}$: The unit vector in the direction normal to the obstacle surface at a point.

$\hat{\boldsymbol{i}}$: The unit vector in the direction of overall fluid flow.

τ_w: The magnitude of the shear stresses acting on the obstacle at a point.

$\boldsymbol{V}$: The fluid flow, made up of its u and v components

d: The distance of a point from another point on the obstacle surface

Appendix C

Pipe Protocol Specification

C.1 Requirements and Definitions

A *field* is defined as the 2D array of one variable. One *field length* is defined as the size of the simulation domain, in cells. This is equal to $i_{max} \times j_{max}$.

There exist 3 types of data that will be sent down the pipe. The first is fixed-length data, where the length of the data is a multiple of the field length (an example would be the obstacle domain data sent by the fronted to the backend). The second is continuous data, which is a continuous stream of field data sent by the backend during simulation. The length of this data is indeterminate and continues until stopped. The third is single-variable data, which is made up of, indeed, single variables. This is used for sending user-defined parameters from the frontend to the backend. The length (in bytes) of this data must be specified before transmission, as with fixed-length data.

There also must exist control commands, so that the server and client understand what the data being sent actually means. Control commands must be understood separately to data, and therefore the control commands must specify what is coming next - another control command or a certain length of data.

C.2 General Structure of Control Bytes

This protocol defines control sequences as occupying a single byte. The general structure is as follows:

Control commands are split into 4 categories - STATUS, REQUEST, MARKER and ERROR. Since there are 4 categories, the first 2 bits of a control byte show which category of command is being sent. Each category also deals with parameters differently.

- STATUS bytes begin with 00 and provide information or commands to the other application to do with current program state - STOP commands, acknowledgement commands and handshaking.
- REQUEST bytes begin with 01 and are sent by the frontend to request data to be calculated and sent from the client.
- MARKER bytes are used when transmitting single-variable and continuous data, to demark start and end of fields, and where timestepping occurs in a continuous stream.
- ERROR bytes define error codes and can be sent by either application.

C.3 Precise Specification

Binary representation	Name	Explanation	Notes
00001000	HELLO	Sent when performing a handshake after initial connection	After a HELLO byte is usually sent the field dimensions, iMax and jMax
00010000	BUSY	Sent if the sender is already calculating something and cannot send data	Currently unused
00011000	OK	Acknowledgement of data sent	Sent after fixed-length data is received correctly
00100000	STOP	Stops sending and calculation for a continuous data stream	
00101000	CLOSE	Tells the backend to shut down	Sent when sending application wants to shut down
010yyyyy	FIXLENREQ	Request for transmission of fixed-length field	y bits are each a field, which allows for multiple fields to be requested if more than one y bit is set
011yyyyy	CONTREQ	Request for transmission of continuous stream of data	y parameters are the same as above
10000000	ITERSTART	Marks the start of a timestep iteration	Goes before a FLDSTART
10001000	ITEREND	Marks the end of a timestep iteration	Goes after a FLDEND
10010zzz	FLDSTART	Start of field data	z bits are the field transmitted (different system to y bits)
10011zzz	FLDEND	End of field data	z bits as above
1010pppp	PRMSTART	Start of parameter transmission	p bits are the parameter
1011pppp	PRMEND	End of parameter transmission	p bits as above

Binary representation	Name	Explanation	Notes
11000001	BADREQ	Request not understood	Generic error code
11000010	BADPARAM	Parameter combination makes no sense	This is referring the extra bits (x, y, p)
11000011	INTERNAL	Fatal error has occured, stops the other application	
11000100	TIMEOUT	Sender of a REQUEST had to wait too long	Currently unused
11000101	BADTYPE	Wrong number of bytes received for the data type that was expected	Used in transmission of single-variable data
11000110	BADLEN	Length of data was not as expected	This is for fixed-length data, or the difference between markers in continuous data

The bits of each of these codes will be numbered from 0 as the leftmost and most significant bit, and 7 as the rightmost and least significant bit.

For REQUEST codes, y bits are defined as follows:

Bit	Field	Symbol
3	Horizontal velocity	u
4	Vertical velocity	v
5	Pressure	p
6	Stream function	ϕ

If the relevant bit is set, this means that that field is requested. The backend will respond with the data in the order of the bits.

For MARKER codes, z bits are defined as follows:

Bits	Field	Symbol	Data type
001	Horizontal velocity	u	REAL
010	Vertical velocity	v	REAL
011	Pressure	p	REAL
100	Stream function	ϕ	REAL
101	Obstacle data	N/A	BOOL

Note that z bits and y bits are defined differently, both because there are required to be fewer z bits (so there can be more types of marker) and because a REQUEST can request multiple fields, but a MARKER can only reference one field.

For p bits (specifically for PARAMSTART and PARAMEND markers), parameters are defined as follows:

Bits	Symbol	Full name	Data type
0001	iMax	Max cell index in x direction	int
0010	jMax	Max cell index in y direction	int
0011	Width	Width of simulation space	REAL
0100	Height	Height of simulation space	REAL
0101	τ	Timestep safety factor	REAL
0110	ω	Pressure relaxation parameter	REAL
0111	r_{max}	Pressure residual tolerance	REAL
1000	iterMax	Max number of pressure iterations	int
1001	Re	Fluid reynolds number	REAL
1010	v_{in}	Fluid inflow speed	REAL
1011	χ	Obstacle surface friction	REAL
1100	μ	Fluid dynamic viscosity	REAL
1101	ρ	Fluid density	REAL
1110	c_d	Drag coefficient	REAL
1111		Unused	

C.4 Sample Usage

Below is the most common control flow between the two programs, including parameter and obstacle send, and the continuous transmission of pressure and stream function data.

Binary representation	Name	Sender	Description
00001000	HELLO	Frontend	Frontend initiates handshake, allows backend to dictate field length
00001000	HELLO	Backend	Backend responds to handshake
10100001	PRMSTART IMAX	Backend	Backend starts to send iMax
[4 bytes]	iMax	Backend	Backend sends the parameter value
10101001	PRMEND IMAX	Backend	Backend finishes sending iMax
10100010	PRMSTART JMAX	Backend	Backend starts to send jMax
[4 bytes]	jMax	Backend	Backend sends the parameter value
10101010	PRMEND JMAX	Backend	Backend finishes sending jMax
00010000	OK	Frontend	Frontend acknowledges field dimensions
10101010	PRMSTART INVEL	Frontend	Frontend starts to send parameters (input velocity here for demonstration)
[4 bytes]	inVel	Frontend	Frontend sends parameter value
10111010	PRMEND INVEL	Frontend	Frontend finishes sending parameters
00010000	OK	Backend	Backend acknowledges parameters
10010101	FLDSTART OBST	Frontend	Frontend starts to send obstacle data
[obstacle data]	obstacles	Frontend	Frontend sends boolean obstacle data
10011101	FLDEND OBST	Frontend	Frontend finishes sending obstacle data
00010000	OK	Backend	Backend acknowledges obstacle data
01100110	CONTREQ PRES STRM	Frontend	Frontend requests pressure and stream function
00010000	OK	Backend	Backend acknowledges request and starts executing

The backend would then continuously send the field data, demarked by ITERSTART and ITEREND, and each field demarked by FLDSTART and FLDEND. After each timestep, the backend waits for a response from the frontend. This will either be an OK byte, a MARKER byte in case more parameters are to be sent, or a STOP or ERROR byte, in which case the backend stops executing.

Appendix D

Source code

D.1 Project code structure

```
NEAFluidDynamics
├ BackendLib
│ ├── BackendCoordinator.cpp (p85)
│ ├── BackendCoordinator.h (p90)
│ ├── BackendLib.cpp (p91)
│ ├── Definitions.h (p91)
│ ├── Flags.cpp (p93)
│ ├── Flags.h (p93)
│ ├── Init.cpp (p94)
│ ├── Init.h (p95)
│ ├── pch.cpp (p95)
│ ├── pch.h (p96)
│ ├── PipeConstants.h (p96)
│ ├── PipeManager.cpp (p97)
│ ├── PipeManager.h (p102)
│ ├── Solver.cpp (p105)
│ └── Solver.h (p106)
├ CPUBackend
│ ├── Boundary.cpp (p109)
│ ├── Boundary.h (p112)
│ ├── Computation.cpp (p112)
│ ├── Computation.h (p121)
│ ├── CPUBackend.cpp (p122)
│ ├── CPUSolver.cpp (p124)
│ ├── CPUSolver.h (p126)
│ ├── DiscreteDerivatives.cpp (p127)
│ ├── DiscreteDerivatives.h (p130)
│ ├── Drag.cpp (p130)
│ └── Drag.h (p135)
├ FileMakerBackend
│ ├── Constraint.cs (p136)
│ ├── FileMaker.cs (p136)
│ └── Program.cs (p144)
├ FileMakerCLI
│ ├── ConstraintParser.cs (p139)
│ ├── Program.cs (p144)
│ ├── ResizableStack.cs (p145)
│ └── RPNConstraint.cs (p147)
├ GPUBackend
│ ├── Boundary.cu (p148)
│ ├── Boundary.cuh (p151)
│ ├── Computation.cu (p153)
│ ├── Computation.cuh (p159)
│ ├── Definitions.cuh (p160)
│ ├── DiscreteDerivatives.cu (p161)
│ ├── DiscreteDerivatives.cuh (p164)
│ ├── Drag.cu (p165)
│ ├── Drag.cuh (p167)
│ ├── DragCalculator.cu (p167)
│ ├── DragCalculator.cuh (p171)
│ ├── GPUSolver.cu (p172)
│ ├── GPUSolver.cuh (p179)
│ ├── kernel.cu (p181)
│ ├── PressureComputation.cu (p182)
│ ├── PressureComputation.cuh (p187)
│ ├── ReductionKernels.cu (p187)
│ └── ReductionKernels.cuh (p193)
├ UserInterface
│ ├── App.xaml (p194)
│ ├── App.xaml.cs (p195)
│ ├── AssemblyInfo.cs (p198)
│ ├ Converters
│ │ ├── BoolToTickPlacement.cs (p198)
│ │ ├── PolarListToRectList.cs (p199)
│ │ ├── RectangularToPolar.cs (p200)
│ │ ├── SignificantFigures.cs (p202)
│ │ ├── UnitClasses.cs (p203)
│ │ └── Units.cs (p206)
│ ├ HelperClasses
│ │ ├── BackendManager.cs (p208)
│ │ ├── CircularQueue.cs (p219)
│ │ ├── Commands.cs (p220)
│ │ ├── DefaultParameters.cs (p227)
│ │ ├── MovingAverage.cs (p228)
│ │ ├── ObstacleHolder.cs (p228)
│ │ ├── ParameterChangedEventArgs.cs (p230)
│ │ ├── ParameterHolder.cs (p231)
│ │ ├── PipeConstants.cs (p236)
│ │ ├── PipeManager.cs (p238)
│ │ ├── PolarPoint.cs (p244)
│ │ ├── PolarSplineCalculator.cs (p246)
│ │ ├── Queue.cs (p250)
│ │ ├── ResizableLinearQueue.cs (p251)
│ │ ├── UnitChangedEventArgs.cs (p253)
│ │ └── UnitHolder.cs (p253)
│ ├ HelperControls
│ │ ├── ObstacleCell.cs (p255)
│ │ ├── ResizableCentredTextBox.xaml (p255)
│ │ ├── ResizableCentredTextBox.xaml.cs (p256)
│ │ ├── SliderWithValue.xaml (p256)
│ │ ├── SliderWithValue.xaml.cs (p257)
│ │ ├── UnitConversionPanel.xaml (p261)
│ │ ├── UnitConversionPanel.xaml.cs (p262)
│ │ └── VisualPoint.cs (p265)
│ ├ ViewModels
│ │ ├── AdvancedParametersVM.cs (p267)
│ │ ├── ConfigScreenVM.cs (p269)
│ │ ├── SimulationScreenVM.cs (p271)
│ │ └── ViewModel.cs (p284)
│ ├ Views
│ │ ├── AdvancedParameters.xaml (p285)
│ │ ├── AdvancedParameters.xaml.cs (p287)
│ │ ├── ConfigScreen.xaml (p287)
│ │ ├── ConfigScreen.xaml.cs (p289)
│ │ ├── MainWindow.xaml (p289)
│ │ ├── MainWindow.xaml.cs (p290)
│ │ ├── PopupWindow.xaml (p290)
│ │ ├── PopupWindow.xaml.cs (p290)
│ │ ├── SimulationScreen.xaml (p291)
│ │ ├── SimulationScreen.xaml.cs (p295)
│ │ ├── VisualisationControl.xaml (p300)
│ │ └── VisualisationControl.xaml.cs (p300)
├ Visualisation
│ ├── contourShader.frag (p304)
│ ├── contourShader.vert (p304)
│ ├── fieldShader.frag (p304)
│ ├── fieldShader.vert (p305)
│ ├── GLHelper.cs (p306)
│ └── ShaderManager.cs (p312)
```

D.2 Source code

BackendCoordinator.cpp

```
#include "pch.h"
#include "BackendCoordinator.h"
#include "PipeConstants.h"
#include <iostream>
#define OBSTACLES

void BackendCoordinator::UnflattenArray(bool** pointerArray, bool* flattenedArray,
↪   int length, int divisions) {
    for (int i = 0; i < length / divisions; i++) {

        memcpy(
            pointerArray[i],                  // Destination address - address at ith
            ↪  pointer
            flattenedArray + i * divisions, // Source start address - move (i *
            ↪  divisions) each iteration
            divisions * sizeof(bool)        // Bytes to copy - divisions
        );

    }
}

void BackendCoordinator::HandleRequest(BYTE requestByte) {
    std::cout << "Starting execution of timestepping loop\n";
    if ((requestByte & ~PipeConstants::Request::PARAMMASK) ==
    ↪   PipeConstants::Request::CONTREQ) {
        if (requestByte == PipeConstants::Request::CONTREQ) {
            pipeManager.SendByte(PipeConstants::Error::BADPARAM);
            std::cerr << "Server sent a blank request, exiting";
            return;
        }

        bool hVelWanted = requestByte & PipeConstants::Request::HVEL;
        bool vVelWanted = requestByte & PipeConstants::Request::VVEL;
        bool pressureWanted = requestByte & PipeConstants::Request::PRES;
        bool streamWanted = requestByte & PipeConstants::Request::STRM;

        bool closeRequested = false;
        pipeManager.SendByte(PipeConstants::Status::OK); // Send OK to say backend
        ↪   is set up and about to start executing

        int iteration = 0;
        REAL cumulativeTimestep = 0;
        solver->PerformSetup();
        while (!closeRequested) {
            std::cout << "Iteration " << iteration << ", " << cumulativeTimestep <<
            ↪  " seconds passed. \n";
            pipeManager.SendByte(PipeConstants::Marker::ITERSTART);

            solver->Timestep(cumulativeTimestep);
```

```
int iMax = solver->GetIMax();
int jMax = solver->GetJMax();

if (hVelWanted) {
    pipeManager.SendByte(PipeConstants::Marker::FLDSTART |
    ↪  PipeConstants::Marker::HVEL);
    pipeManager.SendField(solver->GetHorizontalVelocity(), iMax * jMax);
    pipeManager.SendByte(PipeConstants::Marker::FLDEND |
    ↪  PipeConstants::Marker::HVEL);
}
if (vVelWanted) {
    pipeManager.SendByte(PipeConstants::Marker::FLDSTART |
    ↪  PipeConstants::Marker::VVEL);
    pipeManager.SendField(solver->GetVerticalVelocity(), iMax * jMax);
    pipeManager.SendByte(PipeConstants::Marker::FLDEND |
    ↪  PipeConstants::Marker::VVEL);
}
if (pressureWanted) {
    pipeManager.SendByte(PipeConstants::Marker::FLDSTART |
    ↪  PipeConstants::Marker::PRES);
    pipeManager.SendField(solver->GetPressure(), iMax * jMax);
    pipeManager.SendByte(PipeConstants::Marker::FLDEND |
    ↪  PipeConstants::Marker::PRES);
}
if (streamWanted) {
    pipeManager.SendByte(PipeConstants::Marker::FLDSTART |
    ↪  PipeConstants::Marker::STRM);
    pipeManager.SendField(solver->GetStreamFunction(), iMax * jMax);
    pipeManager.SendByte(PipeConstants::Marker::FLDEND |
    ↪  PipeConstants::Marker::STRM);
}

pipeManager.SendByte(PipeConstants::Marker::PRMSTART |
↪  PipeConstants::Marker::DRAGCOEF);
pipeManager.SendReal(solver->GetDragCoefficient());
pipeManager.SendByte(PipeConstants::Marker::PRMEND |
↪  PipeConstants::Marker::DRAGCOEF);

pipeManager.SendByte(PipeConstants::Marker::ITEREND);

BYTE receivedByte = pipeManager.ReadByte();
if (receivedByte == PipeConstants::Status::STOP) { // Stop means just
↪  wait for the next read
    pipeManager.SendByte(PipeConstants::Status::OK);
    std::cout << "Backend paused.\n";
    receivedByte = pipeManager.ReadByte();
}
if (receivedByte == PipeConstants::Status::CLOSE || receivedByte ==
↪  PipeConstants::Error::INTERNAL) {
    closeRequested = true; // Stop if requested or the frontend fatally
    ↪  errors
}
else { // Anything other than a CLOSE request
```

```
                while ((receivedByte & ~PipeConstants::Marker::PRMMASK) ==
                ↪  PipeConstants::Marker::PRMSTART || receivedByte ==
                ↪  (PipeConstants::Marker::FLDSTART | PipeConstants::Marker::OBST))
                ↪  { // While the received byte is a PRMSTART or obstacle send...
                    ReceiveData(receivedByte); // ...pass the received byte to
                    ↪  ReceiveData to handle parameter reading...
                    receivedByte = pipeManager.ReadByte(); // ...then read the next
                    ↪  byte
                }
                if (receivedByte != PipeConstants::Status::OK) { // Require an OK at
                ↪  the end, whether parameters were sent or not
                    std::cerr << "Server sent malformed data\n";
                    pipeManager.SendByte(PipeConstants::Error::BADREQ);
                }
            }

            iteration++;
        }
        std::cout << "Backend stopped.\n";

        pipeManager.SendByte(PipeConstants::Status::OK); // Send OK then stop
        ↪  executing

    }
    else { // Only continuous requests are supported
        std::cerr << "Server sent an unsupported request\n";
        pipeManager.SendByte(PipeConstants::Error::BADREQ);
    }
}



void BackendCoordinator::ReceiveObstacles()
{
    int iMax = solver->GetIMax();
    int jMax = solver->GetJMax();
    bool* obstaclesFlattened = new bool[(iMax + 2) * (jMax + 2)]();
    pipeManager.ReceiveObstacles(obstaclesFlattened, iMax + 2, jMax + 2);
    bool** obstacles = solver->GetObstacles();
    UnflattenArray(obstacles, obstaclesFlattened, (iMax + 2) * (jMax + 2), jMax +
    ↪  2);
    delete[] obstaclesFlattened;
}

void BackendCoordinator::ReceiveParameters(const BYTE parameterBits,
↪  SimulationParameters& parameters)
{
    if (parameterBits == PipeConstants::Marker::ITERMAX) {
        parameters.pressureMaxIterations = pipeManager.ReadInt();
    }
    else {
        REAL parameterValue = pipeManager.ReadReal(); // All of the other possible
        ↪  parameters have the data type REAL, so read the pipe and convert it to a
        ↪  REAL beforehand
        switch (parameterBits) { // AND the start marker with the parameter mask to
        ↪  see which parameter is sent
        case PipeConstants::Marker::WIDTH:
            parameters.width = parameterValue;
            break;
        case PipeConstants::Marker::HEIGHT:
            parameters.height = parameterValue;
            break;
        case PipeConstants::Marker::TAU:
            parameters.timeStepSafetyFactor = parameterValue;
            break;
        case PipeConstants::Marker::OMEGA:
            parameters.relaxationParameter = parameterValue;
            break;
        case PipeConstants::Marker::RMAX:
            parameters.pressureResidualTolerance = parameterValue;
            break;
        case PipeConstants::Marker::REYNOLDS:
            parameters.reynoldsNo = parameterValue;
            break;
        case PipeConstants::Marker::INVEL:
            parameters.inflowVelocity = parameterValue;
            break;
        case PipeConstants::Marker::CHI:
            parameters.surfaceFrictionalPermissibility = parameterValue;
            break;
        case PipeConstants::Marker::MU:
            parameters.dynamicViscosity = parameterValue;
            break;
        case PipeConstants::Marker::DENSITY:
            parameters.fluidDensity = parameterValue;
            break;
        default:
            break;
        }
    }
}

void BackendCoordinator::ReceiveData(BYTE startMarker) {
    if (startMarker == (PipeConstants::Marker::FLDSTART |
    ↪   PipeConstants::Marker::OBST)) { // Obstacles have a separate handler
        ReceiveObstacles();
        solver->ProcessObstacles();
    }
    else if ((startMarker & ~PipeConstants::Marker::PRMMASK) ==
    ↪   PipeConstants::Marker::PRMSTART) { // Check if startMarker is a PRMSTART by
    ↪   ANDing it with the inverse of the parameter mask
        BYTE parameterBits = startMarker & PipeConstants::Marker::PRMMASK;
        SimulationParameters parameters = solver->GetParameters();
        ReceiveParameters(parameterBits, parameters);

        if (pipeManager.ReadByte() != (PipeConstants::Marker::PRMEND |
        ↪  parameterBits)) { // Need to receive the corresponding PRMEND
```

```
            std::cerr << "Server sent malformed data\n";
            pipeManager.SendByte(PipeConstants::Error::BADREQ);
        }

        solver->SetParameters(parameters);
        pipeManager.SendByte(PipeConstants::Status::OK); // Send an OK to say
        ↪  parameters were received correctly
    }
    else {
        std::cerr << "Server sent unsupported data\n";
        pipeManager.SendByte(PipeConstants::Error::BADREQ); // Error if the start
        ↪  marker was unrecognised.
    }
}

void BackendCoordinator::SetDefaultParameters(SimulationParameters& parameters) {
    parameters.width = 1;
    parameters.height = 1;
    parameters.timeStepSafetyFactor = (REAL)0.5;
    parameters.relaxationParameter = (REAL)1.7;
    parameters.pressureResidualTolerance = 2;
    parameters.pressureMinIterations = 5;
    parameters.pressureMaxIterations = 1000;
    parameters.reynoldsNo = 10000;
    parameters.dynamicViscosity = (REAL)0.00001983;
    parameters.fluidDensity = (REAL)1.293;
    parameters.inflowVelocity = 1;
    parameters.surfaceFrictionalPermissibility = 0;
    parameters.bodyForces.x = 0;
    parameters.bodyForces.y = 0;
}

BackendCoordinator::BackendCoordinator(int iMax, int jMax, std::string pipeName,
↪  Solver* solver)
    : pipeManager(pipeName), solver(solver)
{
    SimulationParameters parameters = SimulationParameters();
    SetDefaultParameters(parameters);
    solver->SetParameters(parameters);
}

int BackendCoordinator::Run() {
    pipeManager.Handshake(solver->GetIMax(), solver->GetJMax());
    std::cout << "Handshake completed ok\n";

    bool closeRequested = false;

    while (!closeRequested) {
        std::cout << "In read loop\n";
        BYTE receivedByte = pipeManager.ReadByte();
        switch (receivedByte & PipeConstants::CATEGORYMASK) { // Gets the category
        ↪  of control byte
        case PipeConstants::Status::GENERIC: // Status bytes
```

```
            switch (receivedByte & ~PipeConstants::Status::PARAMMASK) {
            case PipeConstants::Status::HELLO:
            case PipeConstants::Status::BUSY:
            case PipeConstants::Status::OK:
            case PipeConstants::Status::STOP:
                std::cerr << "Server sent a status byte out of sequence, request not
                ↪  understood\n";
                pipeManager.SendByte(PipeConstants::Error::BADREQ);
                break;
            case PipeConstants::Status::CLOSE:
                closeRequested = true;
                pipeManager.SendByte(PipeConstants::Status::OK);
                std::cout << "Backend closing...\n";
                break;
            default:
                std::cerr << "Server sent a malformed status byte, request not
                ↪  understood\n";
                pipeManager.SendByte(PipeConstants::Error::BADREQ);
                break;
            }
            break;
        case PipeConstants::Request::GENERIC: // Request bytes have a separate
        ↪   handler
            HandleRequest(receivedByte);
            break;
        case PipeConstants::Marker::GENERIC: // So do marker bytes
            ReceiveData(receivedByte);
            break;
        default: // Error bytes
            break;
        }
    }
    return 0;
}
```

BackendCoordinator.h

```
#ifndef BACKEND_COORDINATOR_H
#define BACKEND_COORDINATOR_H

#include "pch.h"
#include "Solver.h"
#include "PipeManager.h"

class BackendCoordinator
{
private:
        PipeManager pipeManager;
        Solver* solver;

        void UnflattenArray(bool** pointerArray, bool* flattenedArray, int length,
        ↪   int divisions);
        void HandleRequest(BYTE requestByte);
        void ReceiveObstacles();
        void ReceiveParameters(const BYTE parameterBits, SimulationParameters&
        ↪  parameters);
        void ReceiveData(BYTE startMarker);
        void SetDefaultParameters(SimulationParameters& parameters);

public:
        /// <summary>
        /// Constructor - sets up field dimensions and pipe name.
        /// </summary>
        /// <param name="iMax">The width, in cells, of the simulation domain
        ↪  excluding boundary cells.</param>
        /// <param name="jMax">The height, in cells, of the simulation domain
        ↪  excluding boundary cells.</param>
        /// <param name="pipeName">The name of the named pipe to use for
        ↪  communication with the frontend.</param>
        /// <param name="solver">The instantiated solver to use.</param>
        BackendCoordinator(int iMax, int jMax, std::string pipeName, Solver*
        ↪  sovler);

        /// <summary>
        /// Main method for BackendCoordinator class, which handles all the data
        ↪  flow and computation.
        /// </summary>
        /// <returns>An exit code to be directly returned by the program</returns>
        int Run();
};

#endif // !BACKEND_COORDINATOR_H
```

BackendLib.cpp

```
// BackendLib.cpp : Defines the functions for the static library.
//

#include "pch.h"
```

Definitions.h

```
#ifndef DEFINITIONS_H
#define DEFINITIONS_H

typedef float REAL;
typedef unsigned __int8 BYTE;

// Definitions for boundary cells. For the last 5 bits, format is [self] [north]
↪  [east] [south] [west]
// Where 1 means the corresponding cell is fluid, and 0 means the corresponding cell
↪  is obstacle.
// Boundary cells are defined as obstacle cells with fluid on 1 side or 2 adjacent
↪  sides
// For fluid cells, XOR the corresponding inverse boundary with FLUID.
constexpr BYTE B_N   = 0b00001000;
constexpr BYTE B_NE  = 0b00001100;
constexpr BYTE B_E   = 0b00000100;
constexpr BYTE B_SE  = 0b00000110;
constexpr BYTE B_S   = 0b00000010;
constexpr BYTE B_SW  = 0b00000011;
constexpr BYTE B_W   = 0b00000001;
constexpr BYTE B_NW  = 0b00001001;
constexpr BYTE OBS   = 0b00000000;
constexpr BYTE FLUID = 0b00011111;

// Constants used for parsing of flags.
constexpr BYTE SELF  = 0b00010000; // SELF bit
constexpr BYTE NORTH = 0b00001000; // NORTH bit
constexpr BYTE EAST  = 0b00000100; // EAST bit
constexpr BYTE SOUTH = 0b00000010; // SOUTH bit
constexpr BYTE WEST  = 0b00000001; // WEST bit

constexpr BYTE SELFSHIFT  = 4; // Amount to shift for SELF bit at LSB.
constexpr BYTE NORTHSHIFT = 3; // Amount to shift for NORTH bit at LSB.
constexpr BYTE EASTSHIFT  = 2; // Amount to shift for EAST bit at LSB.
constexpr BYTE SOUTHSHIFT = 1; // Amount to shift for SOUTH bit at LSB.
constexpr BYTE WESTSHIFT  = 0; // Amount to shift for WEST bit at LSB.

// Constants used for dealing with drag coefficient.
constexpr REAL DIAGONAL_CELL_DISTANCE = (REAL)1.41421356237; // Sqrt 2
constexpr REAL PRESSURE_CONVERSION = (REAL)0.180972; // Conversion ratio for
↪  pressure.
constexpr REAL VISCOSITY_CONVERSION = (REAL)23.62593; // Conversion ratio for
↪  viscosity.

struct DoubleField
{
        REAL** x;
        REAL** y;
        DoubleField(REAL** x, REAL** y) : x(x), y(y) {}
        DoubleField() : x(nullptr), y(nullptr) {}
};

struct DoubleReal
{
        REAL x;
        REAL y;
        DoubleReal(REAL x, REAL y) : x(x), y(y) {}
        DoubleReal() : x(0), y(0) {}
};

struct SimulationParameters
{
        REAL width;
        REAL height;
        REAL timeStepSafetyFactor;
        REAL relaxationParameter;
        REAL pressureResidualTolerance;
        int pressureMinIterations;
        int pressureMaxIterations;
        REAL reynoldsNo;
        REAL dynamicViscosity;
        REAL fluidDensity;
        REAL inflowVelocity;
        REAL surfaceFrictionalPermissibility;
        DoubleReal bodyForces;
};

struct ThreadStatus
{
        bool running;
        bool startNextIterationRequested;
        bool stopRequested;

        ThreadStatus() : running(false), startNextIterationRequested(false),
        ↪  stopRequested(false) {} // Constructor just sets everything to false.
};

#endif
```

Flags.cpp

```
#include "pch.h"
#include "Flags.h"

void SetFlags(bool** obstacles, BYTE** flags, int xLength, int yLength) {
    for (int i = 1; i < xLength - 1; i++) {
        for (int j = 1; j < yLength - 1; j++) {
            flags[i][j] = ((BYTE)obstacles[i][j] << 4) + ((BYTE)obstacles[i][j + 1]
            ↪  << 3) + ((BYTE)obstacles[i + 1][j] << 2) + ((BYTE)obstacles[i][j -
            ↪  1] << 1) + (BYTE)obstacles[i - 1][j]; // 5 bits in the format: self,
            ↪  north, east, south, west.
        }
    }
}

// Counts number of fluid cells in the region [1,iMax]x[1,jMax]
int CountFluidCells(BYTE** flags, int iMax, int jMax) {
    int count = 0;
    for (int i = 0; i <= iMax; i++) {
        for (int j = 0; j <= jMax; j++) {
            count += flags[i][j] >> 4; // This will include only the "self" bit,
            ↪  which is one for fluid cells and 0 for boundary and obstacle cells.
        }
    }
    return count;
}
```

Flags.h

```
#ifndef FLAGS_H

#include "pch.h"
```

```
void SetFlags(bool** obstacles, BYTE** flags, int xLength, int yLength);

int CountFluidCells(BYTE** flags, int iMax, int jMax);

#endif // !FLAGS_H
```

Init.cpp

```
#include "pch.h"
#include "Init.h"
#include <fstream>
#include <iostream>
#include <vector>
#include <algorithm>

REAL** MatrixMAlloc(int xLength, int yLength) {

        // Create array of pointers pointing to more arrays
        REAL** matrix = new REAL* [xLength];

        // Create the arrays inside each outer array
        for (int i = 0; i < xLength; ++i) {
                matrix[i] = new REAL[yLength]();
        }

        return matrix;
}

BYTE** FlagMatrixMAlloc(int xLength, int yLength) {

        // Create array of pointers pointing to more arrays
        BYTE** matrix = new BYTE * [xLength];

        // Create the arrays inside each outer array
        for (int i = 0; i < xLength; ++i) {
                matrix[i] = new BYTE[yLength]();
        }

        return matrix;
}

bool** ObstacleMatrixMAlloc(int xLength, int yLength) {
        // Create array of pointers pointing to more arrays
        bool** matrix = new bool* [xLength];

        // Create the arrays inside each outer array
        for (int i = 0; i < xLength; ++i) {
                matrix[i] = new bool[yLength]();
        }
        return matrix;
}
```

```
void FreeMatrix(REAL** matrix, int xLength) {
        for (int i = 0; i < xLength; ++i) {
                delete[] matrix[i];
        }
        delete[] matrix;
}

void FreeMatrix(BYTE** matrix, int xLength) {
        for (int i = 0; i < xLength; ++i) {
                delete[] matrix[i];
        }
        delete[] matrix;
}

void FreeMatrix(bool** matrix, int xLength) {
        for (int i = 0; i < xLength; ++i) {
                delete[] matrix[i];
        }
        delete[] matrix;
}
```

Init.h

```
#ifndef INIT_H
#define INIT_H

#include "pch.h"

REAL** MatrixMAlloc(int xLength, int yLength);

BYTE** FlagMatrixMAlloc(int xLength, int yLength);

bool** ObstacleMatrixMAlloc(int xLength, int yLength);

void FreeMatrix(REAL** matrix, int xLength);

void FreeMatrix(BYTE** matrix, int xLength);

void FreeMatrix(bool** matrix, int xLength);

#endif
```

pch.cpp

```
// pch.cpp: source file corresponding to the pre-compiled header

#include "pch.h"

// When you are using pre-compiled headers, this source file is necessary for
↪   compilation to succeed.
```

pch.h

```
// pch.h: This is a precompiled header file.
// Files listed below are compiled only once, improving build performance for future
↪  builds.

#ifndef PCH_H
#define PCH_H

#include <utility>
#include <memory>
#include "Definitions.h"

#endif // PCH_H
```

PipeConstants.h

```
#ifndef PIPE_CONSTANTS_H
#define PIPE_CONSTANTS_H
#include "pch.h"

namespace PipeConstants {
    constexpr BYTE CATEGORYMASK = 0b11000000;

    namespace Status
    {
        constexpr BYTE GENERIC = 0b00000000;
        constexpr BYTE HELLO = 0b00001000;
        constexpr BYTE BUSY = 0b00010000;
        constexpr BYTE OK = 0b00011000;
        constexpr BYTE STOP = 0b00100000;
        constexpr BYTE CLOSE = 0b00101000;

        constexpr BYTE PARAMMASK = 0b00000111;
    }
    namespace Request
    {
        constexpr BYTE GENERIC = 0b01000000;
        constexpr BYTE FIXLENREQ = 0b01000000;
        constexpr BYTE CONTREQ = 0b01100000;

        constexpr BYTE PARAMMASK = 0b00011111;

        constexpr BYTE HVEL = 0b00010000;
        constexpr BYTE VVEL = 0b00001000;
        constexpr BYTE PRES = 0b00000100;
        constexpr BYTE STRM = 0b00000010;
    }
    namespace Marker
    {
        constexpr BYTE GENERIC = 0b10000000;
        constexpr BYTE ITERSTART = 0b10000000;
        constexpr BYTE ITEREND = 0b10001000;
        constexpr BYTE FLDSTART = 0b10010000;
        constexpr BYTE FLDEND = 0b10011000;

        constexpr BYTE ITERPRMMASK = 0b00000111;

        constexpr BYTE HVEL = 0b00000001;
        constexpr BYTE VVEL = 0b00000010;
        constexpr BYTE PRES = 0b00000011;
        constexpr BYTE STRM = 0b00000100;
        constexpr BYTE OBST = 0b00000101;

        constexpr BYTE PRMSTART = 0b10100000;
        constexpr BYTE PRMEND = 0b10101000;

        constexpr BYTE PRMMASK = 0b00001111;

        constexpr BYTE IMAX = 0b00000001;
        constexpr BYTE JMAX = 0b00000010;
        constexpr BYTE WIDTH = 0b00000011;
        constexpr BYTE HEIGHT = 0b00000100;
        constexpr BYTE TAU = 0b00000101;
        constexpr BYTE OMEGA = 0b00000110;
        constexpr BYTE RMAX = 0b00000111;
        constexpr BYTE ITERMAX = 0b00001000;
        constexpr BYTE REYNOLDS = 0b00001001;
        constexpr BYTE INVEL = 0b00001010;
        constexpr BYTE CHI = 0b00001011;
        constexpr BYTE MU = 0b00001100;
        constexpr BYTE DENSITY = 0b0001101;
        constexpr BYTE DRAGCOEF = 0b00001110;
    }
    namespace Error
    {
        constexpr BYTE GENERIC = 0b11000000;
        constexpr BYTE BADREQ = 0b11000001;
        constexpr BYTE BADPARAM = 0b11000010;
        constexpr BYTE INTERNAL = 0b11000011;
        constexpr BYTE TIMEOUT = 0b11000100;
        constexpr BYTE BADTYPE = 0b11000101;
        constexpr BYTE BADLEN = 0b11000110;
    }
}


#endif // !PIPE_CONSTANTS_H
```

PipeManager.cpp

```
#include "pch.h"
#include "PipeManager.h"
#include <iostream>
#include "PipeConstants.h"
#include <algorithm>

#pragma region Private Methods
```

```
std::wstring PipeManager::WidenString(std::string input) {
    return std::wstring(input.begin(), input.end());
}

void PipeManager::ReadToNull(BYTE* outBuffer) {
    DWORD read = 0; // Number of bytes read in each ReadFile() call
    int index = 0;
    do {
        if (!ReadFile(pipeHandle, outBuffer + index, 1, &read, NULL)) {
            std::cerr << "Failed to read from the named pipe, error code " <<
             ↪  GetLastError() << std::endl;
            break;
        }
        index++;
    } while (outBuffer[index - 1] != 0); // Stop if the most recent byte was
     ↪  null-termination
}

bool PipeManager::Read(BYTE* outBuffer, int bytesToRead) {
    DWORD bytesRead;
    return ReadFile(pipeHandle, outBuffer, bytesToRead, &bytesRead, NULL) &&
     ↪  bytesRead == bytesToRead; // Success if bytes were read, and enough bytes
     ↪  were read
}

BYTE PipeManager::Read() {
    BYTE outputByte;
    if (!ReadFile(pipeHandle, &outputByte, 1, nullptr, NULL)) {
        std::cerr << "Failed to read from the named pipe, error code " <<
         ↪  GetLastError() << std::endl;
    }
    return outputByte;
}

void PipeManager::Write(const BYTE* buffer, DWORD bufferLength)
{
    if (!WriteFile(pipeHandle, buffer, bufferLength, nullptr, NULL)) {
        std::cerr << "Failed to write to the named pipe, error code " <<
         ↪  GetLastError() << std::endl;
    }
}

void PipeManager::Write(BYTE byte) {
    if (!WriteFile(pipeHandle, &byte, 1, nullptr, NULL)) {
        std::cerr << "Failed to write to the named pipe, error code " <<
         ↪  GetLastError() << std::endl;
    }
}

void PipeManager::SerialiseField(BYTE* buffer, REAL** field, int xLength, int
 ↪  yLength, int xOffset, int yOffset) {
    /*
    * The thinking is thus:
```

```
    * Each "row" of the field will be stored contiguously
    * The relevant part of these rows will span from (yOffset) to (yOffset +
    ↪  yLength)
    * Therefore each row can be copied directly into the buffer
    * The location in the buffer will have to increment by yLength * sizeof(REAL)
    ↪  each time.
    */
    for (int i = 0; i < xLength; i++) { // Copy one row at a time (rows are not
    ↪  guaranteed to be contiguously stored)
        std::memcpy(
            buffer + i * yLength * sizeof(REAL), // Start index of destination,
            ↪  buffer + i * column length * 4
            field[i + xOffset] + yOffset, // Start index of source, start index of
            ↪  the column + y offset
            yLength * sizeof(REAL) // Number of bytes to copy, column size * 4
        );
    }
}
#pragma endregion

#pragma region Constructors/Destructors
// Constructor for a named pipe, yet to be connected to
PipeManager::PipeManager(std::string pipeName) {
    pipeHandle = CreateFile(WidenString("\\\\.\\pipe\\" + pipeName).c_str(),
    ↪  GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING, 0, NULL);
    std::cout << "File opened\n";
}

// Constructor for if the named pipe has already been connected to
PipeManager::PipeManager(HANDLE existingHandle) : pipeHandle(existingHandle) {} //
↪  Pass the handle into the local handle

PipeManager::~PipeManager() {
    CloseHandle(pipeHandle);
}
#pragma endregion

#pragma region Public Methods
bool PipeManager::Handshake(int iMax, int jMax) {
    BYTE receivedByte = Read();
    if (receivedByte != PipeConstants::Status::HELLO) { // We need a HELLO byte
        std::cerr << "Handshake not completed - server sent malformed request";
        Write(PipeConstants::Error::BADREQ);
        return false;
    }


    BYTE buffer[13];
    buffer[0] = PipeConstants::Status::HELLO; // Reply with HELLO byte

    buffer[1] = PipeConstants::Marker::PRMSTART | PipeConstants::Marker::IMAX; //
    ↪  Send iMax, demarked with PRMSTART and PRMEND
    for (int i = 0; i < 4; i++) {
        buffer[i + 2] = iMax >> (i * 8);
    }
    buffer[6] = PipeConstants::Marker::PRMEND | PipeConstants::Marker::IMAX;

    buffer[7] = PipeConstants::Marker::PRMSTART | PipeConstants::Marker::JMAX; //
    ↪  Send jMax, demarked with PRMSTART and PRMEND
    for (int i = 0; i < 4; i++) {
        buffer[i + 8] = jMax >> (i * 8);
    }
    buffer[12] = PipeConstants::Marker::PRMEND | PipeConstants::Marker::JMAX;

    Write(buffer, 13);

    return Read() == PipeConstants::Status::OK; // Success if an OK byte is
    ↪  received
}

std::pair<int, int> PipeManager::Handshake() {
    BYTE receivedByte = Read();

    if (receivedByte != PipeConstants::Status::HELLO) { return std::pair<int,
    ↪  int>(0, 0); } // We need a HELLO byte, (0,0) is the error case

    Write(PipeConstants::Status::HELLO);

    BYTE buffer[12];
    Read(buffer, 12);

    if (buffer[0] != (PipeConstants::Marker::PRMSTART |
    ↪  PipeConstants::Marker::IMAX)) { return std::pair<int, int>(0, 0); } //
    ↪  Should start with PRMSTART
    int iMax = *reinterpret_cast<int*>(buffer + 1);
    if (buffer[5] != (PipeConstants::Marker::PRMEND | PipeConstants::Marker::IMAX))
    ↪  { return std::pair<int, int>(0, 0); } // Should end with PRMEND

    if (buffer[6] != (PipeConstants::Marker::PRMSTART |
    ↪  PipeConstants::Marker::JMAX)) { return std::pair<int, int>(0, 0); }
    int jMax = *reinterpret_cast<int*>(buffer + 7);
    if (buffer[11] != (PipeConstants::Marker::PRMEND | PipeConstants::Marker::JMAX))
    ↪  { return std::pair<int, int>(0, 0); }

    Write(PipeConstants::Status::OK); // Send an OK byte to show the transmission
    ↪  was successful

    return std::pair<int, int>(iMax, jMax);
}

bool PipeManager::ReceiveObstacles(bool* obstacles, int xLength, int yLength) {
    int fieldLength = xLength * yLength;
    int bufferLength = fieldLength / 8 + (fieldLength % 8 == 0 ? 0 : 1);

    // Assume there has been a FLDSTART before
    BYTE* buffer = new BYTE[bufferLength + 1]; // Have to use new keyword because
    ↪  length of array is not a constant expression
    Read(buffer, bufferLength + 1);

    int byteNumber = 0;
    for (int i = 0; i < fieldLength; i++) {
        obstacles[byteNumber * 8 + (i % 8)] = (((buffer[byteNumber] >> (i % 8)) & 1)
        ↪   == 0) ? false : true; // Due to the way bits are shifted into the bytes
        ↪   by the server, they must be shifted off in the opposite order hence the
        ↪   complicated expression for obstacles[...]. Right shift and AND with 1
        ↪   takes that bit only

        if (i % 8 == 7) {
            byteNumber++;
        }
    }

    if (buffer[bufferLength] != (PipeConstants::Marker::FLDEND |
    ↪   PipeConstants::Marker::OBST)) { // Ensure there is a FLDEND after
        std::cerr << "Cannot read obstacles - server sent malformed data. ";
        Write(PipeConstants::Error::BADPARAM);
        return false;
    }
    delete[] buffer;

    Write(PipeConstants::Status::OK); // Send an OK message to server to tell it the
    ↪   data was understood
    return true;
}

BYTE PipeManager::ReadByte() {
    return Read();
}

void PipeManager::SendByte(BYTE byte) {
    Write(byte);
}

REAL PipeManager::ReadReal() {
    BYTE buffer[sizeof(REAL)];
    Read(buffer, sizeof(REAL));
    REAL* pOutput = reinterpret_cast<REAL*>(buffer);
    return *pOutput;
}

void PipeManager::SendReal(REAL datum) {
    BYTE* buffer = reinterpret_cast<BYTE*>(&datum);
    Write(buffer, sizeof(REAL));
}

int PipeManager::ReadInt() {
    BYTE buffer[sizeof(int)];
    Read(buffer, sizeof(int));
    int* pOutput = reinterpret_cast<int*>(buffer);
    return *pOutput;
}

void PipeManager::SendField(REAL** field, int xLength, int yLength, int xOffset, int
↪   yOffset)
{
    BYTE* buffer = new BYTE[xLength * yLength * sizeof(REAL)];

    SerialiseField(buffer, field, xLength, yLength, xOffset, yOffset);

    Write(buffer, xLength * yLength * sizeof(REAL));

    delete[] buffer;
}

void PipeManager::SendField(REAL* field, int numElements) {
    Write(reinterpret_cast<BYTE*>(field), numElements * sizeof(REAL));
}
#pragma endregion
```

PipeManager.h

```
#ifndef PIPE_MANAGER_H
#define PIPE_MANAGER_H

#include "pch.h"
#include <windows.h>
#include <string>

class PipeManager
{
private:
        HANDLE pipeHandle;
        std::wstring WidenString(std::string input);
        void ReadToNull(BYTE* outBuffer);
        bool Read(BYTE* outBuffer, int bytesToRead);
        BYTE Read();
        void Write(const BYTE* buffer, DWORD bufferLength);
        void Write(BYTE byte);

        /// <summary>
        /// A method to convert a 2D array of REALs (field) into a flat array of
        ↪   BYTEs for transmission over the pipe.
        /// </summary>
        /// <param name="buffer">An array of BYTEs, with length <c>sizeof(REAL) *
        ↪   fieldSize</c>.</param>
        /// <param name="field">The 2D array of REALs to serialise.</param>
        /// <param name="xLength">The number of REALs to serialise in the x
        ↪   direction.</param>
        /// <param name="yLength">The number of REALs to serialise in the y
        ↪   direction.</param>
        /// <param name="xOffset">The x-index of the first REAL to be
        ↪   serialised.</param>
        /// <param name="yOffset">The y-index of the first REAL to be
        ↪  serialised.</param>
        void SerialiseField(BYTE* buffer, REAL** field, int xLength, int yLength,
        ↪  int xOffset, int yOffset);

public:
        /// <summary>
        /// Constructor to connect to the named pipe
        /// </summary>
        /// <param name="pipeName">The name of the named pipe for communication with
        ↪  the frontend</param>
        PipeManager(std::string pipeName);

        /// <summary>
        /// Constructor accepting an already connected pipe's handle
        /// </summary>
        /// <param name="pipeHandle">The handle of a connected pipe</param>
        PipeManager(HANDLE pipeHandle);

        /// <summary>
        /// Pipe manager destructor - disconnects from the named pipe then closes
        /// </summary>
        ~PipeManager();

        /// <summary>
        /// Performs a handshake with the frontend.
        /// </summary>
        /// <returns>A <c>bool</c> indicating whether the handshake completed
        ↪  successfully.</returns>
        bool Handshake(int iMax, int jMax);

        /// <summary>
        /// Performs a handshake with the frontend.
        /// </summary>
        /// <returns>A std::pair, with the values of iMax and jMax (the simulation
        ↪  domain's dimensions).</returns>
        std::pair<int, int> Handshake();

        /// <summary>
        /// A subroutine to receive obstacles through the pipe, and convert them to
        ↪  a bool array.
        /// </summary>
        /// <param name="obstacles">The obstacles array to output to.</param>
        /// <param name="xLength">The number of cells in the x direction</param>
        /// <param name="yLength">The number of cells in the y direction</param>
        /// <returns>A <c>bool</c> indicating whether the action was
        ↪  successful.</returns>
        bool ReceiveObstacles(bool* obstacles, int xLength, int yLength);

        /// <summary>
        /// Reads a byte from the pipe, and returns it
        /// </summary>
        /// <returns>The single byte read from the pipe</returns>
        BYTE ReadByte();

        /// <summary>
        /// Writes a single byte to the pipe
        /// </summary>
        /// <param name="byte">The byte to write</param>
        void SendByte(BYTE byte);

        /// <summary>
        /// Reads a <c>REAL</c> data type from the pipe, assuming one has been
        ↪  sent.
        /// </summary>
        /// <returns>The converted <c>REAL</c> read from the pipe.</returns>
        REAL ReadReal();

        /// <summary>
        /// Sends one REAL datum down the pipe.
        /// </summary>
        /// <param name="data">The REAL to send down the pipe.</param>
        void SendReal(REAL data);

        /// <summary>
        /// Reads a <c>int</c> data type from the pipe, assuming one has been sent.
        /// </summary>
        /// <returns>The converted <c>int</c> read from the pipe.</returns>
        int ReadInt();

        /// <summary>
        /// Sends the contents of a field through the pipe.
        /// </summary>
        /// <param name="field">An array of pointers to the rows of the
        ↪  field.</param>
        /// <param name="xLength">The length in the x direction that will be
        ↪  transmitted.</param>
        /// <param name="yLength">The length in the y direction that will be
        ↪  transmitted.</param>
        /// <param name="xOffset">The x-index of the first value to be
        ↪  transmitted.</param>
        /// <param name="yOffset">The y-index of the first value to be
        ↪  transmitted.</param>
        void SendField(REAL** field, int xLength, int yLength, int xOffset, int
        ↪  yOffset);

        /// <summary>
        /// Sends the contents of a field through the pipe.
        /// </summary>
        /// <param name="field">The field to transmit as a flattened array.</param>
        /// <param name="numElements">The number of elements in the field, <c>height
        ↪  * width</c>.</param>
        void SendField(REAL* field, int numElements);
};

#endif // !PIPE_MANAGER_H
```

Solver.cpp

```
#include "pch.h"
#include "Solver.h"

Solver::Solver(SimulationParameters parameters, int iMax, int jMax) : iMax(iMax),
↪  jMax(jMax), parameters(parameters) {}

Solver::~Solver() {

}

template<typename T>
void Solver::UnflattenArray(T** pointerArray, int paDownOffset, int paLeftOffset, T*
↪  flattenedArray, int faDownOffset, int faUpOffset, int faLeftOffset, int xLength,
↪  int yLength) {
        int faTotalYLength = faDownOffset + yLength + faUpOffset;
        for (int i = 0; i < xLength; i++) { // Copy one row at a time
                memcpy(
                        pointerArray[i + paLeftOffset] + paDownOffset,
                        ↪  // Destination address - ptr to row (i + paLeftoffset)
                        ↪  and column starts at paDownOffset
                        flattenedArray + (i + faLeftOffset) * faTotalYLength +
                        ↪  faDownOffset, // Source start address - (i +
                        ↪  faLeftOffset) * size of a column including offsets, and
                        ↪  add down offset for start of copy address
                        yLength * sizeof(T)
                        ↪  // Number of bytes to copy, size of a column, excluding
                        ↪  offsets
                );
        }
}
template void Solver::UnflattenArray(bool** pointerArray, int paDownOffset, int
↪  paLeftOffset, bool* flattenedArray, int faDownOffset, int faUpOffset, int
↪  faLeftOffset, int xLength, int yLength); // Templates for the types I may plan
↪  to use
template void Solver::UnflattenArray(BYTE** pointerArray, int paDownOffset, int
↪  paLeftOffset, BYTE* flattenedArray, int faDownOffset, int faUpOffset, int
↪  faLeftOffset, int xLength, int yLength);
template void Solver::UnflattenArray(REAL** pointerArray, int paDownOffset, int
↪  paLeftOffset, REAL* flattenedArray, int faDownOffset, int faUpOffset, int
↪  faLeftOffset, int xLength, int yLength);


template<typename T>
void Solver::FlattenArray(T** pointerArray, int paDownOffset, int paLeftOffset, T*
↪  flattenedArray, int faDownOffset, int faUpOffset, int faLeftOffset, int xLength,
↪  int yLength) {
        int faTotalYLength = faDownOffset + yLength + faUpOffset;
        for (int i = 0; i < xLength; i++) { // Copy one row at a time (rows are not
        ↪  guaranteed to be contiguously stored)
                memcpy(
                        flattenedArray + (i + faLeftOffset) * faTotalYLength +
                        ↪   faDownOffset,  // Destination address - (i +
                        ↪   faLeftOffset) * size of a column including offsets, and
                        ↪   add down offset for start of copy address
                        pointerArray[i + paLeftOffset] + paDownOffset,
                        ↪   // Source start address - ptr to row (i + paLeftoffset)
                        ↪   and column starts at paDownOffset
                        yLength * sizeof(T)
                        ↪   // Number of bytes to copy, size of a column, excluding
                        ↪   offsets.
                );
        }
}
template void Solver::FlattenArray(bool** pointerArray, int paDownOffset, int
↪   paLeftOffset, bool* flattenedArray, int faDownOffset, int faUpOffset, int
↪   faLeftOffset, int xLength, int yLength);
template void Solver::FlattenArray(BYTE** pointerArray, int paDownOffset, int
↪   paLeftOffset, BYTE* flattenedArray, int faDownOffset, int faUpOffset, int
↪   faLeftOffset, int xLength, int yLength);
template void Solver::FlattenArray(REAL** pointerArray, int paDownOffset, int
↪   paLeftOffset, REAL* flattenedArray, int faDownOffset, int faUpOffset, int
↪   faLeftOffset, int xLength, int yLength);

SimulationParameters Solver::GetParameters() const {
    return parameters;
}

void Solver::SetParameters(SimulationParameters parameters) {
    this->parameters = parameters;
}

int Solver::GetIMax() const {
    return iMax;
}

int Solver::GetJMax() const {
    return jMax;
}
```

Solver.h

```
#ifndef SOLVER_H
#define SOLVER_H

#include "pch.h"

class Solver
{
protected:
    int iMax;
    int jMax;

    SimulationParameters parameters;
    /// <summary>
    /// Unflattens the array specified in <paramref name="flattenedArray" />,
    ↪  storing the result in <paramref name="pointerArray" />.
    /// </summary>
    /// <typeparam name="T">The type of the elements in the array.</typeparam>
    /// <param name="pointerArray">The output 2D array.</param>
    /// <param name="paDownOffset">The number of elements below the region of a
    ↪  column to be copied into the pointer array.</param>
    /// <param name="paLeftOffset">The number of elements left of the region of a
    ↪  row to be copied into the pointer array.</param>
    /// <param name="flattenedArray">The input flattened array</param>
    /// <param name="faDownOffset">The number of elements below the region of a
    ↪  column to be copied from the flattened array.</param>
    /// <param name="faUpOffset">The number of elements above the region of a column
    ↪  to be copied from the flattened array.</param>
    /// <param name="faLeftOffset">The number of elements left of the region of a
    ↪  row to be copied from the flattened array.</param>
    /// <param name="xLength">The number of elements in the x direction that are to
    ↪  be copied.</param>
    /// <param name="yLength">The number of elements in the y direction that are to
    ↪  be copied.</param>
    template<typename T>
    void UnflattenArray(T** pointerArray, int paDownOffset, int paLeftOffset, T*
    ↪  flattenedArray, int faDownOffset, int faUpOffset, int faLeftOffset, int
    ↪  xLength, int yLength);

    /// <summary>
    /// Flattens the 2D array specified in <paramref name="pointerArray" />, storing
    ↪  the result in <paramref name="flattenedArray" />.
    /// </summary>
    /// <typeparam name="T">The type of the elements in the array.</typeparam>
    /// <param name="pointerArray">The input 2D array.</param>
    /// <param name="paDownOffset">The number of elements below the region of a
    ↪  column to be copied from the pointer array.</param>
    /// <param name="paLeftOffset">The number of elements left of the region of a
    ↪  row to be copied from the pointer array.</param>
    /// <param name="flattenedArray">The output flattened array</param>
    /// <param name="faDownOffset">The number of elements below the region of a
    ↪  column to be copied into the flattened array.</param>
    /// <param name="faUpOffset">The number of elements above the region of a column
    ↪  to be copied into the flattened array.</param>
    /// <param name="faLeftOffset">The number of elements left of the region of a
    ↪  row to be copied into the flattened array.</param>
    /// <param name="xLength">The number of elements in the x direction that are to
    ↪  be copied.</param>
    /// <param name="yLength">The number of elements in the y direction that are to
    ↪  be copied.</param>
    template<typename T>
    void FlattenArray(T** pointerArray, int paDownOffset, int paLeftOffset, T*
    ↪  flattenedArray, int faDownOffset, int faUpOffset, int faLeftOffset, int
    ↪  xLength, int yLength);

public:
    /// <summary>
    /// Initialises the class's fields and parameters
    /// </summary>
    /// <param name="parameters">The parameters to use for simulation. This may be
    ↪   changed before calls to <see cref="Timestep" />.</param>
    /// <param name="iMax">The index of the rightmost fluid cell</param>
    /// <param name="jMax">The index of the topmost fluid cell</param>
    Solver(SimulationParameters parameters, int iMax, int jMax);

    ~Solver();

    SimulationParameters GetParameters() const;
    void SetParameters(SimulationParameters parameters);

    int GetIMax() const;
    int GetJMax() const;

    virtual REAL* GetHorizontalVelocity() const = 0;

    virtual REAL* GetVerticalVelocity() const = 0;

    virtual REAL* GetPressure() const = 0;

    virtual REAL* GetStreamFunction() const = 0;

    virtual bool** GetObstacles() const = 0;

    virtual REAL GetDragCoefficient() = 0;

    /// <summary>
    /// Embeds obstacles into the simulation domain. Assumes obstacles have already
    ↪   been set
    /// </summary>
    virtual void ProcessObstacles() = 0;

    /// <summary>
    /// Performs setup for executing timesteps. This function must be called once
    ↪   before the first call to <c>Timestep</c>, and after any changes to
    ↪   <c>parameters</c>.
    /// </summary>
    virtual void PerformSetup() = 0;

    /// <summary>
    /// Computes one timestep, solving each of the fields.
    /// </summary>
    /// <param name="simulationTime">The time that the simulation has been running,
    ↪   to be updated with the new time after the timestep has finished.</param>
    virtual void Timestep(REAL& simulationTime) = 0;
};
#endif // !SOLVER_H
```

Boundary.cpp

```
#include "Boundary.h"
#include <bitset>
#include <vector>
#include <iostream>

#define XVEL velocities.x[coordinates[coord].first][coordinates[coord].second]
#define YVEL velocities.y[coordinates[coord].first][coordinates[coord].second]

constexpr BYTE TOPMASK =    0b00001000;
constexpr BYTE RIGHTMASK =  0b00000100;
constexpr BYTE BOTTOMMASK = 0b00000010;
constexpr BYTE LEFTMASK =   0b00000001;
constexpr int TOPSHIFT = 3;
constexpr int RIGHTSHIFT = 2;
constexpr int BOTTOMSHIFT = 1;

void SetBoundaryConditions(DoubleField velocities, BYTE** flags, std::pair<int,
↪   int>* coordinates, int coordinatesLength, int iMax, int jMax, REAL
↪   inflowVelocity, REAL chi) {
    REAL velocityModifier = 2 * chi - 1; // This converts chi from chi in [0,1] to
    ↪   in [-1,1]

    // Top and bottom: free-slip
    for (int i = 1; i <= iMax; i++) {
        velocities.y[i][0] = 0; // No mass crossing the boundary - velocity is 0
        velocities.y[i][jMax] = 0;

        velocities.x[i][0] = velocities.x[i][1]; // Speed outside the boundary is
        ↪   the same as the speed inside
        velocities.x[i][jMax + 1] = velocities.x[i][jMax];
    }

    for (int j = 1; j <= jMax; j++) {
        // Left: inflow
        velocities.x[0][j] = inflowVelocity; // Fluid flows in the x direction at a
        ↪   set velocity...
        velocities.y[0][j] = 0; // ...and there should be no movement in the y
        ↪   direction

        // Right: outflow
        velocities.x[iMax][j] = velocities.x[iMax - 1][j]; // Copy the velocity
        ↪   values from the previous cell (mass flows out at the boundary)
        velocities.y[iMax + 1][j] = velocities.y[iMax][j];
    }

    // Obstacle boundary cells: partial-slip
    for (int coord = 0; coord < coordinatesLength; coord++) {
        BYTE relevantFlag =
        ↪   flags[coordinates[coord].first][coordinates[coord].second];
        switch (relevantFlag) {
        case B_N:
```

```
        XVEL = velocityModifier *
         ↪  velocities.x[coordinates[coord].first][coordinates[coord].second +
         ↪  1]; // Tangential velocity: friction
        YVEL = 0; // Normal velocity = 0
        break;
    case B_NE:
        XVEL = 0; // Both velocities owned by a B_NE are normal, so set to 0.
        YVEL = 0;
        break;
    case B_E:
        XVEL = 0; // Normal velocity = 0
        YVEL = velocityModifier * velocities.y[coordinates[coord].first +
         ↪  1][coordinates[coord].second]; // Tangential velocity: friction
        break;
    case B_SE:
        XVEL = 0;
        YVEL = velocityModifier * velocities.y[coordinates[coord].first +
         ↪  1][coordinates[coord].second]; // Tangential velocity: friction
        velocities.y[coordinates[coord].first][coordinates[coord].second - 1] =
         ↪  0; // y velocity south of a B_SE must be set to 0
        break;
    case B_S:
        XVEL = velocityModifier *
         ↪  velocities.x[coordinates[coord].first][coordinates[coord].second -
         ↪  1]; // Tangential velocity: friction
        velocities.y[coordinates[coord].first][coordinates[coord].second - 1] =
         ↪  0; // y velocity south of a B_S must be set to 0
        break;
    case B_SW:
        XVEL = velocityModifier *
         ↪  velocities.x[coordinates[coord].first][coordinates[coord].second -
         ↪  1]; // Tangential velocity: friction
        YVEL = velocityModifier * velocities.y[coordinates[coord].first -
         ↪  1][coordinates[coord].second]; // Tangential velocity: friction
        velocities.x[coordinates[coord].first - 1][coordinates[coord].second] =
         ↪  0; // x velocity west of a B_SW must be set to 0
        velocities.y[coordinates[coord].first][coordinates[coord].second - 1] =
         ↪  0; // y velocity south of a B_SW must be set to 0
        break;
    case B_W:
        YVEL = velocityModifier * velocities.y[coordinates[coord].first -
         ↪  1][coordinates[coord].second]; // Tangential velocity: friction
        velocities.x[coordinates[coord].first - 1][coordinates[coord].second] =
         ↪  0; // x velocity west of a B_W must be set to 0
        break;
    case B_NW:
        XVEL = velocityModifier *
         ↪  velocities.x[coordinates[coord].first][coordinates[coord].second +
         ↪  1]; // Tangential velocity: friction
        YVEL = 0; // Normal velocity = 0
        velocities.x[coordinates[coord].first - 1][coordinates[coord].second] =
         ↪  0; // x velocity west of a B_NW must be set to 0
        break;
        }
        // Any velocities for a cell with a north or east bit unset (referring to an
        ↪  obstacle in that direction) must be set to 0, i.e. cells south or west
        ↪  of a boundary.
    }
}

void CopyBoundaryPressures(REAL** pressure, std::pair<int,int>* coordinates, int
↪  numCoords, BYTE** flags, int iMax, int jMax) {
    for (int i = 1; i <= iMax; i++) {
        pressure[i][0] = pressure[i][1];
        pressure[i][jMax + 1] = pressure[i][jMax];
    }
    for (int j = 1; j <= jMax; j++) {
        pressure[0][j] = pressure[1][j];
        pressure[iMax + 1][j] = pressure[iMax][j];
    }
    for (int coord = 0; coord < numCoords; coord++) {
        BYTE relevantFlag =
        ↪  flags[coordinates[coord].first][coordinates[coord].second];
        int numEdges = (int)std::bitset<8>(relevantFlag).count();
        if (numEdges == 1) {
            pressure[coordinates[coord].first][coordinates[coord].second] =
            ↪  pressure[coordinates[coord].first + ((relevantFlag & RIGHTMASK) >>
            ↪  RIGHTSHIFT) - (relevantFlag & LEFTMASK)][coordinates[coord].second +
            ↪  ((relevantFlag & TOPMASK) >> TOPSHIFT) - ((relevantFlag &
            ↪  BOTTOMMASK) >> BOTTOMSHIFT)]; // Copying pressure from the relevant
            ↪  cell. Using anding with bit masks to do things like [i+1][j] using
            ↪  single bits
        }
        else { // These are boundary cells with 2 edges
            pressure[coordinates[coord].first][coordinates[coord].second] =
            ↪  (pressure[coordinates[coord].first + ((relevantFlag & RIGHTMASK) >>
            ↪  RIGHTSHIFT) - (relevantFlag & LEFTMASK)][coordinates[coord].second]
            ↪  + pressure[coordinates[coord].first][coordinates[coord].second +
            ↪  ((relevantFlag & TOPMASK) >> TOPSHIFT) - ((relevantFlag &
            ↪  BOTTOMMASK) >> BOTTOMSHIFT)]) / (REAL)2; // Take the average of the
            ↪  one above/below and the one left/right by keeping j constant for the
            ↪  first one, and I constant for the second one.
        }
    }
}

std::pair<std::pair<int, int>*, int> FindBoundaryCells(BYTE** flags, int iMax, int
↪  jMax) { // Returns size of array and actual array
    std::vector<std::pair<int, int>> coordinates;
    for (int i = 1; i <= iMax; i++) {
        for (int j = 1; j <= jMax; j++) {
            if (flags[i][j] >= 0b00000001 && flags[i][j] <= 0b00001111) { // This
            ↪  defines boundary cells - all cells without the self bit set except
            ↪  when no bits are set. This could probably be optimised.
                coordinates.push_back(std::pair<int, int>(i, j));
            }
        }
    }
    std::pair<int, int>* coordinatesAsArray = new std::pair<int,
    ↪  int>[coordinates.size()]; // Allocate mem for array into already defined
    ↪  pointer
    std::copy(coordinates.begin(), coordinates.end(), coordinatesAsArray); // Copy
    ↪  the elements from the vector to the array
    return std::pair<std::pair<int, int>*, int>(coordinatesAsArray,
    ↪  (int)coordinates.size()); // Return the array with values copied into it and
    ↪  the size
}
```

Boundary.h

```
#ifndef BOUNDARY_H
#define BOUNDARY_H

#include "pch.h"

void SetBoundaryConditions(DoubleField velocities, BYTE** flags, std::pair<int,
↪  int>* coordinates, int coordinatesLength, int iMax, int jMax, REAL
↪  inflowVelocity, REAL chi);

void CopyBoundaryPressures(REAL** pressure, std::pair<int, int>* coordinates, int
↪  numCoords, BYTE** flags, int iMax, int jMax);

std::pair<std::pair<int, int>*, int> FindBoundaryCells(BYTE** flags, int iMax, int
↪  jMax);

#endif
```

Computation.cpp

```
#include "Computation.h"
#include "DiscreteDerivatives.h"
#include "Init.h"
#include "Boundary.h"
#include <iostream>
#include <thread>
//#define DEBUGOUT

REAL ArraySum(REAL* array, int arrayLength) {
    if (arrayLength == 0) return 0;
    if (arrayLength == 1) return array[0];
    int midPoint = arrayLength / 2;
    return ArraySum(array, midPoint) + ArraySum((array + midPoint), arrayLength -
    ↪  midPoint);
}

REAL FieldMax(REAL** field, int xLength, int yLength) {
    REAL max = 0;
    for (int i = 0; i < xLength; ++i) {
        for (int j = 0; j < yLength; ++j) {
            if (field[i][j] > max) {
                max = field[i][j];
            }
        }
    }
    return max;
}

REAL ComputeGamma(DoubleField velocities, int iMax, int jMax, REAL timestep,
↪   DoubleReal stepSizes) {
    REAL horizontalComponent = FieldMax(velocities.x, iMax+2, jMax+2) * (timestep /
    ↪   stepSizes.x);
    REAL verticalComponent = FieldMax(velocities.y, iMax+2, jMax+2) * (timestep /
    ↪   stepSizes.y);

    if (horizontalComponent > verticalComponent) {
        return horizontalComponent;
    }
    return verticalComponent;
}

void ComputeFG(DoubleField velocities, DoubleField FG, BYTE** flags, int iMax, int
↪   jMax, REAL timestep, DoubleReal stepSizes, DoubleReal bodyForces, REAL gamma,
↪   REAL reynoldsNo) {
    // F or G must be set to the corresponding velocity when this references a
    ↪   velocity crossing a boundary
    // F must be set to u when the self bit and the east bit are different (eastern
    ↪   boundary cells and fluid cells to the west of a boundary)
    // G must be set to v when the self bit and the north bit are different
    ↪   (northern boundary cells and fluid cells to the south of a boundary)
    for (int i = 0; i <= iMax; ++i) {
        for (int j = 0; j <= jMax; ++j) {
            if (i == 0 && j == 0) { // Values equal to 0 are boundary cells and are
            ↪   separate with flag 0.
                continue;
            }
            if (i == 0) { // Setting F equal to u and G equal to v at the
            ↪   boundaries
                FG.x[i][j] = velocities.x[i][j];
                continue;
            }
            if (j == 0) {
                FG.y[i][j] = velocities.y[i][j];
                continue;
            }

            if (i == iMax) { // Flag of these will be 00010xxx
                FG.x[i][j] = velocities.x[i][j];
            }
            if (j == jMax) { // Flag of these will be 0001x0xx
                FG.y[i][j] = velocities.y[i][j];
            }

            if (flags[i][j] & SELF && flags[i][j] & EAST) { // If self bit and east
            ↪   bit are both 1 - fluid cell not near a boundary
```

```
                FG.x[i][j] = velocities.x[i][j] + timestep * (1 / reynoldsNo *
                 ↪ (SecondPuPx(velocities.x, i, j, stepSizes.x) +
                 ↪ SecondPuPy(velocities.x, i, j, stepSizes.y)) -
                 ↪ PuSquaredPx(velocities.x, i, j, stepSizes.x, gamma) -
                 ↪ PuvPy(velocities, i, j, stepSizes, gamma) + bodyForces.x);
            }
            else if (!(flags[i][j] & SELF) && !(flags[i][j] & EAST)) { // If self
             ↪ bit and east bit are both 0 - inside an obstacle
                FG.x[i][j] = 0;
            }
            else { // The variable's position lies on a boundary (though the cell
             ↪ may not - a side-effect of the staggered-grid discretisation.
                FG.x[i][j] = velocities.x[i][j];
            }

            if (flags[i][j] & SELF && flags[i][j] & NORTH) { // Same as for G, but
             ↪ the relevant bits are self and north
                FG.y[i][j] = velocities.y[i][j] + timestep * (1 / reynoldsNo *
                 ↪ (SecondPvPx(velocities.y, i, j, stepSizes.x) +
                 ↪ SecondPvPy(velocities.y, i, j, stepSizes.y)) - PuvPx(velocities,
                 ↪ i, j, stepSizes, gamma) - PvSquaredPy(velocities.y, i, j,
                 ↪ stepSizes.y, gamma) + bodyForces.y);
            }
            else if (!(flags[i][j] & SELF) && !(flags[i][j] & NORTH)) {
                FG.y[i][j] = 0;
            }
            else {
                FG.y[i][j] = velocities.y[i][j];
            }
        }
    }
}

void ComputeRHS(DoubleField FG, REAL** RHS, BYTE** flags, int iMax, int jMax, REAL
 ↪ timestep, DoubleReal stepSizes) {
    for (int i = 1; i <= iMax; ++i) {
        for (int j = 1; j <= jMax; ++j) {
            if (!(flags[i][j] & SELF)) { // RHS is defined in the middle of cells,
             ↪ so only check the SELF bit
                continue; // Skip if the cell is not a fluid cell
            }
            RHS[i][j] = (1 / timestep) * (((FG.x[i][j] - FG.x[i - 1][j]) /
             ↪ stepSizes.x) + ((FG.y[i][j] - FG.y[i][j - 1]) / stepSizes.y));
        }
    }
}

void ComputeTimestep(REAL& timestep, int iMax, int jMax, DoubleReal stepSizes,
 ↪ DoubleField velocities, REAL reynoldsNo, REAL safetyFactor) {
    REAL inverseSquareRestriction = (REAL)0.5 * reynoldsNo * (1 / (stepSizes.x *
     ↪ stepSizes.x) + 1 / (stepSizes.y * stepSizes.y));
    REAL xTravelRestriction = stepSizes.x / FieldMax(velocities.x, iMax, jMax);
    REAL yTravelRestriction = stepSizes.y / FieldMax(velocities.y, iMax, jMax);
    REAL smallestRestriction = inverseSquareRestriction; // Choose the smallest
    ↪ restriction
    if (xTravelRestriction < smallestRestriction) {
        smallestRestriction = xTravelRestriction;
    }
    if (yTravelRestriction < smallestRestriction) {
        smallestRestriction = yTravelRestriction;
    }
    timestep = safetyFactor * smallestRestriction;
}

void PoissonSubset(REAL** pressure, REAL** RHS, BYTE** flags, int xOffset, int
↪ yOffset, int iMax, int jMax, DoubleReal stepSizes, REAL omega, REAL
↪ boundaryFraction, REAL& residualNormSquare) {
    for (int i = xOffset + 1; i <= iMax; i++) {
        for (int j = yOffset + 1; j <= jMax; j++) {
            if (!(flags[i][j] & SELF)) { // Pressure is defined in the middle of
            ↪ cells, so only check the SELF bit
                continue; // Skip if the cell is not a fluid cell
            }
            REAL relaxedPressure = (1 - omega) * pressure[i][j];
            REAL pressureAverages = ((pressure[i + 1][j] + pressure[i - 1][j]) /
            ↪ square(stepSizes.x)) + ((pressure[i][j + 1] + pressure[i][j - 1]) /
            ↪ square(stepSizes.y)) - RHS[i][j];

            pressure[i][j] = relaxedPressure + boundaryFraction * pressureAverages;
            residualNormSquare += square(pressureAverages - (2 * pressure[i][j]) /
            ↪ square(stepSizes.x) - (2 * pressure[i][j]) / square(stepSizes.y));
        }
    }
}

void ThreadLoop(REAL** pressure, REAL** RHS, BYTE** flags, int xOffset, int yOffset,
↪ int iMax, int jMax, DoubleReal stepSizes, REAL omega, REAL boundaryFraction,
↪ REAL& residualNormSquare, ThreadStatus& threadStatus) {
    while (!threadStatus.stopRequested) { // Condition to stop the thread entirely
        std::cout << "Thread waiting\n";
        while (!threadStatus.startNextIterationRequested) { // Wait until the next
        ↪ iteration is requested
            if (threadStatus.stopRequested) { // If a request to stop occurs in this
            ↪ loop, do not complete another iteration.
                return;
            }
        }
        std::cout << "Thread running\n";
        threadStatus.running = true;
        threadStatus.startNextIterationRequested = false; // Set it to false so that
        ↪ only 1 iteration occurs if there is no input from thread owner
        for (int i = xOffset + 1; i <= iMax; i++) {
            for (int j = yOffset + 1; j <= jMax; j++) {
                if (!(flags[i][j] & SELF)) { // Pressure is defined in the middle of
                ↪ cells, so only check the SELF bit
```

```
                    continue; // Skip if the cell is not a fluid cell
                }
                REAL relaxedPressure = (1 - omega) * pressure[i][j];
                REAL pressureAverages = ((pressure[i + 1][j] + pressure[i - 1][j]) /
                ↪   square(stepSizes.x)) + ((pressure[i][j + 1] + pressure[i][j -
                ↪   1]) / square(stepSizes.y)) - RHS[i][j];

                pressure[i][j] = relaxedPressure + boundaryFraction *
                ↪   pressureAverages;
                residualNormSquare = square(pressureAverages - (2 * pressure[i][j])
                ↪   / square(stepSizes.x) - (2 * pressure[i][j]) /
                ↪   square(stepSizes.y));
            }
        }
        threadStatus.running = false;
    }
}

int PoissonThreadPool(REAL** pressure, REAL** RHS, BYTE** flags, std::pair<int,
↪   int>* coordinates, int coordinatesLength, int numFluidCells, int iMax, int jMax,
↪   DoubleReal stepSizes, REAL residualTolerance, int minIterations, int
↪   maxIterations, REAL omega, REAL& residualNorm) {
    int currentIteration = 0;
    REAL boundaryFraction = omega / ((2 / square(stepSizes.x)) + (2 /
    ↪   square(stepSizes.y)));

    int totalThreads = std::thread::hardware_concurrency(); // Number of threads
    ↪   returned by hardware, may not be reliable and may be 0 in error case
    int xBlocks, yBlocks; // Number of blocks in the x and y direction

    if (totalThreads % 4 == 0 && totalThreads > 4) { // Encompasses most
    ↪   multi-threaded CPUs (even number of cores, 2 threads per core)
        yBlocks = 4;
        xBlocks = totalThreads / 4;
    }
    else if (totalThreads % 2 == 0 && totalThreads > 2) { // Hopefully a catch-all
    ↪   case given all modern CPUs have even numbers of cores
        yBlocks = 2;
        xBlocks = totalThreads / 2;
    }
    else { // threadHint is odd or 0
        totalThreads = 1;
        yBlocks = 1;
        xBlocks = 1;
    }

    // Initialise the threads to use, which at this point will be sitting in a loop
    ↪   waiting for the next iteration request
    REAL* residualNorms = new REAL[totalThreads]();
    std::thread* threads = new std::thread[totalThreads]; // Array of all running
    ↪   threads, heap allocated because size is runtime-determined
    ThreadStatus* threadStatuses = new ThreadStatus[totalThreads]();
    int threadNum = 0;
    for (int xBlock = 0; xBlock < xBlocks; xBlock++) {
        for (int yBlock = 0; yBlock < yBlocks; yBlock++) {
            threads[threadNum] = std::thread(ThreadLoop, pressure, RHS, flags, (iMax
             ↪  * xBlock) / xBlocks, (jMax * yBlock) / yBlocks, (iMax * (xBlock +
             ↪  1)) / xBlocks, (jMax * (yBlock + 1)) / yBlocks, stepSizes, omega,
             ↪  boundaryFraction, std::ref(residualNorms[threadNum]),
             ↪  std::ref(threadStatuses[threadNum]));
            threadNum++;
        }
    }
    do {
        residualNorm = 0;

        // Dispach threads and perform computation
        for (int threadNum = 0; threadNum < totalThreads; threadNum++) {
            threadStatuses[threadNum].startNextIterationRequested = true; // Loop
             ↪  through the threads and start the iteration
            threadStatuses[threadNum].running = true; // TESTING
        }


        // Wait for threads to finish exection
        for (int threadNum = 0; threadNum < totalThreads; threadNum++) {
            while (threadStatuses[threadNum].running) {} // Wait until the current
             ↪  thread stops running
            residualNorm += residualNorms[threadNum];
        }


        CopyBoundaryPressures(pressure, coordinates, coordinatesLength, flags, iMax,
         ↪  jMax);
        residualNorm = sqrt(residualNorm) / (numFluidCells);
        currentIteration++;
    } while ((currentIteration < maxIterations && residualNorm > residualTolerance)
     ↪  || currentIteration < minIterations);

    // Stop and join the threads
    for (int threadNum = 0; threadNum < totalThreads; threadNum++) {
        threadStatuses[threadNum].stopRequested = true; // Request for stop
        threads[threadNum].join(); // And wait for it to actually stop
    }

    delete[] threadStatuses;
    delete[] threads;
    delete[] residualNorms;
    return currentIteration;
}

int PoissonMultiThreaded(REAL** pressure, REAL** RHS, BYTE** flags, std::pair<int,
 ↪  int>* coordinates, int coordinatesLength, int numFluidCells, int iMax, int jMax,
 ↪  DoubleReal stepSizes, REAL residualTolerance, int minIterations, int
 ↪  maxIterations, REAL omega, REAL& residualNorm) {
    int currentIteration = 0;
    REAL boundaryFraction = omega / ((2 / square(stepSizes.x)) + (2 /
    ↪  square(stepSizes.y)));

    int threadHint = std::thread::hardware_concurrency(); // Number of threads
    ↪  returned by hardware, may not be reliable and may be 0 in error case
    int xBlocks, yBlocks; // Number of blocks in the x and y direction

    if (threadHint % 4 == 0 && threadHint > 4) { // Encompasses most multi-threaded
    ↪  CPUs (even number of cores, 2 threads per core)
        yBlocks = 4;
        xBlocks = threadHint / 4;
    }
    else if (threadHint % 2 == 0 && threadHint > 2) { // Hopefully a catch-all case
    ↪  given all modern CPUs have even numbers of cores
        yBlocks = 2;
        xBlocks = threadHint / 2;
    }
    else { // threadHint is odd or 0
        if (threadHint == 0) threadHint = 1;
        yBlocks = 1;
        xBlocks = 1;
    }
    REAL* residualNorms = new REAL[xBlocks * yBlocks]();

    do {
        CopyBoundaryPressures(pressure, coordinates, coordinatesLength, flags, iMax,
        ↪  jMax);

        residualNorm = 0;
#ifdef DEBUGOUT
        if (currentIteration % 100 == 0)
        {
            std::cout << "Pressure iteration " << currentIteration << std::endl; //
            ↪  DEBUGGING
        }
#endif // DEBUGOUT
        // Dispach threads and perform computation
        std::thread* threads = new std::thread[xBlocks * yBlocks]; // Array of all
        ↪  running threads, heap allocated because size is runtime-determined
        int threadNum = 0;
        for (int xBlock = 0; xBlock < xBlocks; xBlock++) {
            for (int yBlock = 0; yBlock < yBlocks; yBlock++) {
                threads[threadNum] = std::thread(PoissonSubset, pressure, RHS,
                ↪  flags, (iMax * xBlock) / xBlocks, (jMax * yBlock) / yBlocks,
                ↪  (iMax * (xBlock + 1)) / xBlocks, (jMax * (yBlock + 1)) /
                ↪  yBlocks, stepSizes, omega, boundaryFraction,
                ↪  std::ref(residualNorms[threadNum]));
                threadNum++;
            }
        }


        // Wait for threads to finish exection
        for (int threadNum = 0; threadNum < xBlocks * yBlocks; threadNum++) {
            threads[threadNum].join();
        }

        residualNorm = ArraySum(residualNorms, xBlocks * yBlocks);

        delete[] threads;

        residualNorm = sqrt(residualNorm) / (numFluidCells);
#ifdef DEBUGOUT
        if (currentIteration % 100 == 0)
        {
            std::cout << "Residual norm " << residualNorm << std::endl; //
             ↪  DEBUGGING
        }
#endif // DEBUGOUT
        currentIteration++;
    } while ((currentIteration < maxIterations && residualNorm > residualTolerance)
    ↪  || currentIteration < minIterations);
    delete[] residualNorms;
    return currentIteration;
}

int Poisson(REAL** pressure, REAL** RHS, BYTE** flags, std::pair<int, int>*
↪  coordinates, int coordinatesLength, int numFluidCells, int iMax, int jMax,
↪  DoubleReal stepSizes, REAL residualTolerance, int minIterations, int
↪  maxIterations, REAL omega, REAL &residualNorm) {
    int currentIteration = 0;
    REAL boundaryFraction = omega / ((2 / square(stepSizes.x)) + (2 /
    ↪  square(stepSizes.y)));
    do {

        residualNorm = 0;
#ifdef DEBUGOUT
        if (currentIteration % 100 == 0)
        {
            std::cout << "Pressure iteration " << currentIteration << std::endl; //
             ↪  DEBUGGING
        }
#endif // DEBUGOUT
        for (int i = 1; i <= iMax; i++) {
            for (int j = 1; j <= jMax; j++) {
                if (!(flags[i][j] & SELF)) { // Pressure is defined in the middle of
                 ↪  cells, so only check the SELF bit
                    continue; // Skip if the cell is not a fluid cell
                }
                REAL relaxedPressure = (1 - omega) * pressure[i][j];
                REAL pressureAverages = ((pressure[i + 1][j] + pressure[i - 1][j]) /
                 ↪  square(stepSizes.x)) + ((pressure[i][j + 1] + pressure[i][j -
                 ↪  1]) / square(stepSizes.y)) - RHS[i][j];

                pressure[i][j] = relaxedPressure + boundaryFraction *
                 ↪  pressureAverages;
                REAL currentResidual = pressureAverages - (2 * pressure[i][j]) /
                ↪  square(stepSizes.x) - (2 * pressure[i][j]) /
                ↪  square(stepSizes.y);
                residualNorm += square(currentResidual);
            }
        }

        residualNorm = sqrt(residualNorm / numFluidCells);
        CopyBoundaryPressures(pressure, coordinates, coordinatesLength, flags, iMax,
        ↪  jMax);
#ifdef DEBUGOUT
        if (currentIteration % 100 == 0)
        {
            std::cout << "Residual norm " << residualNorm << std::endl; //
            ↪  DEBUGGING
        }
#endif // DEBUGOUT
        currentIteration++;
    } while ((currentIteration < maxIterations && residualNorm > residualTolerance)
    ↪  || currentIteration < minIterations);
    return currentIteration;
}

void ComputeVelocities(DoubleField velocities, DoubleField FG, REAL** pressure,
↪  BYTE** flags, int iMax, int jMax, REAL timestep, DoubleReal stepSizes) {
    for (int i = 1; i <= iMax; i++) {
        for (int j = 1; j <= jMax; j++) {
            if (!(flags[i][j] & SELF)) { // If the cell is not a fluid cell, skip
            ↪  it
                continue;
            }
            if (flags[i][j] & EAST) // If the edge the velocity is defined on is a
            ↪  boundary edge, skip the calculation (this is when the cell to the
            ↪  east is not fluid)
            {
                velocities.x[i][j] = FG.x[i][j] - (timestep / stepSizes.x) *
                ↪  (pressure[i + 1][j] - pressure[i][j]);
            }
            if (flags[i][j] & NORTH) // Same, but in this case for north boundary
            {
                velocities.y[i][j] = FG.y[i][j] - (timestep / stepSizes.y) *
                ↪  (pressure[i][j + 1] - pressure[i][j]);
            }
        }
    }
}

void ComputeStream(DoubleField velocities, REAL** streamFunction, int iMax, int
↪  jMax, DoubleReal stepSizes) {
    for (int i = 0; i <= iMax; i++) {
        streamFunction[i][0] = 0; // Stream function boundary condition
        for (int j = 1; j <= jMax; j++) {
            streamFunction[i][j] = streamFunction[i][j - 1] + velocities.x[i][j] *
            ↪  stepSizes.y; // Obstacle boundary conditions are taken care of by
            ↪  the fact that u = 0 inside obstacle cells.
        }
    }
}
```

Computation.h

```
#ifndef COMPUTATION_H
#define COMPUTATION_H

#include "pch.h"

REAL FieldMax(REAL** field, int xLength, int yLength);

REAL ComputeGamma(DoubleField velocities, int iMax, int jMax, REAL timestep,
↪  DoubleReal stepSizes);

void ComputeFG(DoubleField velocities, DoubleField FG, BYTE** flags, int iMax, int
↪  jMax, REAL timestep, DoubleReal stepSizes, DoubleReal bodyForces, REAL gamma,
↪  REAL reynoldsNo);

void ComputeRHS(DoubleField FG, REAL** RHS, BYTE** flags, int iMax, int jMax, REAL
↪  timestep, DoubleReal stepSizes);

void ComputeTimestep(REAL& timestep, int iMax, int jMax, DoubleReal stepSizes,
↪  DoubleField velocities, REAL reynoldsNo, REAL safetyFactor);

void PoissonSubset(REAL** pressure, REAL** RHS, BYTE** flags, int xOffset, int
↪  yOffset, int iMax, int jMax, DoubleReal stepSizes, REAL omega, REAL
↪  boundaryFraction, REAL& residualNormSquare);

void ThreadLoop(REAL** pressure, REAL** RHS, BYTE** flags, int xOffset, int yOffset,
↪  int iMax, int jMax, DoubleReal stepSizes, REAL omega, REAL boundaryFraction,
↪  REAL& residualNormSquare, ThreadStatus& threadStatus);

int PoissonThreadPool(REAL** pressure, REAL** RHS, BYTE** flags, std::pair<int,
↪  int>* coordinates, int coordinatesLength, int numFluidCells, int iMax, int jMax,
↪  DoubleReal stepSizes, REAL residualTolerance, int minIterations, int
↪  maxIterations, REAL omega, REAL& residualNorm);

int PoissonMultiThreaded(REAL** pressure, REAL** RHS, BYTE** flags, std::pair<int,
↪  int>* coordinates, int coordinatesLength, int numFluidCells, int iMax, int jMax,
↪  DoubleReal stepSizes, REAL residualTolerance, int minIterations, int
↪  maxIterations, REAL omega, REAL& residualNorm);

int Poisson(REAL** pressure, REAL** RHS, BYTE** flags, std::pair<int, int>*
↪  coordinates, int coordinatesLength, int numFluidCells, int iMax, int jMax,
↪  DoubleReal stepSizes, REAL residualTolerance, int minIterations, int
↪  maxIterations, REAL omega, REAL& residualNorm);
```

```
void ComputeVelocities(DoubleField velocities, DoubleField FG, REAL** pressure,
↪ BYTE** flags, int iMax, int jMax, REAL timestep, DoubleReal stepSizes);

void ComputeStream(DoubleField velocities, REAL** streamFunction, int iMax, int
↪ jMax, DoubleReal stepSizes);

#endif
```

CPUBackend.cpp

```
#include "pch.h"
#include "Solver.h"
#include "CPUSolver.h"
#include "BackendCoordinator.h"
#include <iostream>

//#define WAIT_FOR_DEBUGGER_ATTACH

int main(int argc, char** argv) {
#ifdef WAIT_FOR_DEBUGGER_ATTACH
    char nonsense;
    std::cout << "Press a character and press enter once debugger is attched. ";
    std::cin >> nonsense;
#endif // WAIT_FOR_DEBUGGER_ATTACH

    SimulationParameters parameters = SimulationParameters();
    if (argc == 1 || (argc == 2 && strcmp(argv[1], "debug") == 0)) { // Not linked
    ↪ to a frontend.
        std::cout << "Running without a fronted attached.\n";
        int iMax = 200;
        int jMax = 100;
        parameters.width = 1;
        parameters.height = 1;
        parameters.timeStepSafetyFactor = (REAL)0.5;
        parameters.relaxationParameter = (REAL)1.7;
        parameters.pressureResidualTolerance = 2;
        parameters.pressureMinIterations = 5;
        parameters.pressureMaxIterations = 1000;
        parameters.reynoldsNo = 2000;
        parameters.dynamicViscosity = (REAL)0.00001983;
        parameters.fluidDensity = (REAL)1.293;
        parameters.inflowVelocity = 5;
        parameters.surfaceFrictionalPermissibility = 0;
        parameters.bodyForces.x = 0;
        parameters.bodyForces.y = 0;

        CPUSolver solver = CPUSolver(parameters, iMax, jMax);

        bool** obstacles = solver.GetObstacles();
        for (int i = 1; i <= iMax; i++) { for (int j = 1; j <= jMax; j++) {
        ↪ obstacles[i][j] = 1; } } // Set all the cells to fluid

        int boundaryLeft = (int)(0.45 * iMax);
        int boundaryRight = (int)(0.45 * iMax + 2);
        int boundaryBottom = (int)(0.45 * jMax);
        int boundaryTop = (int)(0.55 * jMax);
        for (int i = boundaryLeft; i < boundaryRight; i++) { // Create a square of
        ↪   boundary cells
            for (int j = boundaryBottom; j < boundaryTop; j++) {
                obstacles[i][j] = 0;
            }
        }

        std::cout << "Obstacle set to a line." << std::endl;

        solver.ProcessObstacles();
        solver.PerformSetup();

        REAL cumulativeTimestep = 0;

        int numIterations = 0;
        std::cout << "Enter number of iterations: ";
        std::cin >> numIterations;

        for (int i = 0; i < numIterations; i++) {
            solver.Timestep(cumulativeTimestep);
            REAL dragCoefficient = solver.GetDragCoefficient();
            std::cout << "Iteration " << i << ", time taken: " << cumulativeTimestep
            ↪   << ", drag coefficient " << dragCoefficient << ".\n";
        }
        return 0;
    }
    else if (argc == 4) { // Linked to a frontend.
        std::cout << "Linked to a frontend.\n";

        char* pipeName = argv[1];
        int iMax = atoi(argv[2]);
        int jMax = atoi(argv[3]);
        if (iMax == 0 || jMax == 0) { // Set defaults
            iMax = 200;
            jMax = 100;
        }
        Solver* solver = new CPUSolver(parameters, iMax, jMax);
        BackendCoordinator backendCoordinator(iMax, jMax, std::string(pipeName),
        ↪   solver);
        int retValue = backendCoordinator.Run();
        delete solver;
        return retValue;
    }
    else {
        std::cerr << "Incorrect number of command-line arguments. Run the executable
        ↪   with the pipe name and field dimensions to connect to a frontend, or
        ↪   without to run without a frontend.\n";
        return -1;
    }
}
```

CPUSolver.cpp

```
#include "CPUSolver.h"
#include "Init.h"
#include "Boundary.h"
#include "Flags.h"
#include "Computation.h"
#include "Drag.h"

CPUSolver::CPUSolver(SimulationParameters parameters, int iMax, int jMax) :
↪  Solver(parameters, iMax, jMax) {
    velocities.x = MatrixMAlloc(iMax + 2, jMax + 2);
    velocities.y = MatrixMAlloc(iMax + 2, jMax + 2);

    pressure = MatrixMAlloc(iMax + 2, jMax + 2);
    RHS = MatrixMAlloc(iMax + 2, jMax + 2);
    streamFunction = MatrixMAlloc(iMax + 1, jMax + 1);

    FG.x = MatrixMAlloc(iMax + 2, jMax + 2);
    FG.y = MatrixMAlloc(iMax + 2, jMax + 2);

    obstacles = ObstacleMatrixMAlloc(iMax + 2, jMax + 2);
    flags = FlagMatrixMAlloc(iMax + 2, jMax + 2);

    flattenedHVel = new REAL[iMax * jMax];
    flattenedVVel = new REAL[iMax * jMax];
    flattenedPressure = new REAL[iMax * jMax];
    flattenedStream = new REAL[iMax * jMax];

    coordinates = nullptr;
    coordinatesLength = 0;
    numFluidCells = 0;
    stepSizes = DoubleReal(0, 0);
}

CPUSolver::~CPUSolver() {
    FreeMatrix(velocities.x, iMax + 2);
    FreeMatrix(velocities.y, iMax + 2);
    FreeMatrix(pressure, iMax + 2);
    FreeMatrix(RHS, iMax + 2);
    FreeMatrix(streamFunction, iMax + 1);
    FreeMatrix(FG.x, iMax + 2);
    FreeMatrix(FG.y, iMax + 2);
    FreeMatrix(obstacles, iMax + 2);
    FreeMatrix(flags, iMax + 2);

    delete[] flattenedHVel;
    delete[] flattenedVVel;
    delete[] flattenedPressure;
    delete[] flattenedStream;
}

REAL* CPUSolver::GetHorizontalVelocity() const {
    return flattenedHVel;
```

```
}

REAL* CPUSolver::GetVerticalVelocity() const {
    return flattenedVVel;
}

REAL* CPUSolver::GetPressure() const {
    return flattenedPressure;
}

REAL* CPUSolver::GetStreamFunction() const {
    return flattenedStream;
}

bool** CPUSolver::GetObstacles() const {
    return obstacles;
}

REAL CPUSolver::GetDragCoefficient() {
    return ComputeDragCoefficient(velocities, pressure, flags, coordinates,
    ↪   coordinatesLength, iMax, jMax, stepSizes, parameters.dynamicViscosity,
    ↪   parameters.fluidDensity, parameters.inflowVelocity);
}

void CPUSolver::ProcessObstacles() {
    SetFlags(obstacles, flags, iMax + 2, jMax + 2);

    std::pair<std::pair<int, int>*, int> coordinatesWithLength =
    ↪   FindBoundaryCells(flags, iMax, jMax);
    coordinates = coordinatesWithLength.first;
    coordinatesLength = coordinatesWithLength.second;

    numFluidCells = CountFluidCells(flags, iMax, jMax);
}

void CPUSolver::PerformSetup() {
    stepSizes.x = parameters.width / iMax;
    stepSizes.y = parameters.height / jMax;
}

void CPUSolver::Timestep(REAL& simulationTime) {
    SetBoundaryConditions(velocities, flags, coordinates, coordinatesLength, iMax,
    ↪   jMax, parameters.inflowVelocity,
    ↪   parameters.surfaceFrictionalPermissibility);

    REAL timestep;
    ComputeTimestep(timestep, iMax, jMax, stepSizes, velocities,
    ↪   parameters.reynoldsNo, parameters.timeStepSafetyFactor);
    simulationTime += timestep;

    REAL gamma = ComputeGamma(velocities, iMax, jMax, timestep, stepSizes);
    ComputeFG(velocities, FG, flags, iMax, jMax, timestep, stepSizes,
    ↪   parameters.bodyForces, gamma, parameters.reynoldsNo);
    ComputeRHS(FG, RHS, flags, iMax, jMax, timestep, stepSizes);
    REAL pressureResidualNorm = 0;
    (void)PoissonMultiThreaded(pressure, RHS, flags, coordinates, coordinatesLength,
    ↪   numFluidCells, iMax, jMax, stepSizes, parameters.pressureResidualTolerance,
    ↪   parameters.pressureMinIterations, parameters.pressureMaxIterations,
    ↪   parameters.relaxationParameter, pressureResidualNorm);

    ComputeVelocities(velocities, FG, pressure, flags, iMax, jMax, timestep,
    ↪   stepSizes);
    ComputeStream(velocities, streamFunction, iMax, jMax, stepSizes);

    // Copy all of the 2D arrays to flattened arrays.
    // Parameters:     2D array        2D array offsets/flattened array and
    ↪   offsets/size of copy domain
    FlattenArray<REAL>(velocities.x,   1, 1,            flattenedHVel,     0, 0, 0,
    ↪   iMax, jMax);
    FlattenArray<REAL>(velocities.y,   1, 1,            flattenedVVel,     0, 0, 0,
    ↪   iMax, jMax);
    FlattenArray<REAL>(pressure,       1, 1,            flattenedPressure, 0, 0, 0,
    ↪   iMax, jMax);
    FlattenArray<REAL>(streamFunction, 0, 0,            flattenedStream,   0, 0, 0,
    ↪   iMax, jMax);
}
```

CPUSolver.h

```
#ifndef CPUSOLVER_H
#define CPUSOLVER_H

#include "Solver.h"
class CPUSolver :
    public Solver
{
private:
    DoubleField velocities;
    REAL** pressure;
    REAL** RHS;
    REAL** streamFunction;
    DoubleField FG;

    REAL* flattenedHVel;
    REAL* flattenedVVel;
    REAL* flattenedPressure;
    REAL* flattenedStream;

    DoubleReal stepSizes;

    bool** obstacles;
    BYTE** flags;
    std::pair<int, int>* coordinates;
    int coordinatesLength;
    int numFluidCells;
public:
    CPUSolver(SimulationParameters parameters, int iMax, int jMax);

    ~CPUSolver();

    bool** GetObstacles() const;

    REAL* GetHorizontalVelocity() const;

    REAL* GetVerticalVelocity() const;

    REAL* GetPressure() const;

    REAL* GetStreamFunction() const;

    REAL GetDragCoefficient();

    void ProcessObstacles();

    void PerformSetup();

    void Timestep(REAL& simulationTime); // Implementing abstract inherited method
};

#endif // !CPUSOLVER_H
```

DiscreteDerivatives.cpp

```
#include "DiscreteDerivatives.h"
#include <cmath>

/*
A note on terminology:
Below are functions to represent the calculations of different derivatives used in
↪  the Navier-Stokes equations. They have been discretised.
Average: the sum of 2 quantities, then divided by 2. Taking the mean of the 2
↪  quantities.
Difference: The same as above, but with subtraction.
Forward: applying an average or difference between the current cell (i,j) and the
↪  next cell along (i+1,j) or (i,j+1)
Backward: the same as above, but applied to the cell behind - (i-1,j) or (i,j-1).
Downshift: Any of the above with respect to the cell below the current one, (i,
↪  j-1).
Second derivative: the double application of a derivative.
Donor and non-donor: There are 2 different discretisation methods here, one of which
↪  is donor-cell discretisation. The 2 parts of each discretisation formula are
↪  named as such.
*/

REAL PuPx(REAL** hVel, int i, int j, REAL delx) { // NOTE: P here is used to
↪  represent the partial operator, so PuPx should be read "partial u by partial x"
        return (hVel[i][j] - hVel[i - 1][j]) / delx;
}
```

```
REAL PvPy(REAL** vVel, int i, int j, REAL dely) {
        return (vVel[i][j] - vVel[i][j - 1]) / dely;
}

REAL PuSquaredPx(REAL** hVel, int i, int j, REAL delx, REAL gamma) {
        REAL forwardAverage = (hVel[i][j] + hVel[i + 1][j]) / 2;
        REAL backwardAverage = (hVel[i - 1][j] + hVel[i][j]) / 2;

        REAL forwardDifference = (hVel[i][j] - hVel[i + 1][j]) / 2;
        REAL backwardDifference = (hVel[i - 1][j] - hVel[i][j]) / 2;

        REAL nonDonorTerm = (1 / delx) * (square(forwardAverage) -
        ↪   square(backwardAverage));
        REAL donorTerm = (gamma / delx) * ((abs(forwardAverage) * forwardDifference)
        ↪   - (abs(backwardAverage) * backwardDifference));

        return nonDonorTerm + donorTerm;
}

REAL PvSquaredPy(REAL** vVel, int i, int j, REAL dely, REAL gamma) {
        REAL forwardAverage = (vVel[i][j] + vVel[i][j + 1]) / 2;
        REAL backwardAverage = (vVel[i][j - 1] + vVel[i][j]) / 2;

        REAL forwardDifference = (vVel[i][j] - vVel[i][j + 1]) / 2;
        REAL backwardDifference = (vVel[i][j - 1] - vVel[i][j]) / 2;

        REAL nonDonorTerm = (1 / dely) * (square(forwardAverage) -
        ↪   square(backwardAverage));
        REAL donorTerm = (gamma / dely) * ((abs(forwardAverage) * forwardDifference)
        ↪   - (abs(backwardAverage) * backwardDifference));
        return nonDonorTerm + donorTerm;
}

REAL PuvPx(DoubleField velocities, int i, int j, DoubleReal stepSizes, REAL gamma) {
        REAL jForwardAverageU = (velocities.x[i][j] + velocities.x[i][j + 1]) / 2;
        REAL iForwardAverageV = (velocities.y[i][j] + velocities.y[i + 1][j]) / 2;
        REAL iBackwardAverageV = (velocities.y[i - 1][j] + velocities.y[i][j]) / 2;

        REAL jForwardAverageUDownshift = (velocities.x[i - 1][j] + velocities.x[i -
        ↪   1][j + 1]) / 2;

        REAL iForwardDifferenceV = (velocities.y[i][j] - velocities.y[i + 1][j]) /
        ↪   2;
        REAL iBackwardDifferenceV = (velocities.y[i - 1][j] - velocities.y[i][j]) /
        ↪   2;

        REAL nonDonorTerm = (1 / stepSizes.x) * ((jForwardAverageU *
        ↪   iForwardAverageV) - (jForwardAverageUDownshift * iBackwardAverageV));
        REAL donorTerm = (gamma / stepSizes.x) * ((abs(jForwardAverageU) *
        ↪   iForwardDifferenceV) - (abs(jForwardAverageUDownshift) *
        ↪   iBackwardDifferenceV));
        return nonDonorTerm + donorTerm;
}
```

```
REAL PuvPy(DoubleField velocities, int i, int j, DoubleReal stepSizes, REAL gamma) {
        REAL iForwardAverageV = (velocities.y[i][j] + velocities.y[i + 1][j]) / 2;
        REAL jForwardAverageU = (velocities.x[i][j] + velocities.x[i][j + 1]) / 2;
        REAL jBackwardAverageU = (velocities.x[i][j - 1] + velocities.x[i][j]) / 2;

        REAL iForwardAverageVDownshift = (velocities.y[i][j - 1] + velocities.y[i +
        ↪   1][j - 1]) / 2;

        REAL jForwardDifferenceU = (velocities.x[i][j] - velocities.x[i][j + 1]) /
        ↪   2;
        REAL jBackwardDifferenceU = (velocities.x[i][j - 1] - velocities.x[i][j]) /
        ↪   2;

        REAL nonDonorTerm = (1 / stepSizes.y) * ((iForwardAverageV *
        ↪   jForwardAverageU) - (iForwardAverageVDownshift * jBackwardAverageU));
        REAL donorTerm = (gamma / stepSizes.y) * ((abs(iForwardAverageV) *
        ↪   jForwardDifferenceU) - (abs(iForwardAverageVDownshift) *
        ↪   jBackwardDifferenceU));

        return nonDonorTerm + donorTerm;
}

REAL SecondPuPx(REAL** hVel, int i, int j, REAL delx) {
        return (hVel[i + 1][j] - 2 * hVel[i][j] + hVel[i - 1][j]) / square(delx);
}

REAL SecondPuPy(REAL** hVel, int i, int j, REAL dely) {
        return (hVel[i][j + 1] - 2 * hVel[i][j] + hVel[i][j - 1]) / square(dely);
}

REAL SecondPvPx(REAL** vVel, int i, int j, REAL delx) {
        return (vVel[i + 1][j] - 2 * vVel[i][j] + vVel[i - 1][j]) / square(delx);
}

REAL SecondPvPy(REAL** vVel, int i, int j, REAL dely) {
        return (vVel[i][j + 1] - 2 * vVel[i][j] + vVel[i][j - 1]) / square(dely);
}

REAL PpPx(REAL** pressure, int i, int j, REAL delx) {
        return (pressure[i + 1][j] - pressure[i][j]) / delx;
}

REAL PpPy(REAL** pressure, int i, int j, REAL dely) {
        return (pressure[i][j + 1] - pressure[i][j]) / dely;
}

REAL square(REAL operand) {
        return pow(operand, (REAL)2);
}
```

DiscreteDerivatives.h

```
#ifndef DISCRETE_DERIVATIVES_H
#define DISCRETE_DERIVATIVES_H

#include "pch.h"

REAL PuPx(REAL** hVel, int i, int j, REAL delx);

REAL PvPy(REAL** vVel, int i, int j, REAL dely);

REAL PuSquaredPx(REAL** hVel, int i, int j, REAL delx, REAL gamma);

REAL PvSquaredPy(REAL** vVel, int i, int j, REAL dely, REAL gamma);

REAL PuvPx(DoubleField velocities, int i, int j, DoubleReal stepSizes, REAL gamma);

REAL PuvPy(DoubleField velocities, int i, int j, DoubleReal stepSizes, REAL gamma);

REAL SecondPuPx(REAL** hVel, int i, int j, REAL delx);

REAL SecondPuPy(REAL** hVel, int i, int j, REAL dely);

REAL SecondPvPx(REAL** vVel, int i, int j, REAL delx);

REAL SecondPvPy(REAL** vVel, int i, int j, REAL dely);

REAL PpPx(REAL** pressure, int i, int j, REAL delx);

REAL PpPy(REAL** pressure, int i, int j, REAL dely);

REAL square(REAL operand);

#endif // !DISCRETE_DERIVATIVES_CUH
```

Drag.cpp

```
#include "Drag.h"
#include "Math.h"
#include <bitset>
#include <cstdio> // TESTING

REAL Magnitude(REAL x, REAL y) {
    return sqrtf(x * x + y * y);
}

REAL Dot(DoubleReal left, DoubleReal right) {
    return left.x * right.x + left.y * right.y;
}

DoubleReal GetUnitVector(DoubleReal vector) {
    REAL magnitude = Magnitude(vector.x, vector.y);
    return DoubleReal(vector.x / magnitude, vector.y / magnitude);
}
```

```
/// <summary>
/// Calculates the partial derivative of the velocity field with respect to the
↪  distance from a point.
/// </summary>
/// <param name="unitVector">The direction vector to extend out from the
↪  point.</param>
/// <param name="distance">The separation of the points where velocity is
↪  taken</param>
/// <returns>The partial derivative of velocity with respect to distance from a
↪  point.</returns>
REAL PVPd(DoubleField velocities, REAL distance, int iStart, int jStart, int
↪  iExtended, int jExtended) { // As with DiscreteDerivatives, read this as
↪  "Partial V over Partial d" - Partial derivative of velocity wrt distance from a
↪  point.
    REAL extendedVelocityMagnitude = Magnitude(velocities.x[iExtended][jExtended],
    ↪  velocities.y[iExtended][jExtended]);
    REAL surfaceVelocityMagnitude = Magnitude(velocities.x[iStart][jStart],
    ↪  velocities.y[iStart][jStart]);
    return (extendedVelocityMagnitude - surfaceVelocityMagnitude) / distance;
}

/// <summary>
/// Computes wall shear stress for a single boundary cell.
/// </summary>
/// <param name="unitNormal">The unit vector perpendicular to the direction of the
↪  surface at the cell.</param>
/// <returns>The magnitude of the wall shear stress for one boundary
↪  cell.</returns>
REAL ComputeWallShear(DoubleReal& shearUnitVector, DoubleField velocities,
↪  DoubleReal unitNormal, int i, int j, DoubleReal stepSizes, REAL viscosity) {
    int iExtended, jExtended;
    REAL distance;
    if (unitNormal.x == 0 || unitNormal.y == 0) { // Parallel to an axis
        if (unitNormal.x == 0) {
            distance = stepSizes.x;
        }
        else {
            distance = stepSizes.y;
        }
        iExtended = i + (int)unitNormal.x;
        jExtended = j + (int)unitNormal.y;
    }
    else { // 45 degrees to an axis.
        distance = Magnitude(unitNormal.x * stepSizes.x, unitNormal.y * stepSizes.y)
        ↪  * DIAGONAL_CELL_DISTANCE;
        iExtended = i + (int)roundf(unitNormal.x * DIAGONAL_CELL_DISTANCE) * 2;
        jExtended = j + (int)roundf(unitNormal.y * DIAGONAL_CELL_DISTANCE) * 2;
    }
    shearUnitVector = DoubleReal(velocities.x[iExtended][jExtended],
    ↪  velocities.y[iExtended][jExtended]);
    return viscosity * PVPd(velocities, distance, i, j, iExtended, jExtended);
}
```

```
/// <summary>
/// Computes the viscous drag on the obstacle.
/// </summary>
/// <returns>The magnitude of the viscous drag on the obstacle.</returns>
REAL ComputeViscousDrag(DoubleField velocities, BYTE** flags, std::pair<int, int>*
↪  coordinates, int coordinatesLength, int iMax, int jMax, DoubleReal stepSizes,
↪  DoubleReal fluidVector, REAL viscosity) { // Calculates the cells that make up
↪  the surface and calls ComputeWallShear on each.
    REAL totalViscousDrag = 0;
    for (int coordinateNum = 0; coordinateNum < coordinatesLength; coordinateNum++)
    ↪  {
        std::pair<int, int> coordinate = coordinates[coordinateNum];
        BYTE flag = flags[coordinate.first][coordinate.second];

        BYTE northBit = (flag & NORTH) >> NORTHSHIFT;
        BYTE eastBit = (flag & EAST) >> EASTSHIFT;
        BYTE southBit = (flag & SOUTH) >> SOUTHSHIFT;
        BYTE westBit = (flag & WEST) >> WESTSHIFT;
        int numEdges = (int)std::bitset<8>(flag).count();

        int i = coordinate.first - westBit;
        int j = coordinate.second - southBit;
        DoubleReal unitNormal = DoubleReal((REAL)(eastBit - westBit),
        ↪  (REAL)(northBit - southBit));
        REAL stepSize;
        if (numEdges == 2) {
            unitNormal.x /= DIAGONAL_CELL_DISTANCE;
            unitNormal.y /= DIAGONAL_CELL_DISTANCE;
            stepSize = (stepSizes.x + stepSizes.y) / 2;
        }
        else if ((eastBit | westBit) == 1) {
            stepSize = stepSizes.x;
        }
        else {
            stepSize = stepSizes.y;

        }
        DoubleReal shearDirection;
        REAL wallShear = ComputeWallShear(shearDirection, velocities, unitNormal, i,
        ↪  j, stepSizes, viscosity);

        totalViscousDrag += wallShear * -Dot(fluidVector, shearDirection) *
        ↪  stepSize;
    }

    return totalViscousDrag;
}

/// <summary>
/// Computes a baseline pressure by taking an average of all 4 corners' pressures.
/// </summary>
/// <returns>The average of the pressures in the 4 corners.</returns>
```

```
REAL ComputeBaselinePressure(REAL** pressure, int iMax, int jMax) {
    return (pressure[1][1] + pressure[1][jMax] + pressure[iMax][1] +
     ↪   pressure[iMax][jMax]) / 4;
}

REAL PressureIntegrand(REAL pressure, REAL baselinePressure, DoubleReal unitNormal,
 ↪  DoubleReal fluidVector) {
    return (pressure - baselinePressure) * Dot(unitNormal, fluidVector);
}

/// <summary>
/// Computes the pressure drag on the obstacle. Assumes fluid flowing left to
 ↪  right.
/// </summary>
/// <returns>The magnitude of the pressure drag on the obstacle.</returns>
REAL ComputePressureDrag(REAL** pressure, BYTE** flags, std::pair<int, int>*
 ↪  coordinates, int coordinatesLength, int iMax, int jMax, DoubleReal stepSizes,
 ↪  DoubleReal fluidVector) {
    REAL totalPresureDrag = 0;
    REAL baselinePressure = ComputeBaselinePressure(pressure, iMax, jMax);
    printf("Baseline pressure: %f.\n", baselinePressure);
    for (int coordinateNum = 0; coordinateNum < coordinatesLength; coordinateNum++)
     ↪   {
        std::pair<int, int> coordinate = coordinates[coordinateNum];
        BYTE flag = flags[coordinate.first][coordinate.second];

        BYTE northBit = (flag & NORTH) >> NORTHSHIFT;
        BYTE eastBit  = (flag & EAST)  >> EASTSHIFT;
        BYTE southBit = (flag & SOUTH) >> SOUTHSHIFT;
        BYTE westBit  = (flag & WEST)  >> WESTSHIFT;
        int numEdges = (int)std::bitset<8>(flag).count();

        if (numEdges == 2) { // Corner cell - compute the pressure integrand for the
         ↪   3 fluid cells around it
            int xDirection = eastBit - westBit;
            int yDirection = northBit - southBit;
            DoubleReal unitNormal;

            // Cell 1: east / west
            unitNormal = DoubleReal((REAL)xDirection, 0);
            totalPresureDrag += PressureIntegrand(pressure[coordinate.first +
             ↪   xDirection][coordinate.second], baselinePressure, unitNormal,
             ↪   fluidVector) * stepSizes.x;

            // Cell 2: north / south
            unitNormal = DoubleReal(0, (REAL)yDirection);
            totalPresureDrag +=
             ↪   PressureIntegrand(pressure[coordinate.first][coordinate.second +
             ↪   yDirection], baselinePressure, unitNormal, fluidVector) *
             ↪   stepSizes.y;

            // Cell 3: diagonal
            unitNormal = DoubleReal(xDirection / DIAGONAL_CELL_DISTANCE, yDirection
             ↪   / DIAGONAL_CELL_DISTANCE);
            REAL stepSize = (stepSizes.x + stepSizes.y) / 2;
            totalPresureDrag += PressureIntegrand(pressure[coordinate.first +
            ↪ xDirection][coordinate.second + yDirection], baselinePressure,
            ↪ unitNormal, fluidVector) * stepSize;
        }
        else if (numEdges == 1) { // Edge cell - compute the pressure integrand for
        ↪ the fluid cell next to it
            int i = coordinate.first + eastBit - westBit;
            int j = coordinate.second + northBit - southBit;
            DoubleReal unitNormal = DoubleReal((REAL)(eastBit - westBit),
            ↪ (REAL)(northBit - southBit));
            REAL stepSize;
            if ((eastBit | westBit) == 1) {
                stepSize = stepSizes.x;
            }
            else {
                stepSize = stepSizes.y;
            }

            totalPresureDrag += PressureIntegrand(pressure[i][j], baselinePressure,
            ↪ unitNormal, fluidVector) * stepSize;
        }

    }

    return totalPresureDrag;
}

REAL ComputeObstacleDrag(DoubleField velocities, REAL** pressure, BYTE** flags,
↪ std::pair<int, int>* coordinates, int coordinatesLength, int iMax, int jMax,
↪ DoubleReal stepSizes, REAL viscosity)
{
    DoubleReal fluidVector = DoubleReal(-1, 0);

    REAL viscousDrag = ComputeViscousDrag(velocities, flags, coordinates,
    ↪ coordinatesLength, iMax, jMax, stepSizes, fluidVector, viscosity) *
    ↪ VISCOSITY_CONVERSION;
    REAL pressureDrag = ComputePressureDrag(pressure, flags, coordinates,
    ↪ coordinatesLength, iMax, jMax, stepSizes, fluidVector) *
    ↪ PRESSURE_CONVERSION;
    return viscousDrag + pressureDrag;
}

/// <summary>
/// Finds the area of the object projected onto the y axis by finding the highest
↪ and lowest y coordinates of the obstacle.
/// </summary>
/// <returns>The area projected onto the y axis.</returns>
REAL ComputeProjectionArea(std::pair<int, int>* coordinates, int coordinatesLength,
↪ REAL delY) {
    int lowestY = coordinates[0].second;
    int highestY = coordinates[0].second;
    for (int i = 0; i < coordinatesLength; i++) {
        int yCoord = coordinates[i].second;
        if (yCoord > highestY) {
            highestY = yCoord;
        }
        if (yCoord < lowestY) {
            lowestY = yCoord;
        }
    }

    return (highestY - lowestY) * delY;
}

REAL ComputeDragCoefficient(DoubleField velocities, REAL** pressure, BYTE** flags,
↪   std::pair<int, int>* coordinates, int coordinatesLength, int iMax, int jMax,
↪   DoubleReal stepSizes, REAL viscosity, REAL density, REAL inflowVelocity)
{
    // Normal operation:
    REAL dragForce = ComputeObstacleDrag(velocities, pressure, flags, coordinates,
    ↪   coordinatesLength, iMax, jMax, stepSizes, viscosity);
    REAL projectedArea = ComputeProjectionArea(coordinates, coordinatesLength,
    ↪   stepSizes.y);
    return (2 * dragForce) / (density * inflowVelocity * inflowVelocity *
    ↪   projectedArea);
}
```

Drag.h

```
#ifndef DRAG_H
#define DRAG_H

#include "pch.h"

/// <summary>
/// Computes the drag on the obstacle in the simulation domain.
/// </summary>
/// <returns>The overall drag on the object.</returns>
REAL ComputeObstacleDrag(DoubleField velocities, REAL** pressure, BYTE** flags,
↪   std::pair<int, int>* coordinates, int coordinatesLength, int iMax, int jMax,
↪   DoubleReal stepSizes, REAL viscosity);

/// <summary>
/// Computes the drag coefficient for the obstacle in the simulation domain.
/// </summary>
/// <returns>The drag coefficient for the object.</returns>
REAL ComputeDragCoefficient(DoubleField velocities, REAL** pressure, BYTE** flags,
↪   std::pair<int, int>* coordinates, int coordinatesLength, int iMax, int jMax,
↪   DoubleReal stepSizes, REAL viscosity, REAL density, REAL inflowVelocity);
#endif // !DRAG_H
```

Constraint.cs

```
namespace FileMakerBackend
{
    public enum Inequality
    {
        LessThan,
        LessThanOrEqual,
        GreaterThan,
        GreaterThanOrEqual
    }

    /// <summary>
    /// Represents a constraint on x and y in the form f(x) + g(y) ? k, where f and
    ↪  g are functions, k is a constant, and ? is an inequality symbol.
    /// </summary>
    public class Constraint
    {
        protected readonly Func<float, float, float> function;
        protected readonly Inequality inequality;
        protected readonly float constant;

        protected bool InequalityCompare(float left)
        {
            return inequality switch
            {
                Inequality.LessThan => left < constant,
                Inequality.LessThanOrEqual => left <= constant,
                Inequality.GreaterThan => left > constant,
                Inequality.GreaterThanOrEqual => left >= constant,
                _ => false
            };
        }

        public virtual bool Evaluate(float x, float y)
        {
            float leftHandSide = function(x, y);
            return InequalityCompare(leftHandSide);
        }

        public Constraint(Func<float, float, float> function, Inequality inequality,
        ↪  float constant)
        {
            this.function = function;
            this.inequality = inequality;
            this.constant = constant;
        }
    }
}
```

FileMaker.cs

```
namespace FileMakerBackend
{
```

```
public static class FileMaker
{
    private static bool[] CreateObstacleArray(int xLength, int yLength,
    ↪   Predicate<(int, int)> ObstacleDefiningFunction)
    {
        bool[] obstacleArray = new bool[xLength * yLength];
        for (int i = 0; i < xLength; i++)
        {
            for (int j = 0; j < yLength; j++)
            {
                obstacleArray[i * yLength + j] = !ObstacleDefiningFunction((i,
                ↪   j)); // 1 means fluid cell, 0 means obstacle cell (opposite
                ↪   to defining function).
            }
        }
        return obstacleArray;
    }

    private static bool[] CreateObstacleArray(int xLength, int yLength,
    ↪   Constraint[] obstacleDefiningConstraints)
    {
        bool[] obstacleArray = new bool[xLength * yLength];
        for (int i = 0; i < xLength; i++)
        {
            for (int j = 0; j < yLength; j++)
            {
                bool cellIsObstacle = true;
                foreach (Constraint constraint in obstacleDefiningConstraints)
                {
                    cellIsObstacle &= constraint.Evaluate(i, j);
                }
                obstacleArray[i * yLength + j] = !cellIsObstacle; // 1 means
                ↪   fluid cell, 0 means obstacle cell (opposite to defining
                ↪   function).
            }
        }
        return obstacleArray;
    }


    private static void WriteFile(string filePath, bool[] obstacleArray, int
    ↪   xLength, int yLength)
    {
        byte[] buffer = new byte[obstacleArray.Length / 8 +
        ↪   (obstacleArray.Length % 8 == 0 ? 0 : 1)]; // Divide the length by 8
        ↪   and add one if the length does not divide evenly. Also add 1 byte
        ↪   for FLDEND

        int index = 0;
        for (int i = 0; i < obstacleArray.Length; i++)
        {
            buffer[index] |= (byte)((obstacleArray[i] ? 1 : 0) << i % 8); //
            ↪   Convert the bool to 1 or 0, shift it left the relevant amount of
            ↪   times and OR it with the current value in the buffer
```

```
                if (i % 8 == 7) // Add one to the index if the byte is full
                {
                    index++;
                }
            }

            using Stream stream = File.OpenWrite(filePath);
            using BinaryWriter writer = new BinaryWriter(stream);

            writer.Write(xLength - 2); // Subtract 2 for iMax and jMax.
            writer.Write(yLength - 2);
            writer.Write(buffer);
        }

        public static void CreateFile(string filePath, int xLength, int yLength,
        ↪   Predicate<(int, int)> ObstacleDefiningFunction)
        {
            bool[] obstacleArray = CreateObstacleArray(xLength, yLength,
            ↪   ObstacleDefiningFunction);
            WriteFile(filePath, obstacleArray, xLength, yLength);
        }

        public static void CreateFile(string filePath, int xLength, int yLength,
        ↪   Constraint[] obstacleDefiningConstraints)
        {
            bool[] obstacleArray = CreateObstacleArray(xLength, yLength,
            ↪   obstacleDefiningConstraints);
            WriteFile(filePath, obstacleArray, xLength, yLength);
        }
    }
}
```

Program.cs

```
namespace FileMakerBackend
{
    public class Program
    {
        private const int iMax = 200;
        private const int jMax = 100;
        private const string filePath = "square.simobst";

        private static bool ObstacleDefiningFunction((int, int) pointTuple)
        {
            // 1 is an obstacle cell, 0 is a fluid cell.
            (int x, int y) = pointTuple;
            return x > iMax * 0.45 && x < iMax * 0.55 && y > jMax * 0.45 && y < jMax
            ↪   * 0.55;
        }

        Constraint[] GetConstraints()
        {
            List<Constraint> constraints = new List<Constraint>();
            string userInput = "";
            while (userInput != "q")
            {
                Console.WriteLine("Type a constraint in the format f(x) + g(y) ? k,
                ↪ where f and g are functions, k is a constant, and ? is an
                ↪ inequality symbol.\nTo stop entering constraints, type q and
                ↪ press enter.");
            }
            return constraints.ToArray();
        }

        static void Main(string[] args)
        {
            FileMaker.CreateFile(filePath, iMax + 2, jMax + 2,
            ↪ ObstacleDefiningFunction);
        }
    }
}
```

ConstraintParser.cs

```
using FileMakerBackend;

namespace FileMakerCLI
{
    public static class ConstraintParser
    {
        public static readonly char[] Operators = ['^', '/', '*', '+', '-']; // In
        ↪ order of precendence
        private static readonly Dictionary<char, (int, bool)> operatorData = new
        ↪ Dictionary<char, (int, bool)>() // lowest int means highest precendence,
        ↪ true means right associativity, false means left associativity
        {
            {'^', (1, true) },
            {'/', (2, false) },
            {'*', (2, false) },
            {'-', (3, false) },
            {'+', (3, false) }
        };
        public static readonly string[] MathsFunctions = ["cos", "sin", "tan"];
        public static readonly string[] OperatorsAndFunctions = Operators.Select(x
        ↪ => x.ToString()).Concat(MathsFunctions).ToArray();
        private static readonly int functionLength = 3;

        private static string[] Tokenise(string input)
        {
            input = input.ToLower();
            List<string> tokens = new List<string>();
            int stringPos = 0;
            //int tokenPos = 0;
            char lastChar = '\0';
            char currentChar;
            while (stringPos < input.Length)
            {
```

```
currentChar = input[stringPos];
if (currentChar == ' ')
{
    stringPos++;
    continue; // Skip spaces
}

if (char.IsDigit(currentChar))
{
    if (char.IsDigit(lastChar) || lastChar == '.') // If the last
     ↪  char was a digit, use same token.
    {
        tokens[^1] += currentChar.ToString();
    }
    else // If not, start a new token.
    {
        tokens.Add(currentChar.ToString());
    }
}
else if (currentChar == '.')
{
    if (char.IsDigit(lastChar)) // Most common use case: decimal
     ↪  point after a number
    {
        tokens[^1] += currentChar.ToString();
    }
    else // Less common use case: decimal point implying a zero
    {
        tokens.Add("0.");
    }
}
else if (currentChar == 'x' || currentChar == 'y') // variables
{
    if (!Operators.Contains(lastChar) && lastChar != '(' && lastChar
     ↪  != '\0') // last char was not an operator, open bracket or
     ↪  start of string.
    {
        tokens.Add("*"); // Add a multiplication sign between the 2
         ↪  tokens if it was not an operator before.
    }
    tokens.Add(currentChar.ToString());
}
else if (Operators.Contains(currentChar) || currentChar == '(' ||
 ↪  currentChar == ')') // Operators
{
    tokens.Add(currentChar.ToString());
}
else if (input.Length - stringPos > functionLength &&
 ↪  MathsFunctions.Contains(input[stringPos..(stringPos +
 ↪  functionLength)])) // Check there is enough characters left and
 ↪  then see if the next characters are a function
{
```

```
                if (!Operators.Contains(lastChar) && lastChar != '(' && lastChar
                ↪  != '\0') // last char was not an operator, open bracket or
                ↪  start of string.
                {
                    tokens.Add("*"); // Add a multiplication sign between the 2
                    ↪  tokens if it was not an operator before.
                }
                tokens.Add(input[stringPos..(stringPos + functionLength)]);
                stringPos += functionLength - 1;
            }
            else
            {
                throw new FormatException("Input was not in the correct
                ↪  format.");
            }
            stringPos++;
            lastChar = currentChar;
        }
        return tokens.ToArray();
    }

    public static string[] ConvertToRPN(string infix)
    {
        string[] tokens = Tokenise(infix);
        List<string> output = new List<string>();
        ResizableStack<string> operatorStack = new ResizableStack<string>();
        int tokenPos = 0;
        while (tokenPos < tokens.Length)
        {
            string currentToken = tokens[tokenPos];
            if (float.TryParse(currentToken, out _)) // Token is a number
            {
                output.Add(currentToken);
            }
            else if (currentToken.Length == functionLength) // Current token is
            ↪  a function
            {
                operatorStack.Push(currentToken);
            }
            else
            {
                char symbol = currentToken[0]; // All other tokens have only 1
                ↪  character.
                if (symbol == 'x' || symbol == 'y')
                {
                    output.Add(currentToken);
                }
                else if (Operators.Contains(symbol))
                {
                    if (operatorStack.IsEmpty)
                    {
                        operatorStack.Push(currentToken);
                    }
                    else
                    {
                        char topOfStack = operatorStack.Peek()[0];
                        while (topOfStack != '(' &&
                            ↪   (operatorData[topOfStack].Item1 <
                            ↪   operatorData[symbol].Item1 ||
                            ↪   (operatorData[topOfStack].Item1 ==
                            ↪   operatorData[symbol].Item1 &&
                            ↪   !operatorData[symbol].Item2)))
                        {
                            output.Add(operatorStack.Pop());
                            if (operatorStack.IsEmpty) break;
                            topOfStack = operatorStack.Peek()[0];
                        }
                        operatorStack.Push(currentToken);
                    }
                }
                else if (symbol == '(')
                {
                    operatorStack.Push(currentToken);
                }
                else if (symbol == ')')
                {
                    string topOfStack;
                    do
                    {
                        if (operatorStack.IsEmpty)
                        {
                            throw new FormatException("Input was not a valid
                             ↪   infix expression");
                        }
                        topOfStack = operatorStack.Pop();
                        output.Add(topOfStack);
                    }
                    while (topOfStack != "(");
                    output.RemoveAt(output.Count - 1); // Remove the last
                     ↪   element (a "(").
                    if (!operatorStack.IsEmpty && operatorStack.Peek().Length ==
                     ↪   functionLength) // Function at the top of the stack
                    {
                        output.Add(operatorStack.Pop());
                    }
                }
            }
            tokenPos++;
        }
        if (!operatorStack.IsEmpty)
        {
            do // Pop the rest of the stack onto the output list
            {
                string topOfStack = operatorStack.Pop();
                if (topOfStack == "(" || topOfStack == ")")
                {
```

```
                        throw new FormatException("Input was not a valid infix
                        ↪   expression.");
                    }
                    output.Add(topOfStack);
                } while (!operatorStack.IsEmpty);
            }
            return output.ToArray();
        }

        public static RPNConstraint Parse(string stringConstraint)
        {
            string leftHandSide;
            Inequality inequality;
            float constant;
            if (stringConstraint.Contains("<="))
            {
                string[] equationSides = stringConstraint.Split("<=");
                leftHandSide = equationSides[0];
                inequality = Inequality.LessThanOrEqual;
                constant = float.Parse(equationSides[1]);
            }
            else if (stringConstraint.Contains('<'))
            {
                string[] equationSides = stringConstraint.Split('<');
                leftHandSide = equationSides[0];
                inequality = Inequality.LessThan;
                constant = float.Parse(equationSides[1]);
            }
            else if (stringConstraint.Contains(">="))
            {
                string[] equationSides = stringConstraint.Split(">=");
                leftHandSide = equationSides[0];
                inequality = Inequality.GreaterThanOrEqual;
                constant = float.Parse(equationSides[1]);
            }
            else if (stringConstraint.Contains('>'))
            {
                string[] equationSides = stringConstraint.Split('>');
                leftHandSide = equationSides[0];
                inequality = Inequality.GreaterThan;
                constant = float.Parse(equationSides[1]);
            }
            else
            {
                throw new FormatException("Constraint was incorrectly formatted.");
            }
            string[] postFix = ConvertToRPN(leftHandSide);
            return new RPNConstraint(postFix, inequality, constant);
        }
    }
}
```

Program.cs

```
using FileMakerBackend;

namespace FileMakerCLI
{
    internal class Program
    {
        static T ConstrainedInput<T>(string prompt, Predicate<string> requirements,
        ↪  Func<string, T> conversionFunction)
        {
            Console.WriteLine(prompt);
            string? userInput = Console.ReadLine();
            while (userInput is null || !requirements(userInput))
            {
                Console.WriteLine("Input was not valid.");
                Console.WriteLine(prompt);
                userInput = Console.ReadLine();
            }
            return conversionFunction(userInput);
        }

        static int IntInput(string prompt, bool requirePositive = false)
        {
            Predicate<string> requirements;
            if (requirePositive)
            {
                requirements = input => int.TryParse(input, out int value) && value
                ↪  > 0;
            }
            else
            {
                requirements = input => int.TryParse(input, out _);
            }
            return ConstrainedInput(prompt, requirements, int.Parse);
        }

        static string NonNullInput(string prompt)
        {
            return ConstrainedInput(prompt, input => input is not null, input =>
            ↪  input);
        }

        static Constraint[] GetConstraints()
        {
            List<Constraint> constraints = new List<Constraint>();
            Console.WriteLine("Type a constraint in the format f(x) + g(y) ? k,
            ↪  where f and g are functions, k is a constant, and ? is an inequality
            ↪  symbol.\nFor example, you could enter 2x^2+3y>=5.\nTo stop entering
            ↪  constraints, type q and press enter.");
            string? userInput = Console.ReadLine();
            while (userInput != "q" && userInput is not null)
            {
                try
                {
                    constraints.Add(ConstraintParser.Parse(userInput));
                } catch (FormatException)
                {
                    Console.WriteLine("Constraint was incorrectly formatted and has
                     ↪  not been added.");
                }
                Console.WriteLine("Type a constraint in the format f(x) + g(y) ? k,
                ↪  where f and g are functions, k is a constant, and ? is an
                ↪  inequality symbol.\nTo stop entering constraints, type q and
                ↪  press enter.");
                userInput = Console.ReadLine();
            }
            return constraints.ToArray();
        }

        static void RunProgram(string[] args)
        {
            int xLength = IntInput("Enter number of cells in x direction: ", true);
            int yLength = IntInput("Enter number of cells in y direction: ", true);
            string filePath = NonNullInput("Enter file path to output to.");
            Constraint[] constraints = GetConstraints();

            FileMaker.CreateFile(filePath, xLength, yLength, constraints);
        }

        static void Main(string[] args)
        {
            RunProgram(args);
        }
    }
}
```

ResizableStack.cs

```
namespace FileMakerCLI
{
    public class ResizableStack<T>
    {
        T[] array;

        protected int length;
        protected int pointerPosition = -1;

        public ResizableStack() //The default constructor for the method - useful
        ↪  for when the number of items to be pushed on the stack is not known
        ↪  (sets the initial size of the array to 1 item)
        {
            array = new T[1];
            length = 1;
        }
```

```
public ResizableStack(int length) //A constructor in case the minimum size
↪  the stack needs to be is known - therefore it would be more efficient
↪  (both time and space) to declare this at the start.
{
    array = new T[length];
    this.length = length;
}

public void Push(T value) //Add a new item to the stack
{
    if (IsFull)
    {
        length *= 2;
        Array.Resize(ref array, length); //If the array is full, resize the
        ↪  stack
    }
    pointerPosition = pointerPosition + 1;
    array[pointerPosition] = value;

}

public T Peek() //Peek the item from the top of the stack
{
    if (IsEmpty)
    {
        throw new InvalidOperationException("Stack was empty when Peek was
        ↪  called");
    }
    return array[pointerPosition];
}

public T Pop() //Take an item off the stack, returning it.
{
    if (IsEmpty)
    {
        throw new InvalidOperationException("Stack was empty when Pop was
        ↪  called");
    }
    pointerPosition--;
    if (pointerPosition < length / 4 && length > 1)
    {
        length /= 2;
        Array.Resize(ref array, length); //If less than 1/4 of the array is
        ↪  in use (calcalated by pointer being less than 1/4 of the length
        ↪  of the array)
    }
    return array[pointerPosition + 1];
}

public bool IsFull
{
    get { return pointerPosition + 1 == length; } //Full if pointer is 1
    ↪  less than length
```

```
        }

        public bool IsEmpty
        {
            get { return pointerPosition < 0; } //Empty if pointer is at -1 (below
            ↪  1st element)
        }
    }
}
```

RPNConstraint.cs

```
using FileMakerBackend;

namespace FileMakerCLI
{
    public class RPNConstraint : Constraint
    {
        protected readonly string[] postFixExpression;

        public float RPNProcess(float x, float y)
        {
            int tokenPos = 0;
            ResizableStack<float> evaluationStack = new ResizableStack<float>();
            while (tokenPos < postFixExpression.Length)
            {
                string token = postFixExpression[tokenPos];
                if (ConstraintParser.OperatorsAndFunctions.Contains(token))
                {
                    float operand1, operand2;
                    if (ConstraintParser.Operators.Contains(token[0]))
                    {
                        operand2 = evaluationStack.Pop();
                        operand1 = evaluationStack.Pop();
                    }
                    else // Function rather than operand
                    {
                        operand1 = evaluationStack.Pop();
                        operand2 = 0; // Not used.
                    }
                    evaluationStack.Push(token switch
                    {
                        "+" => operand1 + operand2,
                        "-" => operand1 - operand2,
                        "*" => operand1 * operand2,
                        "/" => operand1 / operand2,
                        "^" => (float)Math.Pow(operand1, operand2),
                        "sin" => (float)Math.Sin(operand1),
                        "cos" => (float)Math.Cos(operand1),
                        "tan" => (float)Math.Tan(operand1),
                        _ => 0
                    });
                }
                else
                {
                    switch (token)
                    {
                        case "x":
                            evaluationStack.Push(x);
                            break;
                        case "y":
                            evaluationStack.Push(y);
                            break;
                        default:
                            evaluationStack.Push(float.Parse(token));
                            break;
                    }
                }

                tokenPos++;
            }
            return evaluationStack.Pop();
        }

        public override bool Evaluate(float x, float y)
        {
            return InequalityCompare(RPNProcess(x, y));
        }

        public RPNConstraint(string[] postFixExpression, Inequality inequality,
        ↪   float constant) : base((float _, float _) => 0, inequality, constant)
        {
            this.postFixExpression = postFixExpression;
        }
    }
}
```

Boundary.cu

```
#include "Boundary.cuh"
#include <cmath>
#include <vector>

__global__ void SetFlags(PointerWithPitch<bool> obstacles, PointerWithPitch<BYTE>
↪   flags, int iMax, int jMax) {
    int rowNum = blockIdx.x * blockDim.x + threadIdx.x + 1;
    int colNum = blockIdx.y * blockDim.y + threadIdx.y + 1;
    if (rowNum > iMax) return;
    if (colNum > jMax) return;

    F_PITCHACCESS(flags.ptr, flags.pitch, rowNum, colNum) =
    ↪   ((BYTE)B_PITCHACCESS(obstacles.ptr, obstacles.pitch, rowNum, colNum) << 4) +
    ↪   ((BYTE)B_PITCHACCESS(obstacles.ptr, obstacles.pitch, rowNum, colNum + 1) <<
    ↪   3) + ((BYTE)B_PITCHACCESS(obstacles.ptr, obstacles.pitch, rowNum + 1,
    ↪   colNum) << 2) + ((BYTE)B_PITCHACCESS(obstacles.ptr, obstacles.pitch, rowNum,
    ↪   colNum - 1) << 1) + (BYTE)B_PITCHACCESS(obstacles.ptr, obstacles.pitch,
    ↪   rowNum - 1, colNum); // 5 bits in the format: self, north, east, south,
    ↪   west.
}

__global__ void TopBoundary(PointerWithPitch<REAL> hVel, PointerWithPitch<REAL>
↪  vVel, int iMax, int jMax)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x + 1;

    if (index > iMax) return;

    F_PITCHACCESS(hVel.ptr, hVel.pitch, index, jMax + 1) = F_PITCHACCESS(hVel.ptr,
    ↪  hVel.pitch, index, jMax); // Copy hVel from the cell below
    F_PITCHACCESS(vVel.ptr, vVel.pitch, index, jMax) = 0; // Set vVel along the top
    ↪  to 0
}

__global__ void BottomBoundary(PointerWithPitch<REAL> hVel, PointerWithPitch<REAL>
↪  vVel, int iMax, int jMax)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x + 1;

    if (index > iMax) return;

    F_PITCHACCESS(hVel.ptr, hVel.pitch, index, 0) = F_PITCHACCESS(hVel.ptr,
    ↪  hVel.pitch, index, 1); // Copy hVel from the cell above
    F_PITCHACCESS(vVel.ptr, vVel.pitch, index, 0) = 0; // Set vVel along the bottom
    ↪  to 0
}

__global__ void LeftBoundary(PointerWithPitch<REAL> hVel, PointerWithPitch<REAL>
↪  vVel, int iMax, int jMax, REAL inflowVelocity)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x + 1;

    if (index > jMax) return;

    F_PITCHACCESS(hVel.ptr, hVel.pitch, 0, index) = inflowVelocity; // Set hVel to
    ↪  inflow velocity on left boundary
    F_PITCHACCESS(vVel.ptr, vVel.pitch, 0, index) = 0; // Set vVel to 0
}

__global__ void RightBoundary(PointerWithPitch<REAL> hVel, PointerWithPitch<REAL>
↪  vVel, int iMax, int jMax)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x + 1;

    if (index > jMax) return;

    F_PITCHACCESS(hVel.ptr, hVel.pitch, iMax, index) = F_PITCHACCESS(hVel.ptr,
    ↪  hVel.pitch, iMax - 1, index); // Copy the velocity values from the previous
    ↪  cell (mass flows out at the boundary)
    F_PITCHACCESS(vVel.ptr, vVel.pitch, iMax + 1, index) = F_PITCHACCESS(vVel.ptr,
    ↪  vVel.pitch, iMax, index);
}
```

```
__global__ void ObstacleBoundary(PointerWithPitch<REAL> hVel, PointerWithPitch<REAL>
↪  vVel, PointerWithPitch<BYTE> flags, uint2* coordinates, int coordinatesLength,
↪  REAL chi) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index >= coordinatesLength) return;


    uint2 coordinate = coordinates[index];

    BYTE flag = B_PITCHACCESS(flags.ptr, flags.pitch, coordinate.x, coordinate.y);
    int northBit = (flag & NORTH) >> NORTHSHIFT;
    int eastBit  = (flag & EAST)  >> EASTSHIFT;
    int southBit = (flag & SOUTH) >> SOUTHSHIFT;
    int westBit  = (flag & WEST) >> WESTSHIFT;

    REAL velocityModifier = 2 * chi - 1; // This converts chi from chi in [0,1] to
    ↪   in [-1,1]

    F_PITCHACCESS(hVel.ptr, hVel.pitch, coordinate.x, coordinate.y) = (1 - eastBit)
    ↪   // If the cell is an eastern boudary, hVel is 0
        * (northBit * velocityModifier * F_PITCHACCESS(hVel.ptr, hVel.pitch,
        ↪   coordinate.x, coordinate.y + 1) // For northern boundaries, use the
        ↪   horizontal velocity above...
            + southBit * velocityModifier * F_PITCHACCESS(hVel.ptr, hVel.pitch,
            ↪   coordinate.x, coordinate.y - 1)); // ...and for southern boundaries,
            ↪   use the horizontal velocity below.

    F_PITCHACCESS(vVel.ptr, vVel.pitch, coordinate.x, coordinate.y) = (1 - northBit)
    ↪   // If the cell is a northern boundary, vVel is 0
        * (eastBit * velocityModifier * F_PITCHACCESS(vVel.ptr, vVel.pitch,
        ↪   coordinate.x + 1, coordinate.y) // For eastern boundaries, use the
        ↪   vertical velocity to the right...
            + westBit * velocityModifier * F_PITCHACCESS(vVel.ptr, vVel.pitch,
            ↪   coordinate.x - 1, coordinate.y)); // ...and for western boundaries,
            ↪   use the vertical velocity to the left.

    // The following lines are unavoidable branches.
    if (southBit != 0) { // If south bit is set,...
        F_PITCHACCESS(vVel.ptr, vVel.pitch, coordinate.x, coordinate.y - 1) = 0; //
        ↪   ...then set the velocity coming into the boundary to 0.
    }

    if (westBit != 0) { // If west bit is set,...
        F_PITCHACCESS(hVel.ptr, hVel.pitch, coordinate.x - 1, coordinate.y) = 0; //
        ↪   ...then set the velocity coming into the boundary to 0.
    }
}

cudaError_t SetBoundaryConditions(cudaStream_t* streams, int threadsPerBlock,
↪  PointerWithPitch<REAL> hVel, PointerWithPitch<REAL> vVel, PointerWithPitch<BYTE>
↪  flags, uint2* coordinates, int coordinatesLength, int iMax, int jMax, REAL
↪  inflowVelocity, REAL chi) {
    int numBlocksTopBottom = INT_DIVIDE_ROUND_UP(iMax, threadsPerBlock);
    int numBlocksLeftRight = INT_DIVIDE_ROUND_UP(jMax, threadsPerBlock);
    int numBlocksObstacle = INT_DIVIDE_ROUND_UP(coordinatesLength, threadsPerBlock);

    uint2* testingCoordinates = new uint2[coordinatesLength];
    cudaMemcpy(testingCoordinates, coordinates, coordinatesLength * sizeof(uint2),
    ↪  cudaMemcpyDeviceToHost);

    TopBoundary KERNEL_ARGS(numBlocksTopBottom, threadsPerBlock, 0, streams[0])
    ↪  (hVel, vVel, iMax, jMax);
    BottomBoundary KERNEL_ARGS(numBlocksTopBottom, threadsPerBlock, 0, streams[1])
    ↪  (hVel, vVel, iMax, jMax);
    LeftBoundary KERNEL_ARGS(numBlocksLeftRight, threadsPerBlock, 0, streams[2])
    ↪  (hVel, vVel, iMax, jMax, inflowVelocity);
    RightBoundary KERNEL_ARGS(numBlocksLeftRight, threadsPerBlock, 0, streams[3])
    ↪  (hVel, vVel, iMax, jMax);

    cudaError_t retVal = cudaStreamSynchronize(streams[0]);
    if (retVal != cudaSuccess) return retVal;
    ObstacleBoundary KERNEL_ARGS(numBlocksObstacle, threadsPerBlock, 0, streams[0])
    ↪  (hVel, vVel, flags, coordinates, coordinatesLength, chi);

    return cudaDeviceSynchronize();
}

void FindBoundaryCells(BYTE** flags, uint2*& coordinates, int& coordinatesLength,
↪  int iMax, int jMax) {
    std::vector<uint2> coordinatesVec;
    for (int i = 1; i <= iMax; i++) {
        for (int j = 1; j <= jMax; j++) {
            if (flags[i][j] >= 0b00000001 && flags[i][j] <= 0b00001111) { // This
            ↪  defines boundary cells - all cells without the self bit set except
            ↪  when no bits are set.
                uint2 coordinate = uint2();
                coordinate.x = i;
                coordinate.y = j;
                coordinatesVec.push_back(coordinate);
            }
        }
    }
    coordinates = new uint2[coordinatesVec.size()]; // Allocate mem for array into
    ↪  already defined pointer
    std::copy(coordinatesVec.begin(), coordinatesVec.end(), coordinates); // Copy
    ↪  the elements from the vector to the array
    coordinatesLength = (int)coordinatesVec.size();
}
```

Boundary.cuh

```
#ifndef BOUNDARY_CUH
#define BOUNDARY_CUH

#include "Definitions.cuh"
```

```
/// <summary>
/// Sets the flags for each cell based on the value of surrounding cells. Requires
↪ iMax x jMax threads.
/// </summary>
/// <param name="obstacles">A boolean array indicating whether each cell is obstacle
↪ or fluid.</param>
/// <param name="flags">A BYTE array to hold the flags.</param>
__global__ void SetFlags(PointerWithPitch<bool> obstacles, PointerWithPitch<BYTE>
↪ flags, int iMax, int jMax);

/// <summary>
/// Applies top boundary conditions. Requires iMax threads.
/// </summary>
/// <param name="hVel">Pointer with pitch for horizontal velocity.</param>
/// <param name="vVel">Pointer with pitch for vertical velocity.</param>
/// <param name="jMax">The number of fluid cells in the y direction.</param>
__global__ void TopBoundary(PointerWithPitch<REAL> hVel, PointerWithPitch<REAL>
↪ vVel, int iMax, int jMax);

/// <summary>
/// Applies bottom boundary conditions. Requires iMax threads.
/// </summary>
/// <param name="hVel">Pointer with pitch for horizontal velocity.</param>
/// <param name="vVel">Pointer with pitch for vertical velocity.</param>
__global__ void BottomBoundary(PointerWithPitch<REAL> hVel, PointerWithPitch<REAL>
↪ vVel, int iMax, int jMax);

/// <summary>
/// Applies left boundary conditions. Requires jMax threads.
/// </summary>
/// <param name="hVel">Pointer with pitch for horizontal velocity.</param>
/// <param name="vVel">Pointer with pitch for vertical velocity.</param>
/// <param name="inflowVelocity">The velocity of fluid on the left boundary</param>
__global__ void LeftBoundary(PointerWithPitch<REAL> hVel, PointerWithPitch<REAL>
↪ vVel, int iMax, int jMax, REAL inflowVelocity);

/// <summary>
/// Applies right boundary conditions. Requires jMax threads.
/// </summary>
/// <param name="hVel">Pointer with pitch for horizontal velocity.</param>
/// <param name="vVel">Pointer with pitch for vertical velocity.</param>
/// <param name="iMax">The number of fluid cells in the x direction.</param>
__global__ void RightBoundary(PointerWithPitch<REAL> hVel, PointerWithPitch<REAL>
↪ vVel, int iMax, int jMax);

/// <summary>
/// Applies boundary conditions on obstacles. Requires <paramref
↪ name="coordinatesLength" /> threads.
/// </summary>
__global__ void ObstacleBoundary(PointerWithPitch<REAL> hVel, PointerWithPitch<REAL>
↪ vVel, PointerWithPitch<BYTE> flags, uint2* coordinates, int coordinatesLength,
↪ REAL chi);
```

```
/// <summary>
/// Sets boundary conditions. Handles kernel launching internally. Requires 4
↪  streams.
/// </summary>
cudaError_t SetBoundaryConditions(cudaStream_t* streams, int threadsPerBlock,
↪  PointerWithPitch<REAL> hVel, PointerWithPitch<REAL> vVel, PointerWithPitch<BYTE>
↪  flags, uint2* coordinates, int coordinatesLength, int iMax, int jMax, REAL
↪  inflowVelocity, REAL chi);

void FindBoundaryCells(BYTE** flags, uint2*& coordinates, int& coordinatesLength,
↪  int iMax, int jMax);

#endif // !BOUNDARY_CUH
```

Computation.cu

```
#include "Computation.cuh"
#include "DiscreteDerivatives.cuh"
#include "ReductionKernels.cuh"
#include <cmath>

/// <summary>
/// Performs the unparallelisable part of ComputeGamma on the GPU to avoid having to
↪  copy memory to the CPU. Requires 1 thread.
/// </summary>
__global__ void FinishComputeGamma(REAL* gamma, REAL* hVelMax, REAL* vVelMax, REAL*
↪  timestep, REAL delX, REAL delY) {
    REAL horizontalComponent = *hVelMax * (*timestep / delX);
    REAL verticalComponent = *vVelMax * (*timestep / delY);

    if (horizontalComponent > verticalComponent) {
        *gamma = horizontalComponent;
    }
    else {
        *gamma = verticalComponent;
    }
}

cudaError_t ComputeGamma(REAL* gamma, cudaStream_t* streams, int threadsPerBlock,
↪  PointerWithPitch<REAL> hVel, PointerWithPitch<REAL> vVel, int iMax, int jMax,
↪  REAL* timestep, REAL delX, REAL delY) {
    cudaError_t retVal;
    REAL* hVelMax;
    retVal = cudaMalloc(&hVelMax, sizeof(REAL));
    if (retVal != cudaSuccess) goto free;

    REAL* vVelMax;
    retVal = cudaMalloc(&vVelMax, sizeof(REAL));
    if (retVal != cudaSuccess) goto free;

    FieldMax(hVelMax, streams[0], hVel, iMax + 2, jMax + 2);

    retVal = cudaStreamSynchronize(streams[0]);
    if (retVal != cudaSuccess) goto free;
    FieldMax(vVelMax, streams[1], vVel, iMax + 2, jMax + 2);

    retVal = cudaStreamSynchronize(streams[1]);
    if (retVal != cudaSuccess) goto free;

    FinishComputeGamma KERNEL_ARGS(1, 1, 0, streams[0]) (gamma, hVelMax, vVelMax,
     ↪  timestep, delX, delY);

    free:
    cudaFree(hVelMax);
    cudaFree(vVelMax);
    return retVal;
}

/// <summary>
/// Performs the unparallelisable part of ComputeTimestep on the GPU to avoid having
↪  to copy memory to the CPU. Requires 1 thread.
/// </summary>
__global__ void FinishComputeTimestep(REAL* timestep, REAL* hVelMax, REAL* vVelMax,
↪  REAL delX, REAL delY, REAL reynoldsNo, REAL safetyFactor)
{
    REAL inverseSquareRestriction = (REAL)0.5 * reynoldsNo * (1 / square(delX) + 1 /
     ↪  square(delY));
    REAL xTravelRestriction = delX / *hVelMax;
    REAL yTravelRestriction = delY / *vVelMax;

    REAL smallestRestriction = inverseSquareRestriction; // Choose the smallest
     ↪  restriction
    if (xTravelRestriction < smallestRestriction) {
        smallestRestriction = xTravelRestriction;
    }
    if (yTravelRestriction < smallestRestriction) {
        smallestRestriction = yTravelRestriction;
    }
    *timestep = safetyFactor * smallestRestriction;
}

cudaError_t ComputeTimestep(REAL* timestep, cudaStream_t* streams,
↪  PointerWithPitch<REAL> hVel, PointerWithPitch<REAL> vVel, int iMax, int jMax,
↪  REAL delX, REAL delY, REAL reynoldsNo, REAL safetyFactor)
{
    cudaError_t retVal;
    REAL* hVelMax;
    retVal = cudaMalloc(&hVelMax, sizeof(REAL));
    if (retVal != cudaSuccess) goto free;

    REAL* vVelMax;
    retVal = cudaMalloc(&vVelMax, sizeof(REAL));
    if (retVal != cudaSuccess) goto free;

    FieldMax(hVelMax, streams[0], hVel, iMax + 2, jMax + 2);
    retVal = cudaStreamSynchronize(streams[0]);
    if (retVal != cudaSuccess) goto free;

    FieldMax(vVelMax, streams[1], vVel, iMax + 2, jMax + 2);

    retVal = cudaStreamSynchronize(streams[1]);
    if (retVal != cudaSuccess) goto free;

    FinishComputeTimestep KERNEL_ARGS(1, 1, 0, streams[0]) (timestep, hVelMax,
    ↪   vVelMax, delX, delY, reynoldsNo, safetyFactor);

free:
    cudaFree(hVelMax);
    cudaFree(vVelMax);
    return retVal;
}

/// <summary>
/// Computes F on the top and bottom of the simulation domain. Requires jMax
↪   threads.
/// </summary>
__global__ void ComputeFBoundary(PointerWithPitch<REAL> hVel, PointerWithPitch<REAL>
↪   F, int iMax, int jMax) {
    int colNum = blockIdx.x * blockDim.x + threadIdx.x;
    if (colNum > jMax) return;

    F_PITCHACCESS(F.ptr, F.pitch, 0, colNum) = F_PITCHACCESS(hVel.ptr, hVel.pitch,
    ↪   0, colNum);
    F_PITCHACCESS(F.ptr, F.pitch, iMax, colNum) = F_PITCHACCESS(hVel.ptr,
    ↪   hVel.pitch, iMax, colNum);
}

/// <summary>
/// Computes G on the left and right of the simulation domain. Requires iMax
↪   threads.
/// </summary>
__global__ void ComputeGBoundary(PointerWithPitch<REAL> vVel, PointerWithPitch<REAL>
↪   G, int iMax, int jMax) {
    int rowNum = blockIdx.x * blockDim.x + threadIdx.x;
    if (rowNum > iMax) return;

    F_PITCHACCESS(G.ptr, G.pitch, rowNum, 0) = F_PITCHACCESS(vVel.ptr, vVel.pitch,
    ↪   rowNum, 0);
    F_PITCHACCESS(G.ptr, G.pitch, rowNum, jMax) = F_PITCHACCESS(vVel.ptr,
    ↪   vVel.pitch, rowNum, jMax);
}

/// <summary>
/// Computes quantity F. Requires (iMax - 1) x (jMax) threads.
/// </summary>
__global__ void ComputeF(PointerWithPitch<REAL> hVel, PointerWithPitch<REAL> vVel,
↪   PointerWithPitch<REAL> F, PointerWithPitch<BYTE> flags, int iMax, int jMax,
↪   REAL* timestep, REAL delX, REAL delY, REAL xForce, REAL* gamma, REAL
↪   reynoldsNum) {
    int rowNum = blockIdx.x * blockDim.x + threadIdx.x + 1;
    int colNum = blockIdx.y * blockDim.y + threadIdx.y + 1;

    if (rowNum >= iMax) return;
    if (colNum > jMax) return;

    int selfBit = (B_PITCHACCESS(flags.ptr, flags.pitch, rowNum, colNum) & SELF) >>
    ↪   SELFSHIFT; // SELF bit of the cell's flag
    int eastBit = (B_PITCHACCESS(flags.ptr, flags.pitch, rowNum, colNum) & EAST) >>
    ↪   EASTSHIFT; // EAST bit of the cell's flag

    F_PITCHACCESS(F.ptr, F.pitch, rowNum, colNum) =
        F_PITCHACCESS(hVel.ptr, hVel.pitch, rowNum, colNum) * (selfBit | eastBit) //
        ↪   For boundary cells or fluid cells, add hVel
        + *timestep * (1 / reynoldsNum * (SecondPuPx(hVel, rowNum, colNum, delX) +
        ↪   SecondPuPy(hVel, rowNum, colNum, delY)) - PuSquaredPx(hVel, rowNum,
        ↪   colNum, delX, *gamma) - PuvPy(hVel, vVel, rowNum, colNum, delX, delY,
        ↪   *gamma) + xForce) * (selfBit & eastBit); // For fluid cells only,
        ↪   perform the computation. Obstacle cells without an eastern boundary are
        ↪   set to 0.
}

/// <summary>
/// Computes quantity G. Requires (iMax) x (jMax - 1) threads.
/// </summary>
__global__ void ComputeG(PointerWithPitch<REAL> hVel, PointerWithPitch<REAL> vVel,
↪   PointerWithPitch<REAL> G, PointerWithPitch<BYTE> flags, int iMax, int jMax,
↪   REAL* timestep, REAL delX, REAL delY, REAL yForce, REAL* gamma, REAL
↪   reynoldsNum) {
    int rowNum = blockIdx.x * blockDim.x + threadIdx.x + 1;
    int colNum = blockIdx.y * blockDim.y + threadIdx.y + 1;

    if (rowNum > iMax) return;
    if (colNum >= jMax) return;

    int selfBit = (B_PITCHACCESS(flags.ptr, flags.pitch, rowNum, colNum) & SELF) >>
    ↪   SELFSHIFT;    // SELF bit of the cell's flag
    int northBit = (B_PITCHACCESS(flags.ptr, flags.pitch, rowNum, colNum) & NORTH)
    ↪   >> NORTHSHIFT; // NORTH bit of the cell's flag

    F_PITCHACCESS(G.ptr, G.pitch, rowNum, colNum) =
        F_PITCHACCESS(vVel.ptr, vVel.pitch, rowNum, colNum) * (selfBit | northBit)
        ↪   // For boundary cells or fluid cells, add vVel
        + *timestep * (1 / reynoldsNum * (SecondPvPx(vVel, rowNum, colNum, delX) +
        ↪   SecondPvPy(vVel, rowNum, colNum, delY)) - PuvPx(hVel, vVel, rowNum,
        ↪   colNum, delX, delY, *gamma) - PvSquaredPy(vVel, rowNum, colNum, delY,
        ↪   *gamma) + yForce) * (selfBit & northBit); // For fluid cells only,
        ↪   perform the computation. Obstacle cells without a northern boundary are
        ↪   set to 0.
}
```

```
cudaError_t ComputeFG(cudaStream_t* streams, dim3 threadsPerBlock,
↪  PointerWithPitch<REAL> hVel, PointerWithPitch<REAL> vVel, PointerWithPitch<REAL>
↪  F, PointerWithPitch<REAL> G, PointerWithPitch<BYTE> flags, int iMax, int jMax,
↪  REAL* timestep, REAL delX, REAL delY, REAL xForce, REAL yForce, REAL* gamma,
↪  REAL reynoldsNum) {
    dim3 numBlocksF(INT_DIVIDE_ROUND_UP(iMax - 1, threadsPerBlock.x),
     ↪  INT_DIVIDE_ROUND_UP(jMax, threadsPerBlock.y));
    dim3 numBlocksG(INT_DIVIDE_ROUND_UP(iMax, threadsPerBlock.x),
     ↪  INT_DIVIDE_ROUND_UP(jMax - 1, threadsPerBlock.y));

    int threadsPerBlockFlat = threadsPerBlock.x * threadsPerBlock.y;
    int numBlocksIMax = INT_DIVIDE_ROUND_UP(iMax, threadsPerBlockFlat);
    int numBlocksJMax = INT_DIVIDE_ROUND_UP(jMax, threadsPerBlockFlat);

    ComputeF KERNEL_ARGS(numBlocksF, threadsPerBlock, 0, streams[0]) (hVel, vVel, F,
     ↪  flags, iMax, jMax, timestep, delX, delY, xForce, gamma, reynoldsNum); //
     ↪  Launch the kernels in separate streams, to be concurrently executed if the
     ↪  GPU is able to.
    ComputeG KERNEL_ARGS(numBlocksG, threadsPerBlock, 0, streams[1]) (hVel, vVel, G,
     ↪  flags, iMax, jMax, timestep, delX, delY, yForce, gamma, reynoldsNum);

    ComputeFBoundary KERNEL_ARGS(numBlocksJMax, threadsPerBlockFlat, 0, streams[2])
     ↪  (hVel, F, iMax, jMax);
    ComputeGBoundary KERNEL_ARGS(numBlocksIMax, threadsPerBlockFlat, 0, streams[3])
     ↪  (vVel, G, iMax, jMax);

    return cudaDeviceSynchronize();
}

__global__ void ComputeRHS(PointerWithPitch<REAL> F, PointerWithPitch<REAL> G,
↪  PointerWithPitch<REAL> RHS, PointerWithPitch<BYTE> flags, int iMax, int jMax,
↪  REAL* timestep, REAL delX, REAL delY) {
    int rowNum = blockIdx.x * blockDim.x + threadIdx.x + 1;
    int colNum = blockIdx.y * blockDim.y + threadIdx.y + 1;

    if (rowNum > iMax) return;
    if (colNum > jMax) return;

    F_PITCHACCESS(RHS.ptr, RHS.pitch, rowNum, colNum) =
        ((B_PITCHACCESS(flags.ptr, flags.pitch, rowNum, colNum) & SELF) >>
        ↪  SELFSHIFT) // Sets the entire expression to 0 if the cell is not fluid
        * (1 / *timestep) * (((F_PITCHACCESS(F.ptr, F.pitch, rowNum, colNum) -
        ↪  F_PITCHACCESS(F.ptr, F.pitch, rowNum - 1, colNum)) / delX) +
        ↪  ((F_PITCHACCESS(G.ptr, G.pitch, rowNum, colNum) - F_PITCHACCESS(G.ptr,
        ↪  G.pitch, rowNum, colNum - 1)) / delY));
}

/// <summary>
/// Computes horizontal velocity. Requires iMax x jMax threads, called by
↪  ComputeVelocities.
/// </summary>
__global__ void ComputeHVel(PointerWithPitch<REAL> hVel, PointerWithPitch<REAL> F,
↪  PointerWithPitch<REAL> pressure, PointerWithPitch<BYTE> flags, int iMax, int
↪  jMax, REAL* timestep, REAL delX)
```

```
{
    int rowNum = blockIdx.x * blockDim.x + threadIdx.x + 1;
    int colNum = blockIdx.y * blockDim.y + threadIdx.y + 1;

    if (rowNum > iMax) return; // Bounds checking
    if (colNum > jMax) return;

    F_PITCHACCESS(hVel.ptr, hVel.pitch, rowNum, colNum) =
        ((B_PITCHACCESS(flags.ptr, flags.pitch, rowNum, colNum) & SELF) >>
        ↪  SELFSHIFT) // Equal to 0 if the cell is not a fluid cell
        * ((B_PITCHACCESS(flags.ptr, flags.pitch, rowNum, colNum) & EAST) >>
        ↪  EASTSHIFT) // Equal to 0 if the cell has an obstacle cell next to it in
        ↪  +ve x direction (east)
        * (F_PITCHACCESS(F.ptr, F.pitch, rowNum, colNum) - (*timestep / delX) *
        ↪  (F_PITCHACCESS(pressure.ptr, pressure.pitch, rowNum + 1, colNum) -
        ↪  F_PITCHACCESS(pressure.ptr, pressure.pitch, rowNum, colNum)));
}

/// <summary>
/// Computes vertical velocity. Requires iMax x jMax threads, called by
↪  ComputeVelocities.
/// </summary>
__global__ void ComputeVVel(PointerWithPitch<REAL> vVel, PointerWithPitch<REAL> G,
↪  PointerWithPitch<REAL> pressure, PointerWithPitch<BYTE> flags, int iMax, int
↪  jMax, REAL* timestep, REAL delY)
{
    int rowNum = blockIdx.x * blockDim.x + threadIdx.x + 1;
    int colNum = blockIdx.y * blockDim.y + threadIdx.y + 1;

    if (rowNum > iMax) return; // Bounds checking
    if (colNum > jMax) return;

    F_PITCHACCESS(vVel.ptr, vVel.pitch, rowNum, colNum) =
        ((B_PITCHACCESS(flags.ptr, flags.pitch, rowNum, colNum) & SELF) >>
        ↪  SELFSHIFT) // Equal to 0 if the cell is not a fluid cell
        * ((B_PITCHACCESS(flags.ptr, flags.pitch, rowNum, colNum) & NORTH) >>
        ↪  NORTHSHIFT) // Equal to 0 if the cell has an obstacle cell next to it in
        ↪  +ve y direction (north)
        * (F_PITCHACCESS(G.ptr, G.pitch, rowNum, colNum) - (*timestep / delY) *
        ↪  (F_PITCHACCESS(pressure.ptr, pressure.pitch, rowNum, colNum + 1) -
        ↪  F_PITCHACCESS(pressure.ptr, pressure.pitch, rowNum, colNum)));
}

cudaError_t ComputeVelocities(cudaStream_t* streams, dim3 threadsPerBlock,
↪  PointerWithPitch<REAL> hVel, PointerWithPitch<REAL> vVel, PointerWithPitch<REAL>
↪  F, PointerWithPitch<REAL> G, PointerWithPitch<REAL> pressure,
↪  PointerWithPitch<BYTE> flags, int iMax, int jMax, REAL* timestep, REAL delX,
↪  REAL delY)
{
    dim3 numBlocks(INT_DIVIDE_ROUND_UP(iMax, threadsPerBlock.x),
    ↪  INT_DIVIDE_ROUND_UP(jMax, threadsPerBlock.y));
    ComputeHVel KERNEL_ARGS(numBlocks, threadsPerBlock, 0, streams[0]) (hVel, F,
    ↪  pressure, flags, iMax, jMax, timestep, delX); // Launch the kernels in
    ↪  separate streams, to be concurrently executed if the GPU is able to.
    ComputeVVel KERNEL_ARGS(numBlocks, threadsPerBlock, 0, streams[1]) (vVel, G,
    ↪  pressure, flags, iMax, jMax, timestep, delY);
    return cudaDeviceSynchronize();
}

__global__ void ComputeStream(PointerWithPitch<REAL> hVel, PointerWithPitch<REAL>
↪  streamFunction, int iMax, int jMax, REAL delY)
{
    int rowNum = blockIdx.x * blockDim.x + threadIdx.x;
    if (rowNum > iMax) return;

    F_PITCHACCESS(streamFunction.ptr, streamFunction.pitch, rowNum, 0) = 0; //
    ↪  Stream function boundary condition
    for (int colNum = 1; colNum <= jMax; colNum++) {
        F_PITCHACCESS(streamFunction.ptr, streamFunction.pitch, rowNum, colNum) =
        ↪  F_PITCHACCESS(streamFunction.ptr, streamFunction.pitch, rowNum, colNum -
        ↪  1) + F_PITCHACCESS(hVel.ptr, hVel.pitch, rowNum, colNum) * delY;
    }
}
```

Computation.cuh

```
#ifndef COMPUTATION_CUH
#define COMPUTATION_CUH

#include "Definitions.cuh"

/// <summary>
/// Computes gamma using a reduction kernel. Handles kernel launching internally.
↪  Requires 2 streams.
/// </summary>
/// <param name="gamma">A pointer to the location to output the calculated
↪  gamma.</param>
/// <param name="streams">A pointer to an array of at least 2 streams.</param>
/// <param name="threadsPerBlock">The maximum number of threads per thread
↪  block.</param>
cudaError_t ComputeGamma(REAL* gamma, cudaStream_t* streams, int threadsPerBlock,
↪  PointerWithPitch<REAL> hVel, PointerWithPitch<REAL> vVel, int iMax, int jMax,
↪  REAL* timestep, REAL delX, REAL delY);

cudaError_t ComputeTimestep(REAL* timestep, cudaStream_t* streams,
↪  PointerWithPitch<REAL> hVel, PointerWithPitch<REAL> vVel, int iMax, int jMax,
↪  REAL delX, REAL delY, REAL reynoldsNo, REAL safetyFactor);

/// <summary>
/// Computes F and G. Handles kernel launching internally. Requires 4 threads.
/// </summary>
cudaError_t ComputeFG(cudaStream_t* streams, dim3 threadsPerBlock,
↪  PointerWithPitch<REAL> hVel, PointerWithPitch<REAL> vVel, PointerWithPitch<REAL>
↪  F, PointerWithPitch<REAL> G, PointerWithPitch<BYTE> flags, int iMax, int jMax,
↪  REAL* timestep, REAL delX, REAL delY, REAL xForce, REAL yForce, REAL* gamma,
↪  REAL reynoldsNum);

/// <summary>
/// Computes pressure RHS. Requires iMax x jMax threads.
/// </summary>
__global__ void ComputeRHS(PointerWithPitch<REAL> F, PointerWithPitch<REAL> G,
↪  PointerWithPitch<REAL> RHS, PointerWithPitch<BYTE> flags, int iMax, int jMax,
↪  REAL* timestep, REAL delX, REAL delY);

/// <summary>
/// Computes both vertical and horizontal velocities. Handles kernel launching
↪  internally.
/// </summary>
cudaError_t ComputeVelocities(cudaStream_t* streams, dim3 threadsPerBlock,
↪  PointerWithPitch<REAL> hVel, PointerWithPitch<REAL> vVel, PointerWithPitch<REAL>
↪  F, PointerWithPitch<REAL> G, PointerWithPitch<REAL> pressure,
↪  PointerWithPitch<BYTE> flags, int iMax, int jMax, REAL* timestep, REAL delX,
↪  REAL delY);

/// <summary>
/// Computes stream function in the y direction. Requires (iMax + 1) threads.
/// </summary>
__global__ void ComputeStream(PointerWithPitch<REAL> hVel, PointerWithPitch<REAL>
↪  streamFunction, int iMax, int jMax, REAL delY);

#endif // !COMPUTATION_CUH
```

Definitions.cuh

```
#ifndef DEFINITIONS_CUH
#define DEFINITIONS_CUH

#include "Definitions.h"
#include "cuda.h"
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#define F_PITCHACCESS(basePtr, pitch, i, j) (*((REAL*)((char*)(basePtr) + (i) *
↪  (pitch)) + (j))) // Used for accessing a location in a pitched array (F for
↪  float, FP for float pointer, B for byte, BP for byte pointer.)
#define FP_PITCHACCESS(basePtr, pitch, i, j) ((REAL*)((char*)(basePtr) + (i) *
↪  (pitch)) + (j)) // Used for accessing a location in a pitched array (F for
↪  float, FP for float pointer, B for byte, BP for byte pointer.)
#define B_PITCHACCESS(basePtr, pitch, i, j) (*((basePtr) + (i) * (pitch) + (j))) //
↪  Used for accessing a location in a pitched array (F for float, FP for float
↪  pointer, B for byte, BP for byte pointer.)
#define BP_PITCHACCESS(basePtr, pitch, i, j) ((basePtr) + (i) * (pitch) + (j)) //
↪  Used for accessing a location in a pitched array (F for float, FP for float
↪  pointer, B for byte, BP for byte pointer.)

// Horrific macros to make intellisense stop complaining about the triple angle
↪  bracket syntax for kernel launches
#ifndef __INTELLISENSE__
#define KERNEL_ARGS(numBlocks, numThreads, sh_mem, stream) <<< numBlocks,
↪  numThreads, sh_mem, stream >>> // Launch a kernel with shared memory and stream
↪  specified.
#else
```

```
#define KERNEL_ARGS(numBlocks, numThreads, sh_mem, stream) // Launch a kernel with
↪  shared memory and stream specified.
#endif

#define INT_DIVIDE_ROUND_UP(numerator, denominator) (((numerator) + (denominator) -
↪  1) / (denominator))

#define make_REAL2 make_float2
typedef float2 REAL2;

template <typename T>
struct PointerWithPitch
{
    T* ptr;
    size_t pitch;
};

struct DragCoordinate
{
    uint2 coordinate;
    REAL2 unitNormal;
    REAL stepSize;
};

#endif // !DEFINITIONS_CUH
```

DiscreteDerivatives.cu

```
#include "DiscreteDerivatives.cuh"
#include <cmath>

/*
A note on terminology:
Below are functions to represent the calculations of different derivatives used in
↪  the Navier-Stokes equations. They have been discretised.
Average: the sum of 2 quantities, then divided by 2. Taking the mean of the 2
↪  quantities.
Difference: The same as above, but with subtraction.
Forward: applying an average or difference between the current cell (i,j) and the
↪  next cell along (i+1,j) or (i,j+1)
Backward: the same as above, but applied to the cell behind - (i-1,j) or (i,j-1).
Downshift: Any of the above with respect to the cell below the current one, (i,
↪  j-1).
Second derivative: the double application of a derivative.
Donor and non-donor: There are 2 different discretisation methods here, one of which
↪  is donor-cell discretisation. The 2 parts of each discretisation formula are
↪  named as such.
*/

__device__ REAL PuPx(PointerWithPitch<REAL> hVel, int i, int j, REAL delx) { //
↪  NOTE: P here is used to represent the partial operator, so PuPx should be read
↪  "partial u by partial x"
    return (F_PITCHACCESS(hVel.ptr, hVel.pitch, i, j) - F_PITCHACCESS(hVel.ptr,
    ↪  hVel.pitch, i - 1, j)) / delx;
```

```
}

__device__ REAL PvPy(PointerWithPitch<REAL> vVel, int i, int j, REAL dely) {
    return (F_PITCHACCESS(vVel.ptr, vVel.pitch, i, j) - F_PITCHACCESS(vVel.ptr,
     ↪  vVel.pitch, i, j - 1)) / dely;
}

__device__ REAL PuSquaredPx(PointerWithPitch<REAL> hVel, int i, int j, REAL delx,
↪  REAL gamma) {
    REAL forwardAverage = (F_PITCHACCESS(hVel.ptr, hVel.pitch, i, j) +
     ↪  F_PITCHACCESS(hVel.ptr, hVel.pitch, i + 1, j)) / 2;
    REAL backwardAverage = (F_PITCHACCESS(hVel.ptr, hVel.pitch, i - 1, j) +
     ↪  F_PITCHACCESS(hVel.ptr, hVel.pitch, i, j)) / 2;

    REAL forwardDifference = (F_PITCHACCESS(hVel.ptr, hVel.pitch, i, j) -
     ↪  F_PITCHACCESS(hVel.ptr, hVel.pitch, i + 1, j)) / 2;
    REAL backwardDifference = (F_PITCHACCESS(hVel.ptr, hVel.pitch, i - 1, j) -
     ↪  F_PITCHACCESS(hVel.ptr, hVel.pitch, i, j)) / 2;

    REAL nonDonorTerm = (1 / delx) * (square(forwardAverage) -
     ↪  square(backwardAverage));
    REAL donorTerm = (gamma / delx) * ((abs(forwardAverage) * forwardDifference) -
     ↪  (abs(backwardAverage) * backwardDifference));

    return nonDonorTerm + donorTerm;
}

__device__ REAL PvSquaredPy(PointerWithPitch<REAL> vVel, int i, int j, REAL dely,
↪  REAL gamma) {
    REAL forwardAverage = (F_PITCHACCESS(vVel.ptr, vVel.pitch, i, j) +
     ↪  F_PITCHACCESS(vVel.ptr, vVel.pitch, i, j + 1)) / 2;
    REAL backwardAverage = (F_PITCHACCESS(vVel.ptr, vVel.pitch, i, j - 1) +
     ↪  F_PITCHACCESS(vVel.ptr, vVel.pitch, i, j)) / 2;

    REAL forwardDifference = (F_PITCHACCESS(vVel.ptr, vVel.pitch, i, j) -
     ↪  F_PITCHACCESS(vVel.ptr, vVel.pitch, i, j + 1)) / 2;
    REAL backwardDifference = (F_PITCHACCESS(vVel.ptr, vVel.pitch, i, j - 1) -
     ↪  F_PITCHACCESS(vVel.ptr, vVel.pitch, i, j)) / 2;

    REAL nonDonorTerm = (1 / dely) * (square(forwardAverage) -
     ↪  square(backwardAverage));
    REAL donorTerm = (gamma / dely) * ((abs(forwardAverage) * forwardDifference) -
     ↪  (abs(backwardAverage) * backwardDifference));
    return nonDonorTerm + donorTerm;
}

__device__ REAL PuvPx(PointerWithPitch<REAL> hVel, PointerWithPitch<REAL> vVel, int
↪  i, int j, REAL delX, REAL delY, REAL gamma) {
    REAL jForwardAverageU = (F_PITCHACCESS(hVel.ptr, hVel.pitch, i, j) +
     ↪  F_PITCHACCESS(hVel.ptr, hVel.pitch, i, j + 1)) / 2;
    REAL iForwardAverageV = (F_PITCHACCESS(vVel.ptr, vVel.pitch, i, j) +
     ↪  F_PITCHACCESS(vVel.ptr, vVel.pitch, i + 1, j)) / 2;
    REAL iBackwardAverageV = (F_PITCHACCESS(vVel.ptr, vVel.pitch, i - 1, j) +
     ↪  F_PITCHACCESS(vVel.ptr, vVel.pitch, i, j)) / 2;
    REAL jForwardAverageUDownshift = (F_PITCHACCESS(hVel.ptr, hVel.pitch, i - 1, j)
     ↪  + F_PITCHACCESS(hVel.ptr, hVel.pitch, i - 1, j + 1)) / 2;

    REAL iForwardDifferenceV = (F_PITCHACCESS(vVel.ptr, vVel.pitch, i, j) -
     ↪  F_PITCHACCESS(vVel.ptr, vVel.pitch, i + 1, j)) / 2;
    REAL iBackwardDifferenceV = (F_PITCHACCESS(vVel.ptr, vVel.pitch, i - 1, j) -
     ↪  F_PITCHACCESS(vVel.ptr, vVel.pitch, i, j)) / 2;

    REAL nonDonorTerm = (1 / delX) * ((jForwardAverageU * iForwardAverageV) -
     ↪  (jForwardAverageUDownshift * iBackwardAverageV));
    REAL donorTerm = (gamma / delX) * ((abs(jForwardAverageU) * iForwardDifferenceV)
     ↪  - (abs(jForwardAverageUDownshift) * iBackwardDifferenceV));
    return nonDonorTerm + donorTerm;
}

__device__ REAL PuvPy(PointerWithPitch<REAL> hVel, PointerWithPitch<REAL> vVel, int
 ↪  i, int j, REAL delX, REAL delY, REAL gamma) {
    REAL iForwardAverageV = (F_PITCHACCESS(vVel.ptr, vVel.pitch, i, j) +
     ↪  F_PITCHACCESS(vVel.ptr, vVel.pitch, i + 1, j)) / 2;
    REAL jForwardAverageU = (F_PITCHACCESS(hVel.ptr, hVel.pitch, i, j) +
     ↪  F_PITCHACCESS(hVel.ptr, hVel.pitch, i, j + 1)) / 2;
    REAL jBackwardAverageU = (F_PITCHACCESS(hVel.ptr, hVel.pitch, i, j - 1) +
     ↪  F_PITCHACCESS(hVel.ptr, hVel.pitch, i, j)) / 2;

    REAL iForwardAverageVDownshift = (F_PITCHACCESS(vVel.ptr, vVel.pitch, i, j - 1)
     ↪  + F_PITCHACCESS(vVel.ptr, vVel.pitch, i + 1, j - 1)) / 2;

    REAL jForwardDifferenceU = (F_PITCHACCESS(hVel.ptr, hVel.pitch, i, j) -
     ↪  F_PITCHACCESS(hVel.ptr, hVel.pitch, i, j + 1)) / 2;
    REAL jBackwardDifferenceU = (F_PITCHACCESS(hVel.ptr, hVel.pitch, i, j - 1) -
     ↪  F_PITCHACCESS(hVel.ptr, hVel.pitch, i, j)) / 2;

    REAL nonDonorTerm = (1 / delY) * ((iForwardAverageV * jForwardAverageU) -
     ↪  (iForwardAverageVDownshift * jBackwardAverageU));
    REAL donorTerm = (gamma / delY) * ((abs(iForwardAverageV) * jForwardDifferenceU)
     ↪  - (abs(iForwardAverageVDownshift) * jBackwardDifferenceU));

    return nonDonorTerm + donorTerm;
}

__device__ REAL SecondPuPx(PointerWithPitch<REAL> hVel, int i, int j, REAL delx) {
    return (F_PITCHACCESS(hVel.ptr, hVel.pitch, i + 1, j) - 2 *
     ↪  F_PITCHACCESS(hVel.ptr, hVel.pitch, i, j) + F_PITCHACCESS(hVel.ptr,
     ↪  hVel.pitch, i - 1, j)) / square(delx);
}

__device__ REAL SecondPuPy(PointerWithPitch<REAL> hVel, int i, int j, REAL dely) {
    return (F_PITCHACCESS(hVel.ptr, hVel.pitch, i, j + 1) - 2 *
     ↪  F_PITCHACCESS(hVel.ptr, hVel.pitch, i, j) + F_PITCHACCESS(hVel.ptr,
     ↪  hVel.pitch, i, j - 1)) / square(dely);
}
```

```
__device__ REAL SecondPvPx(PointerWithPitch<REAL> vVel, int i, int j, REAL delx) {
    return (F_PITCHACCESS(vVel.ptr, vVel.pitch, i + 1, j) - 2 *
    ↪  F_PITCHACCESS(vVel.ptr, vVel.pitch, i, j) + F_PITCHACCESS(vVel.ptr,
    ↪  vVel.pitch, i - 1, j)) / square(delx);
}

__device__ REAL SecondPvPy(PointerWithPitch<REAL> vVel, int i, int j, REAL dely) {
    return (F_PITCHACCESS(vVel.ptr, vVel.pitch, i, j + 1) - 2 *
    ↪  F_PITCHACCESS(vVel.ptr, vVel.pitch, i, j) + F_PITCHACCESS(vVel.ptr,
    ↪  vVel.pitch, i, j - 1)) / square(dely);
}

__device__ REAL PpPx(PointerWithPitch<REAL> pressure, int i, int j, REAL delx) {
    return (F_PITCHACCESS(pressure.ptr, pressure.pitch, i + 1, j) -
    ↪  F_PITCHACCESS(pressure.ptr, pressure.pitch, i, j)) / delx;
}

__device__ REAL PpPy(PointerWithPitch<REAL> pressure, int i, int j, REAL dely) {
    return (F_PITCHACCESS(pressure.ptr, pressure.pitch, i, j + 1) -
    ↪  F_PITCHACCESS(pressure.ptr, pressure.pitch, i, j)) / dely;
}

__host__ __device__ REAL square(REAL operand) {
    return operand * operand;
}
```

DiscreteDerivatives.cuh

```
#ifndef DISCRETE_DERIVATIVES_CUH
#define DISCRETE_DERIVATIVES_CUH

#include "Definitions.cuh"

__device__ REAL PuPx(PointerWithPitch<REAL> hVel, int i, int j, REAL delx);

__device__ REAL PvPy(PointerWithPitch<REAL> vVel, int i, int j, REAL dely);

__device__ REAL PuSquaredPx(PointerWithPitch<REAL> hVel, int i, int j, REAL delx,
↪  REAL gamma);

__device__ REAL PvSquaredPy(PointerWithPitch<REAL> vVel, int i, int j, REAL dely,
↪  REAL gamma);

__device__ REAL PuvPx(PointerWithPitch<REAL> hVel, PointerWithPitch<REAL> vVel, int
↪  i, int j, REAL delX, REAL delY, REAL gamma);

__device__ REAL PuvPy(PointerWithPitch<REAL> hVel, PointerWithPitch<REAL> vVel, int
↪  i, int j, REAL delX, REAL delY, REAL gamma);

__device__ REAL SecondPuPx(PointerWithPitch<REAL> hVel, int i, int j, REAL delx);

__device__ REAL SecondPuPy(PointerWithPitch<REAL> hVel, int i, int j, REAL dely);

__device__ REAL SecondPvPx(PointerWithPitch<REAL> vVel, int i, int j, REAL delx);
__device__ REAL SecondPvPy(PointerWithPitch<REAL> vVel, int i, int j, REAL dely);

__device__ REAL PpPx(PointerWithPitch<REAL> pressure, int i, int j, REAL delx);

__device__ REAL PpPy(PointerWithPitch<REAL> pressure, int i, int j, REAL dely);

__host__ __device__ REAL square(REAL operand);

#endif // !DISCRETE_DERIVATIVES_CUH
```

Drag.cu

```
#include "Drag.cuh"
#include <vector>
#include <bitset>

__device__ REAL Magnitude(REAL x, REAL y) {
    return sqrtf(x * x + y * y);
}

__device__ REAL Dot(REAL2 left, REAL2 right) {
    return left.x * right.x + left.y * right.y;
}

__device__ REAL2 GetUnitVector(REAL2 vector) {
    REAL magnitude = Magnitude(vector.x, vector.y);
    return make_REAL2(vector.x / magnitude, vector.y / magnitude);
}

__device__ REAL PVPd(PointerWithPitch<REAL> hVel, PointerWithPitch<REAL> vVel, REAL
↪   distance, int iStart, int jStart, int iExtended, int jExtended) { // As with
↪   DiscreteDerivatives, read this as "Partial V over Partial d" - Partial
↪   derivative of velocity wrt distance from a point.
    REAL extendedVelocityMagnitude = Magnitude(F_PITCHACCESS(hVel.ptr, hVel.pitch,
    ↪   iExtended, jExtended), F_PITCHACCESS(vVel.ptr, vVel.pitch, iExtended,
    ↪   jExtended));
    REAL surfaceVelocityMagnitude = Magnitude(F_PITCHACCESS(hVel.ptr, hVel.pitch,
    ↪   iStart, jStart), F_PITCHACCESS(vVel.ptr, vVel.pitch, iStart, jStart));
    return (extendedVelocityMagnitude - surfaceVelocityMagnitude) / distance;
}

__device__ REAL ComputeWallShear(REAL2* shearUnitVector, PointerWithPitch<REAL>
↪   hVel, PointerWithPitch<REAL> vVel, REAL2 unitNormal, int i, int j, REAL delX,
↪   REAL delY, REAL viscosity) {
    int iExtended, jExtended;
    REAL distance;
    if (unitNormal.x == 0 || unitNormal.y == 0) { // Parallel to an axis
        if (unitNormal.x == 0) {
            distance = delX;
        }
        else {
            distance = delY;
        }
        iExtended = i + (int)unitNormal.x;
        jExtended = j + (int)unitNormal.y;
    }
    else { // 45 degrees to an axis.
        distance = Magnitude(unitNormal.x * delX, unitNormal.y * delY) *
        ↪  DIAGONAL_CELL_DISTANCE;
        iExtended = i + (int)roundf(unitNormal.x * DIAGONAL_CELL_DISTANCE) * 2;
        jExtended = j + (int)roundf(unitNormal.y * DIAGONAL_CELL_DISTANCE) * 2;
    }
    *shearUnitVector = GetUnitVector(make_REAL2(F_PITCHACCESS(hVel.ptr, hVel.pitch,
    ↪  iExtended, jExtended), F_PITCHACCESS(vVel.ptr, vVel.pitch, iExtended,
    ↪  jExtended)));
    REAL wallShear = viscosity * PVPd(hVel, vVel, distance, i, j, iExtended,
    ↪  jExtended);
    return wallShear;
}

__global__ void ComputeViscousDrag(REAL* integrandArray, DragCoordinate*
↪  viscosityCoordinates, int viscosityCoordinatesLength, PointerWithPitch<REAL>
↪  hVel, PointerWithPitch<REAL> vVel, int iMax, int jMax, REAL2 fluidVector, REAL
↪  delX, REAL delY, REAL viscosity) {
    int coordinateNum = blockIdx.x * blockDim.x + threadIdx.x;
    if (coordinateNum >= viscosityCoordinatesLength) return;
    DragCoordinate dragCoordinate = viscosityCoordinates[coordinateNum];
    REAL2 shearDirection;
    REAL wallShear = ComputeWallShear(&shearDirection, hVel, vVel,
    ↪  dragCoordinate.unitNormal, dragCoordinate.coordinate.x,
    ↪  dragCoordinate.coordinate.y, delX, delY, viscosity);
    integrandArray[coordinateNum] = abs(wallShear * Dot(fluidVector,
    ↪  shearDirection)) * dragCoordinate.stepSize;
}

__global__ void ComputeBaselinePressure(REAL* baselinePressure,
↪  PointerWithPitch<REAL> pressure, int iMax, int jMax) {
    *baselinePressure = (F_PITCHACCESS(pressure.ptr, pressure.pitch, 1, 1) +
    ↪  F_PITCHACCESS(pressure.ptr, pressure.pitch, 1, jMax) +
    ↪  F_PITCHACCESS(pressure.ptr, pressure.pitch, iMax, 1) +
    ↪  F_PITCHACCESS(pressure.ptr, pressure.pitch, iMax, jMax)) / 4;
}

__device__ REAL PressureIntegrand(REAL pressure, REAL baselinePressure, REAL2
↪  unitNormal, REAL2 fluidVector) {
    return (pressure - baselinePressure) * Dot(unitNormal, fluidVector);
}

__global__ void ComputePressureDrag(REAL* integrandArray, DragCoordinate*
↪  pressureCoordinates, int pressureCoordinatesLength, PointerWithPitch<REAL>
↪  pressure, int iMax, int jMax, REAL delX, REAL delY, REAL2 fluidVector, REAL*
↪  baselinePressure) {
    int coordinateNum = blockIdx.x * blockDim.x + threadIdx.x;
    if (coordinateNum >= pressureCoordinatesLength) return;
    DragCoordinate dragCoordinate = pressureCoordinates[coordinateNum];
    integrandArray[coordinateNum] = PressureIntegrand(F_PITCHACCESS(pressure.ptr,
    ↪  pressure.pitch, dragCoordinate.coordinate.x, dragCoordinate.coordinate.y),
    ↪  *baselinePressure, dragCoordinate.unitNormal, fluidVector) *
    ↪  dragCoordinate.stepSize;
}
```

Drag.cuh

```
#ifndef DRAG_CUH
#define DRAG_CUH

#include "Definitions.cuh"

__device__ REAL Magnitude(REAL x, REAL y);

__device__ REAL Dot(REAL2 left, REAL2 right);

__device__ REAL2 GetUnitVector(REAL2 vector);

__device__ REAL PVPd(PointerWithPitch<REAL> hVel, PointerWithPitch<REAL> vVel, REAL
↪  distance, int iStart, int jStart, int iExtended, int jExtended);

__device__ REAL ComputeWallShear(REAL2* shearUnitVector, PointerWithPitch<REAL>
↪  hVel, PointerWithPitch<REAL> vVel, REAL2 unitNormal, int i, int j, REAL delX,
↪  REAL delY, REAL viscosity);

__global__ void ComputeViscousDrag(REAL* integrandArray, DragCoordinate*
↪  viscosityCoordinates, int viscosityCoordinatesLength, PointerWithPitch<REAL>
↪  hVel, PointerWithPitch<REAL> vVel, int iMax, int jMax, REAL2 fluidVector, REAL
↪  delX, REAL delY, REAL viscosity);

__global__ void ComputeBaselinePressure(REAL* baselinePressure,
↪  PointerWithPitch<REAL> pressure, int iMax, int jMax);

__device__ REAL PressureIntegrand(REAL pressure, REAL baselinePressure, REAL2
↪  unitNormal, REAL2 fluidVector);

__global__ void ComputePressureDrag(REAL* integrandArray, DragCoordinate*
↪  pressureCoordinates, int pressureCoordinatesLength, PointerWithPitch<REAL>
↪  pressure, int iMax, int jMax, REAL delX, REAL delY, REAL2 fluidVector, REAL*
↪  baselinePressure);
#endif // !DRAG_CUH
```

DragCalculator.cu

```
#include "DragCalculator.cuh"
#include <vector>
#include <bitset>
#include "Drag.cuh"
#include "ReductionKernels.cuh"

REAL DragCalculator::ComputeObstacleDrag(cudaStream_t streamToUse,
↪  PointerWithPitch<REAL> hVel, PointerWithPitch<REAL> vVel, PointerWithPitch<REAL>
↪  pressure, int iMax, int jMax, REAL delX, REAL delY, REAL viscosity)
```

```
{
    REAL2 fluidVector = make_REAL2(-1, 0);

    REAL* viscosityIntegrandArray;
    cudaMalloc(&viscosityIntegrandArray, viscosityCoordinatesLength * sizeof(REAL));
    REAL* pressureIntegrandArray;
    cudaMalloc(&pressureIntegrandArray, pressureCoordinatesLength * sizeof(REAL));

    // Viscous drag calculation
    ComputeViscousDrag KERNEL_ARGS(1, threadsPerBlock, 0, streamToUse)
     ↪  (viscosityIntegrandArray, viscosityCoordinates, viscosityCoordinatesLength,
     ↪  hVel, vVel, iMax, jMax, fluidVector, delX, delY, viscosity);

    REAL* viscosityIntegral;
    cudaMalloc(&viscosityIntegral, sizeof(REAL));
    ComputeFinalSum KERNEL_ARGS(1, threadsPerBlock, threadsPerBlock * sizeof(REAL),
     ↪  streamToUse) (viscosityIntegral, viscosityIntegrandArray,
     ↪  viscosityCoordinatesLength);
    REAL viscousDrag;
    cudaMemcpy(&viscousDrag, viscosityIntegral, sizeof(REAL),
     ↪  cudaMemcpyDeviceToHost);

    REAL* baselinePressure;
    cudaMalloc(&baselinePressure, sizeof(REAL));
    ComputeBaselinePressure KERNEL_ARGS(1, 1, 0, streamToUse) (baselinePressure,
     ↪  pressure, iMax, jMax);

    // Pressure drag calculation
    ComputePressureDrag KERNEL_ARGS(1, threadsPerBlock, 0, streamToUse)
     ↪  (pressureIntegrandArray, pressureCoordinates, pressureCoordinatesLength,
     ↪  pressure, iMax, jMax, delX, delY, fluidVector, baselinePressure);

    REAL* pressureIntegral;
    cudaMalloc(&pressureIntegral, sizeof(REAL));
    ComputeFinalSum KERNEL_ARGS(1, threadsPerBlock, threadsPerBlock * sizeof(REAL),
     ↪  streamToUse) (pressureIntegral, pressureIntegrandArray,
     ↪  pressureCoordinatesLength);
    REAL pressureDrag;
    cudaMemcpy(&pressureDrag, pressureIntegral, sizeof(REAL),
     ↪  cudaMemcpyDeviceToHost);

    cudaFree(viscosityIntegrandArray);
    cudaFree(pressureIntegrandArray);
    cudaFree(baselinePressure);
    cudaFree(pressureIntegral);
    printf("Pressure drag: %f, viscous drag: %f.\n", pressureDrag, viscousDrag);
    return viscousDrag * VISCOSITY_CONVERSION + pressureDrag * PRESSURE_CONVERSION;
}

void DragCalculator::FindPressureCoordinates(BYTE** flags, int iMax, int jMax, REAL
 ↪  delX, REAL delY) {
    std::vector<DragCoordinate> coordinatesVec;
    for (int i = 1; i <= iMax; i++) {
```

```
for (int j = 1; j <= jMax; j++) {
    BYTE flag = flags[i][j];
    if (flag >= 0b00000001 && flag <= 0b00001111) { // This defines boundary
    ↪  cells - all cells without the self bit set except when no bits are
    ↪  set.
        BYTE northBit = (flag & NORTH) >> NORTHSHIFT;
        BYTE eastBit = (flag & EAST) >> EASTSHIFT;
        BYTE southBit = (flag & SOUTH) >> SOUTHSHIFT;
        BYTE westBit = (flag & WEST) >> WESTSHIFT;
        int numEdges = (int)std::bitset<8>(flag).count();
        if (numEdges == 1) {
            DragCoordinate dragCoordinate = DragCoordinate();
            dragCoordinate.coordinate = make_uint2(i + eastBit - westBit, j
            ↪  + northBit - southBit);
            dragCoordinate.unitNormal = make_REAL2((REAL)(eastBit -
            ↪  westBit), (REAL)(northBit - southBit));
            if ((eastBit | westBit) == 1) {
                dragCoordinate.stepSize = delX;
            }
            else {
                dragCoordinate.stepSize = delY;
            }
            coordinatesVec.push_back(dragCoordinate);
        }
        else if (numEdges == 2) {
            int xDirection = eastBit - westBit;
            int yDirection = northBit - southBit;

            // Cell 1: east / west
            DragCoordinate horizontalCoordinate = DragCoordinate();
            horizontalCoordinate.coordinate = make_uint2(i + xDirection, j);
            horizontalCoordinate.unitNormal = make_REAL2((REAL)xDirection,
            ↪  0);
            horizontalCoordinate.stepSize = delX;
            coordinatesVec.push_back(horizontalCoordinate);

            // Cell 2: north / south
            DragCoordinate verticalCoordinate = DragCoordinate();
            verticalCoordinate.coordinate = make_uint2(i, j + yDirection);
            verticalCoordinate.unitNormal = make_REAL2(0, (REAL)yDirection);
            verticalCoordinate.stepSize = delY;
            coordinatesVec.push_back(verticalCoordinate);

            // Cell 3: diagonal
            DragCoordinate diagonalCoordinate = DragCoordinate();
            diagonalCoordinate.unitNormal = make_REAL2(xDirection /
            ↪  DIAGONAL_CELL_DISTANCE, yDirection /
            ↪  DIAGONAL_CELL_DISTANCE);
            diagonalCoordinate.stepSize = (delX + delY) / 2;
            diagonalCoordinate.coordinate = make_uint2(i + xDirection, j +
            ↪  yDirection);
            coordinatesVec.push_back(diagonalCoordinate);
        }
```

```
            }
        }
    }

    pressureCoordinatesLength = (int)coordinatesVec.size();
    cudaMalloc(&pressureCoordinates, pressureCoordinatesLength *
    ↪   sizeof(DragCoordinate));
    cudaMemcpy(pressureCoordinates, coordinatesVec.data(), pressureCoordinatesLength
    ↪   * sizeof(DragCoordinate), cudaMemcpyHostToDevice);
}

void DragCalculator::FindViscosityCoordinates(BYTE** flags, int iMax, int jMax, REAL
↪   delX, REAL delY) {
    std::vector<DragCoordinate> coordinatesVec;
    int lowestY = -1, highestY = -1;
    for (int i = 1; i <= iMax; i++) {
        for (int j = 1; j <= jMax; j++) {
            BYTE flag = flags[i][j];
            if (flag >= 0b00000001 && flag <= 0b00001111) { // This defines boundary
            ↪   cells - all cells without the self bit set except when no bits are
            ↪   set.
                BYTE northBit = (flag & NORTH) >> NORTHSHIFT;
                BYTE eastBit = (flag & EAST) >> EASTSHIFT;
                BYTE southBit = (flag & SOUTH) >> SOUTHSHIFT;
                BYTE westBit = (flag & WEST) >> WESTSHIFT;
                int numEdges = (int)std::bitset<8>(flag).count();
                DragCoordinate dragCoordinate = DragCoordinate();
                dragCoordinate.coordinate = make_uint2(i - westBit, j - southBit);
                if (numEdges == 1) {
                    dragCoordinate.unitNormal = make_REAL2((REAL)(eastBit -
                    ↪   westBit), (REAL)(northBit - southBit));
                    if ((eastBit | westBit) == 1) {
                        dragCoordinate.stepSize = delX;
                    }
                    else {
                        dragCoordinate.stepSize = delY;
                    }
                }
                else if (numEdges == 2) {
                    dragCoordinate.unitNormal = make_REAL2((REAL)(eastBit - westBit)
                    ↪   / DIAGONAL_CELL_DISTANCE, (REAL)(northBit - southBit) /
                    ↪   DIAGONAL_CELL_DISTANCE);
                    dragCoordinate.stepSize = (delX + delY) / 2;
                }
                coordinatesVec.push_back(dragCoordinate);

                if (lowestY == -1 || highestY == -1) {
                    highestY = j;
                    lowestY = j;
                }
                else {
                    if (j > highestY) {
                        highestY = j;
                    }
                    if (j < lowestY) {
                        lowestY = j;
                    }
                }
            }
        }
    }

    viscosityCoordinatesLength = (int)coordinatesVec.size();
    cudaMalloc(&viscosityCoordinates, viscosityCoordinatesLength *
    ↪   sizeof(DragCoordinate));
    cudaMemcpy(viscosityCoordinates, coordinatesVec.data(),
    ↪   viscosityCoordinatesLength * sizeof(DragCoordinate),
    ↪   cudaMemcpyHostToDevice);
    projectionArea = (highestY - lowestY) * delY;
    projectionArea *= projectionArea; // Square to get area.
}

DragCalculator::DragCalculator() : viscosityCoordinates(nullptr),
↪   viscosityCoordinatesLength(0), pressureCoordinates(nullptr),
↪   pressureCoordinatesLength(0), projectionArea(0), threadsPerBlock(1024) {}

DragCalculator::DragCalculator(int threadsPerBlock) : viscosityCoordinates(nullptr),
↪   viscosityCoordinatesLength(0), pressureCoordinates(nullptr),
↪   pressureCoordinatesLength(0), projectionArea(0),
↪   threadsPerBlock(threadsPerBlock) {}

void DragCalculator::ProcessObstacles(BYTE** flags, int iMax, int jMax, REAL delX,
↪   REAL delY) {
    FindPressureCoordinates(flags, iMax, jMax, delX, delY);
    FindViscosityCoordinates(flags, iMax, jMax, delX, delY);
}

REAL DragCalculator::GetDragCoefficient(cudaStream_t streamToUse,
↪   PointerWithPitch<REAL> hVel, PointerWithPitch<REAL> vVel, PointerWithPitch<REAL>
↪   pressure, int iMax, int jMax, REAL delX, REAL delY, REAL viscosity, REAL
↪   density, REAL inflowVelocity) {
    REAL dragForce = ComputeObstacleDrag(streamToUse, hVel, vVel, pressure, iMax,
    ↪   jMax, delX, delY, viscosity);
    return (2 * dragForce) / (density * inflowVelocity * inflowVelocity *
    ↪   projectionArea);
}
```

DragCalculator.cuh

```
#ifndef DRAG_CALCULATOR_H
#define DRAG_CALCULATOR_H

#include "Definitions.cuh"


class DragCalculator
{
```

```
private:
    DragCoordinate* viscosityCoordinates;
    int viscosityCoordinatesLength;

    DragCoordinate* pressureCoordinates;
    int pressureCoordinatesLength;

    REAL projectionArea;

    int threadsPerBlock;

    REAL ComputeObstacleDrag(cudaStream_t streamToUse, PointerWithPitch<REAL> hVel,
     ↪  PointerWithPitch<REAL> vVel, PointerWithPitch<REAL> pressure, int iMax, int
     ↪  jMax, REAL delX, REAL delY, REAL viscosity);

    void FindPressureCoordinates(BYTE** flags, int iMax, int jMax, REAL delX, REAL
     ↪  delY);

    void FindViscosityCoordinates(BYTE** flags, int iMax, int jMax, REAL delX, REAL
     ↪  delY);
public:
    DragCalculator();
    DragCalculator(int threadsPerBlock);

    /// <summary>
    /// Performs internal processing when obstacles are changed.
    /// </summary>
    void ProcessObstacles(BYTE** flags, int iMax, int jMax, REAL delX, REAL delY);

    /// <summary>
    /// Gets the drag coefficient for the obstacle given the fluid flow.
    /// </summary>
    /// <returns>The drag coefficient for the obstacle.</returns>
    REAL GetDragCoefficient(cudaStream_t streamToUse, PointerWithPitch<REAL> hVel,
     ↪  PointerWithPitch<REAL> vVel, PointerWithPitch<REAL> pressure, int iMax, int
     ↪  jMax, REAL delX, REAL delY, REAL viscosity, REAL density, REAL
     ↪  inflowVelocity);
};
#endif // !DRAG_CALCULATOR_H
```

GPUSolver.cu

```
#include "GPUSolver.cuh"
#include "Init.h"
#include "Flags.h"
#include "Boundary.cuh"
#include "Computation.cuh"
#include "PressureComputation.cuh"
#include "math.h"
#include <iostream>

constexpr int GPU_MIN_MAJOR_VERSION = 6;
```

```
GPUSolver::GPUSolver(SimulationParameters parameters, int iMax, int jMax) :
↪  Solver(parameters, iMax, jMax) {
    hVel = PointerWithPitch<REAL>();
    cudaMallocPitch(&hVel.ptr, &hVel.pitch, (jMax + 2) * sizeof(REAL), iMax + 2);

    vVel = PointerWithPitch<REAL>();
    cudaMallocPitch(&vVel.ptr, &vVel.pitch, (jMax + 2) * sizeof(REAL), iMax + 2);

    pressure = PointerWithPitch<REAL>();
    cudaMallocPitch(&pressure.ptr, &pressure.pitch, (jMax + 2) * sizeof(REAL), iMax
    ↪  + 2);

    RHS = PointerWithPitch<REAL>();
    cudaMallocPitch(&RHS.ptr, &RHS.pitch, (jMax + 2) * sizeof(REAL), iMax + 2);

    F = PointerWithPitch<REAL>();
    cudaMallocPitch(&F.ptr, &F.pitch, (jMax + 2) * sizeof(REAL), iMax + 2);

    G = PointerWithPitch<REAL>();
    cudaMallocPitch(&G.ptr, &G.pitch, (jMax + 2) * sizeof(REAL), iMax + 2);

    streamFunction = PointerWithPitch<REAL>();
    cudaMallocPitch(&streamFunction.ptr, &streamFunction.pitch, (jMax + 1) *
    ↪  sizeof(REAL), iMax + 1);

    devFlags = PointerWithPitch<BYTE>();
    cudaMallocPitch(&devFlags.ptr, &devFlags.pitch, (jMax + 2) * sizeof(BYTE), iMax
    ↪  + 2);

    obstacles = ObstacleMatrixMAlloc(iMax + 2, jMax + 2);

    dragCalculator = DragCalculator();

    transmissionHVel = new REAL[iMax * jMax];
    transmissionVVel = new REAL[iMax * jMax];
    transmissionPressure = new REAL[iMax * jMax];
    transmissionStream = new REAL[iMax * jMax];

    copiedHVel = new REAL[(iMax + 2) * (jMax + 2)];
    copiedVVel = new REAL[(iMax + 2) * (jMax + 2)];
    copiedPressure = new REAL[(iMax + 2) * (jMax + 2)];
    copiedStream = new REAL[(iMax + 1) * (jMax + 1)];

    devCoordinates = nullptr; // Initialised in ProcessObstacles.
    streams = nullptr; // Initialised in PerformSetup.
}

GPUSolver::~GPUSolver() {
    if (streams != nullptr) {
        for (int i = 0; i < totalStreams; i++) {
            cudaStreamDestroy(streams[i]); // Destroy all of the streams
        }
    }
    cudaFree(hVel.ptr);
    cudaFree(vVel.ptr);
    cudaFree(pressure.ptr);
    cudaFree(RHS.ptr);
    cudaFree(F.ptr);
    cudaFree(G.ptr);
    cudaFree(streamFunction.ptr);
    cudaFree(devFlags.ptr);
    cudaFree(devCoordinates);

    FreeMatrix(obstacles, iMax + 2);

    delete[] streams;

    delete[] transmissionHVel;
    delete[] transmissionVVel;
    delete[] transmissionPressure;
    delete[] transmissionStream;

    delete[] copiedHVel;
    delete[] copiedVVel;
    delete[] copiedPressure;
    delete[] copiedStream;
}

template<typename T>
cudaError_t GPUSolver::CopyFieldToDevice(PointerWithPitch<T> devField, T**
↪   hostField, int xLength, int yLength)
{
    T* hostFieldFlattened = new T[xLength * yLength];
    FlattenArray(hostField, 0, 0, hostFieldFlattened, 0, 0, 0, xLength, yLength);

    cudaError_t retVal = cudaMemcpy2D(devField.ptr, devField.pitch,
    ↪   hostFieldFlattened, yLength * sizeof(T), yLength * sizeof(T), xLength,
    ↪   cudaMemcpyHostToDevice);
    delete[] hostFieldFlattened;

    return retVal;
}

template<typename T>
cudaError_t GPUSolver::CopyFieldToHost(PointerWithPitch<T> devField, T** hostField,
↪   int xLength, int yLength) {
    T* hostFieldFlattened = new T[xLength * yLength];

    cudaError_t retVal = cudaMemcpy2D(hostFieldFlattened, yLength * sizeof(T),
    ↪   devField.ptr, devField.pitch, yLength * sizeof(T), xLength,
    ↪   cudaMemcpyDeviceToHost);

    UnflattenArray(hostField, 0, 0, hostFieldFlattened, 0, 0, 0, xLength, yLength);
    delete[] hostFieldFlattened;
    return retVal;
}


void GPUSolver::SetBlockDimensions()
{
    // The below code takes the square root of the number of threads, but if the
    ↪  number of threads per block is not a square it takes the powers of 2 either
    ↪  side of the square root.
    // For example, a maxThreadsPerBlock of 1024 would mean threadsPerBlock becomes
    ↪  32 and 32, but a maxThreadsPerBlock of 512 would mean threadsPerBlock would
    ↪  become 32 and 16
    int maxThreadsPerBlock = deviceProperties.maxThreadsPerBlock;
    int log2ThreadsPerBlock = (int)ceil(log2f((float)maxThreadsPerBlock)); //
    ↪  Threads per block should be a power of 2, but ceil just in case
    int log2XThreadsPerBlock = (int)ceilf((float)log2ThreadsPerBlock / 2.0f); //
    ↪  Divide by 2, if log2(threadsPerBlock) was odd, ceil
    int log2YThreadsPerBlock = (int)floorf((float)log2ThreadsPerBlock / 2.0f); // As
    ↪  above, but floor for smaller one
    int xThreadsPerBlock = (int)powf(2, (float)log2XThreadsPerBlock); // Now
    ↪  exponentiate to get the actual number of threads
    int yThreadsPerBlock = (int)powf(2, (float)log2YThreadsPerBlock);
    threadsPerBlock = dim3(xThreadsPerBlock, yThreadsPerBlock);

    int blocksForIMax = (int)ceilf((float)iMax / threadsPerBlock.x);
    int blocksForJMax = (int)ceilf((float)jMax / threadsPerBlock.y);
    numBlocks = dim3(blocksForIMax, blocksForJMax);
}

void GPUSolver::ResizeField(REAL* enlargedField, int enlargedXLength, int
↪  enlargedYLength, int xOffset, int yOffset, REAL* transmissionField, int xLength,
↪  int yLength) {
    for (int i = 0; i < xLength; i++) {
        memcpy(
            transmissionField + i * yLength,
            enlargedField + (i + xOffset) * enlargedYLength + yOffset,
            yLength * sizeof(REAL)
        );
    }
}

REAL* GPUSolver::GetHorizontalVelocity() const {
    return transmissionHVel;
}

REAL* GPUSolver::GetVerticalVelocity() const {
    return transmissionVVel;
}

REAL* GPUSolver::GetPressure() const {
    return transmissionPressure;
}
```

```
REAL* GPUSolver::GetStreamFunction() const {
    return transmissionStream;
}

bool** GPUSolver::GetObstacles() const {
    return obstacles;
}

REAL GPUSolver::GetDragCoefficient() {
    return dragCalculator.GetDragCoefficient(streams[0], hVel, vVel, pressure, iMax,
    ↪   jMax, delX, delY, parameters.dynamicViscosity, parameters.fluidDensity,
    ↪   parameters.inflowVelocity);
}

void GPUSolver::ProcessObstacles() { // When this function is called, no streams
↪   have been created and block dimensions have not been calculated. Therefore, no
↪   kernels can be launched here.
    BYTE** hostFlags = FlagMatrixMAlloc(iMax + 2, jMax + 2);
    SetFlags(obstacles, hostFlags, iMax + 2, jMax + 2); // Set the flags on the
    ↪   host.

    uint2* hostCoordinates; // Obstacle coordinates are put here first, then copied
    ↪   to the GPU.
    FindBoundaryCells(hostFlags, hostCoordinates, coordinatesLength, iMax, jMax);

    numFluidCells = CountFluidCells(hostFlags, iMax, jMax);

    dragCalculator.ProcessObstacles(hostFlags, iMax, jMax, parameters.width / iMax,
    ↪   parameters.height / jMax);

    cudaMalloc(&devCoordinates, coordinatesLength * sizeof(uint2));

    cudaMemcpy(devCoordinates, hostCoordinates, coordinatesLength * sizeof(uint2),
    ↪   cudaMemcpyHostToDevice); // Copy the flags and coordinates arrays to the
    ↪   device.
    CopyFieldToDevice(devFlags, hostFlags, iMax + 2, jMax + 2);

    FreeMatrix(hostFlags, iMax + 2);
    delete[] hostCoordinates;
}

void GPUSolver::PerformSetup() {
    cudaGetDeviceProperties(&deviceProperties, 0);

    SetBlockDimensions();

    streams = new cudaStream_t[totalStreams];
    for (int i = 0; i < totalStreams; i++) {
        cudaStreamCreate(&streams[i]);
    }

    delX = parameters.width / iMax;
    delY = parameters.height / jMax;
```

```
}

void GPUSolver::Timestep(REAL& simulationTime) {
    REAL* hostTimestep = nullptr; // Set heap or global mem pointers to nullptr so
     ↪  freeing has no effect and only free label is needed.
    REAL* timestep = nullptr;
    REAL* gamma = nullptr;
    REAL pressureResidualNorm = 0;
    int pressureIterations = 0;
    dim3 numBlocksForStreamCalc(INT_DIVIDE_ROUND_UP(iMax + 1, threadsPerBlock.x),
     ↪  INT_DIVIDE_ROUND_UP(jMax + 1, threadsPerBlock.y));

    // Perform computations
    if (SetBoundaryConditions(streams, threadsPerBlock.x * threadsPerBlock.y, hVel,
     ↪  vVel, devFlags, devCoordinates, coordinatesLength, iMax, jMax,
     ↪  parameters.inflowVelocity, parameters.surfaceFrictionalPermissibility) !=
     ↪  cudaSuccess) goto free; // Illegal address

    cudaMalloc(&timestep, sizeof(REAL)); // Allocate a new device variable for
     ↪  timestep

    if (ComputeTimestep(timestep, streams, hVel, vVel, iMax, jMax, delX, delY,
     ↪  parameters.reynoldsNo, parameters.timeStepSafetyFactor) != cudaSuccess) goto
     ↪  free;

    hostTimestep = new REAL; // Copy the device timestep so it can be added to
     ↪  simulation time
    if (cudaMemcpyAsync(hostTimestep, timestep, sizeof(REAL),
     ↪  cudaMemcpyDeviceToHost, streams[computationStreams + 0]) != cudaSuccess)
     ↪  goto free;

    if (cudaMalloc(&gamma, sizeof(REAL)) != cudaSuccess) goto free; // Allocate
     ↪  gamma on the device and then calculate it
    if (ComputeGamma(gamma, streams, threadsPerBlock.x * threadsPerBlock.y, hVel,
     ↪  vVel, iMax, jMax, timestep, delX, delY) != cudaSuccess) goto free;

    if (ComputeFG(streams, threadsPerBlock, hVel, vVel, F, G, devFlags, iMax, jMax,
     ↪  timestep, delX, delY, parameters.bodyForces.x, parameters.bodyForces.y,
     ↪  gamma, parameters.reynoldsNo) != cudaSuccess) goto free;

    ComputeRHS KERNEL_ARGS(numBlocks, threadsPerBlock, 0, streams[0]) (F, G, RHS,
     ↪  devFlags, iMax, jMax, timestep, delX, delY); // ComputeRHS is simple enough
     ↪  not to need a wrapper
    if (cudaStreamSynchronize(streams[0]) != cudaSuccess) goto free; // Need to
     ↪  synchronise because pressure depends on RHS.

    pressureIterations = Poisson(streams, threadsPerBlock, pressure, RHS, devFlags,
     ↪  devCoordinates, coordinatesLength, numFluidCells, iMax, jMax, numColoursSOR,
     ↪  delX, delY, parameters.pressureResidualTolerance,
     ↪  parameters.pressureMinIterations, parameters.pressureMaxIterations,
     ↪  parameters.relaxationParameter, &pressureResidualNorm);
    if (pressureIterations == 0) goto free; // Here 0 is the error case.
    cudaMemcpy2DAsync(copiedPressure, (jMax + 2) * sizeof(REAL), pressure.ptr,
     ↪   pressure.pitch, (jMax + 2) * sizeof(REAL), iMax + 2, cudaMemcpyDeviceToHost,
     ↪   streams[computationStreams + 0]); // Pressure is unchanged after this point,
     ↪   so can copy it async

    if (ComputeVelocities(streams, threadsPerBlock, hVel, vVel, F, G, pressure,
     ↪   devFlags, iMax, jMax, timestep, delX, delY) != cudaSuccess) goto free;

    cudaMemcpy2DAsync(copiedHVel, (jMax + 2) * sizeof(REAL), hVel.ptr, hVel.pitch,
     ↪   (jMax + 2) * sizeof(REAL), iMax + 2, cudaMemcpyDeviceToHost,
     ↪   streams[computationStreams + 1]); // Velocities are unchanged after this
     ↪   point, copy them
    cudaMemcpy2DAsync(copiedVVel, (jMax + 2) * sizeof(REAL), vVel.ptr, vVel.pitch,
     ↪   (jMax + 2) * sizeof(REAL), iMax + 2, cudaMemcpyDeviceToHost,
     ↪   streams[computationStreams + 2]);

    ComputeStream KERNEL_ARGS(numBlocksForStreamCalc, threadsPerBlock, 0,
     ↪   streams[0]) (hVel, streamFunction, iMax, jMax, delY);

    cudaMemcpy2DAsync(copiedStream, (jMax + 1) * sizeof(REAL), streamFunction.ptr,
     ↪   streamFunction.pitch, (jMax + 1) * sizeof(REAL), iMax + 1,
     ↪   cudaMemcpyDeviceToHost, streams[computationStreams + 3]); // Stream function
     ↪   is the last to be calculated, copy it once it is ready.

    // Resize all of the fields for transmission.
    ResizeField(copiedHVel, iMax + 2, jMax + 2, 1, 1, transmissionHVel, iMax, jMax);
    ResizeField(copiedVVel, iMax + 2, jMax + 2, 1, 1, transmissionVVel, iMax, jMax);
    ResizeField(copiedPressure, iMax + 2, jMax + 2, 1, 1, transmissionPressure,
     ↪   iMax, jMax);
    ResizeField(copiedStream, iMax + 1, jMax + 1, 1, 1, transmissionStream, iMax,
     ↪   jMax);

    simulationTime += *hostTimestep; // Only add to the simulation time if the
     ↪   timestep was successful. Error case is therefore simulationTime unchanged
     ↪   after Timestep returns.


free: // Pointers that need to be freed even if timestep is unsuccessful.
    delete hostTimestep;
    cudaFree(timestep);
    cudaFree(gamma);
}

bool GPUSolver::IsDeviceSupported() {
    int count;
    cudaGetDeviceCount(&count);
    if (count > 0) {
     cudaDeviceProp properties;
     cudaGetDeviceProperties(&properties, 0);
     if (properties.major >= GPU_MIN_MAJOR_VERSION) {
         return true;
     }
    }
    return false;
}
```

GPUSolver.cuh

```
#ifndef GPUSOLVER_CUH
#define GPUSOLVER_CUH

#include "Definitions.cuh"
#include "Solver.h"
#include "DragCalculator.cuh"

constexpr int computationStreams = 4; // Number of streams for launching
↪  parallelisable computation kernels
constexpr int memcpyStreams = 4; // Number of streams for launching parallel memory
↪  copies
constexpr int totalStreams = computationStreams + memcpyStreams;

class GPUSolver :
    public Solver
{
private:
    PointerWithPitch<REAL> hVel; // Horizontal velocity, resides on device.
    PointerWithPitch<REAL> vVel; // Vertical velocity, resides on device.
    PointerWithPitch<REAL> pressure; // Pressure, resides on device.
    PointerWithPitch<REAL> RHS; // Pressure equation RHS, resides on device.
    PointerWithPitch<REAL> streamFunction; // Stream function, resides on device.
    PointerWithPitch<REAL> F; // Quantity F, resides on device.
    PointerWithPitch<REAL> G; // Quantity G, resides on device.
    PointerWithPitch<BYTE> devFlags; // Cell flags, resides on device.

    REAL* transmissionHVel;
    REAL* transmissionVVel;
    REAL* transmissionPressure;
    REAL* transmissionStream;
    REAL* copiedHVel;
    REAL* copiedVVel;
    REAL* copiedPressure;
    REAL* copiedStream;

    REAL delX; // Step size in x direction, resides on host.
    REAL delY; // Step size in y direction, resides on host.
    REAL* timestep; // Timestep, resides on device.

    uint2* devCoordinates; // Array of obstacle coordinates, resides on device.
    int coordinatesLength; // Length of coordinates array
    int numFluidCells;
    const int numColoursSOR = 2;

    dim3 numBlocks; // Number of blocks for a grid of iMax x jMax threads.
    dim3 threadsPerBlock; // Maximum number of threads per block in a 2D square
    ↪  allocation.
    DragCalculator dragCalculator;

    bool** obstacles; // 2D array of obstacles, resides on host.

    cudaDeviceProp deviceProperties;
    cudaStream_t* streams; // Streams that can be used. First are the computation
    ↪  streams (0 to the number of computation streams), then are memcpy streams.
    ↪  To access memcpy streams, first add computationStreams as an offset

    template<typename T>
    cudaError_t CopyFieldToDevice(PointerWithPitch<T> devField, T** hostField, int
    ↪  xLength, int yLength);

    template<typename T>
    cudaError_t CopyFieldToHost(PointerWithPitch<T> devField, T** hostField, int
    ↪  xLength, int yLength);

    void SetBlockDimensions();

    void ResizeField(REAL* enlargedField, int enlargedXLength, int enlargedYLength,
    ↪  int xOffset, int yOffset, REAL* transmissionField, int xLength, int
    ↪  yLength);

public:
    GPUSolver(SimulationParameters parameters, int iMax, int jMax);

    ~GPUSolver();

    REAL* GetHorizontalVelocity() const;

    REAL* GetVerticalVelocity() const;

    REAL* GetPressure() const;

    REAL* GetStreamFunction() const;

    bool** GetObstacles() const;

    REAL GetDragCoefficient();

    void ProcessObstacles();

    void PerformSetup();

    void Timestep(REAL& simulationTime); // Implementing abstract inherited method

    static bool IsDeviceSupported();
};

#endif // !GPUSOLVER_CUH
```

kernel.cu

```
#include "pch.h"
#include "Solver.h"
#include "GPUSolver.cuh"
#include "BackendCoordinator.h"
#include <iostream>
#include <chrono>

int main(int argc, char** argv) {
    SimulationParameters parameters = SimulationParameters();
    if (argc == 1 || (argc == 2 && strcmp(argv[1], "debug") == 0)) { // Not linked
    ↪  to a frontend.
        std::cout << "Running without a fronted attached.\n";
        int iMax = 500;
        int jMax = 250;
        parameters.width = 1;
        parameters.height = 1;
        parameters.timeStepSafetyFactor = (REAL)0.5;
        parameters.relaxationParameter = (REAL)1.7;
        parameters.pressureResidualTolerance = 2;
        parameters.pressureMinIterations = 5;
        parameters.pressureMaxIterations = 2000;
        parameters.reynoldsNo = 10000;
        parameters.dynamicViscosity = (REAL)0.00001983;
        parameters.fluidDensity = (REAL)1.293;
        parameters.inflowVelocity = 5;
        parameters.surfaceFrictionalPermissibility = 0;
        parameters.bodyForces.x = 0;
        parameters.bodyForces.y = 0;

        GPUSolver solver = GPUSolver(parameters, iMax, jMax);

        bool** obstacles = solver.GetObstacles();
        for (int i = 1; i <= iMax; i++) { for (int j = 1; j <= jMax; j++) {
        ↪  obstacles[i][j] = 1; } } // Set all the cells to fluid

        int boundaryLeft = (int)(0.45 * iMax);
        int boundaryRight = (int)(0.55 * iMax);
        int boundaryBottom = (int)(0.45 * jMax);
        int boundaryTop = (int)(0.55 * jMax);
        for (int i = boundaryLeft; i < boundaryRight; i++) { // Create a square of
        ↪  boundary cells
            for (int j = boundaryBottom; j < boundaryTop; j++) {
                obstacles[i][j] = 0;
            }
        }

        std::cout << "Obstacle set to a box.\n";

        solver.ProcessObstacles();
        solver.PerformSetup();

        REAL cumulativeTimestep = 0;
        int numIterations = 100;
        std::cerr << "2 seconds to attach profiler / debugger\n";
        Sleep(2000);

        for (int i = 0; i < numIterations; i++) {
            solver.Timestep(cumulativeTimestep);
            REAL dragCoefficient = solver.GetDragCoefficient();
            std::cout << "Iteration " << i << ", time taken: " << cumulativeTimestep
            ↪  << ", drag coefficient " << dragCoefficient << ".\n";
        }
        return 0;
    }
    else if (argc == 4) {
        std::cout << "Linked to a frontend.\n";

        char* pipeName = argv[1];
        int iMax = atoi(argv[2]);
        int jMax = atoi(argv[3]);
        if (iMax == 0 || jMax == 0) { // Set defaults
            iMax = 200;
            jMax = 100;
        }
        std::cout << "iMax and jMax set to " << iMax << " and " << jMax << ".\n";
        Solver* solver = new GPUSolver(parameters, iMax, jMax);
        BackendCoordinator backendCoordinator(iMax, jMax, std::string(pipeName),
        ↪  solver);
        int retValue = backendCoordinator.Run();
        delete solver;
        return retValue;
    }
    else {
        std::cerr << "Incorrect number of command-line arguments. Run the executable
        ↪  with the pipe name and field dimensions to connect to a frontend, or
        ↪  without to run without a frontend.\n";
        return -1;
    }
}
```

PressureComputation.cu

```
#include "PressureComputation.cuh"
#include "DiscreteDerivatives.cuh"
#include "ReductionKernels.cuh"
#include <cmath>


/// <summary>
/// Calculates and validates the coordinates of a thread based on the coloured cell
↪  system.
/// </summary>
/// <param name="rowNum">The output row number (x coordinate).</param>
/// <param name="colNum">The output column number (y coordinate).</param>
/// <param name="threadPosX">X position of the thread in the grid.</param>
/// <param name="threadPosY">Y position of the thread in the grid.</param>
/// <param name="colourNum">The desired colour of the returned coordinates</param>
/// <param name="numberOfColours">The number of different colours assigned to
↪   cells.</param>
/// <returns>Whether the thread coordinates map to a valid cell.</returns>
__device__ bool CalculateColouredCoordinates(int* rowNum, int* colNum, int
↪   threadPosX, int threadPosY, int colourNum, int numberOfColours, int iMax, int
↪   jMax) {
    // Require (rowNum + colNum) % numberOfColours = colourNum
    *rowNum = threadPosX + 1; // Normal rowNum - x position of thread in grid.
    int colOffset = colourNum - *rowNum % numberOfColours - 1; // The number to add
    ↪   to the column number (the required result of colNum % numberOfColours,
    ↪   subtracted 1 to avoid colNum being 0).
    if (colOffset < 0) {
        colOffset += numberOfColours;
    }

    *colNum = numberOfColours * threadPosY + colOffset + 1; // Generate colNum such
    ↪   that colNum % numberOfColours = colOffset.

    return *rowNum <= iMax && *colNum <= jMax;
}

/// <summary>
/// Gets the parity (number of bits that are set mod 2) of a byte.
/// </summary>
/// <param name="input">The input byte</param>
/// <returns>The parity of the byte, 1 or 0.</returns>
__host__ __device__ BYTE GetParity(BYTE input) {
    input ^= input >> 4; // Repeatedly shift-XOR to end up with the XOR of all of
    ↪   the bits in the LSB.
    input ^= input >> 2;
    input ^= input >> 1;
    return input & 1; // The parity is stored in the last bit, so XOR the result
    ↪   with 1 and return.
}


/// <summary>
/// Calculates pressures of one colour of the grid. Requires iMax x (jMax /
↪   numberOfColours) threads.
/// </summary>
__global__ void SingleColourSOR(int numberOfColours, int colourNum,
↪   PointerWithPitch<REAL> pressure, PointerWithPitch<REAL> RHS,
↪   PointerWithPitch<BYTE> flags, PointerWithPitch<REAL> residualArray, int iMax,
↪   int jMax, REAL delX, REAL delY, REAL omega, REAL boundaryFraction)
{
    int rowNum, colNum;
    bool validCell = CalculateColouredCoordinates(&rowNum, &colNum, blockIdx.x *
    ↪   blockDim.x + threadIdx.x, blockIdx.y * blockDim.y + threadIdx.y, colourNum,
    ↪   numberOfColours, iMax, jMax);

    if (!validCell) return; // If the cell is not valid, do not perform the
    ↪   computation
    if ((B_PITCHACCESS(flags.ptr, flags.pitch, rowNum, colNum) & SELF) == 0) return;
    ↪  // If the cell is not a fluid cell, also do not perform the computation.

    REAL relaxedPressure = (1 - omega) * F_PITCHACCESS(pressure.ptr, pressure.pitch,
    ↪  rowNum, colNum);
    REAL pressureAverages = ((F_PITCHACCESS(pressure.ptr, pressure.pitch, rowNum +
    ↪  1, colNum) + F_PITCHACCESS(pressure.ptr, pressure.pitch, rowNum - 1,
    ↪  colNum)) / square(delX)) + ((F_PITCHACCESS(pressure.ptr, pressure.pitch,
    ↪  rowNum, colNum + 1) + F_PITCHACCESS(pressure.ptr, pressure.pitch, rowNum,
    ↪  colNum - 1)) / square(delY)) - F_PITCHACCESS(RHS.ptr, RHS.pitch, rowNum,
    ↪  colNum);
    F_PITCHACCESS(pressure.ptr, pressure.pitch, rowNum, colNum) = relaxedPressure +
    ↪  boundaryFraction * pressureAverages;

    REAL currentResidual = pressureAverages - (2 * F_PITCHACCESS(pressure.ptr,
    ↪  pressure.pitch, rowNum, colNum)) / square(delX) - (2 *
    ↪  F_PITCHACCESS(pressure.ptr, pressure.pitch, rowNum, colNum)) / square(delY);

    F_PITCHACCESS(residualArray.ptr, residualArray.pitch, rowNum - 1, colNum - 1) =
    ↪  square(currentResidual); // Residual array is shifted down and left 1 so
    ↪  less memory is needed.
}

/// <summary>
/// Copies pressure values at the top and bottom of the simulation domain. Requires
↪  iMax threads.
/// </summary>
__global__ void CopyHorizontalPressures(PointerWithPitch<REAL> pressure, int iMax,
↪  int jMax) {
    int rowNum = blockIdx.x * blockDim.x + threadIdx.x + 1;
    if (rowNum > iMax) return;

    F_PITCHACCESS(pressure.ptr, pressure.pitch, rowNum, 0) =
    ↪  F_PITCHACCESS(pressure.ptr, pressure.pitch, rowNum, 1);
    F_PITCHACCESS(pressure.ptr, pressure.pitch, rowNum, jMax + 1) =
    ↪  F_PITCHACCESS(pressure.ptr, pressure.pitch, rowNum, jMax);
}

/// <summary>
/// Copies pressure values at the top and bottom of the simulation domain. Requires
↪  jMax threads.
/// </summary>
__global__ void CopyVerticalPressures(PointerWithPitch<REAL> pressure, int iMax, int
↪  jMax) {
    int colNum = blockIdx.x * blockDim.x + threadIdx.x + 1;
    if (colNum > jMax) return;

    F_PITCHACCESS(pressure.ptr, pressure.pitch, 0, colNum) =
    ↪  F_PITCHACCESS(pressure.ptr, pressure.pitch, 1, colNum);
    F_PITCHACCESS(pressure.ptr, pressure.pitch, iMax + 1, colNum) =
    ↪  F_PITCHACCESS(pressure.ptr, pressure.pitch, iMax, colNum);
}
```

```
/// <summary>
/// Copies the pressures for the boundary cells given in <paramref
↪ name="coordinates" />. Requires <paramref name="coordinatesLength" /> threads.
/// </summary>
__global__ void CopyBoundaryPressures(PointerWithPitch<REAL> pressure, uint2*
↪ coordinates, int coordinatesLength, PointerWithPitch<BYTE> flags, int iMax, int
↪ jMax) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index >= coordinatesLength) return;

    uint2 coordinate = coordinates[index]; // Get coordinate from global memory...
    BYTE relevantFlag = B_PITCHACCESS(flags.ptr, flags.pitch, coordinate.x,
    ↪ coordinate.y); // ...and the flag for that coordinate.

    int xShift = ((relevantFlag & EAST) >> EASTSHIFT) - ((relevantFlag & WEST) >>
    ↪ WESTSHIFT); // Relative position of cell to copy in x direction. -1, 0 or
    ↪ 1.
    int yShift = ((relevantFlag & NORTH) >> NORTHSHIFT) - ((relevantFlag & SOUTH) >>
    ↪ SOUTHSHIFT); // Relative position of cell to copy in y direction. -1, 0 or
    ↪ 1.

    if (GetParity(relevantFlag) == 1) { // Only boundary cells with one edge - copy
    ↪ from that fluid cell
        F_PITCHACCESS(pressure.ptr, pressure.pitch, coordinate.x, coordinate.y) =
        ↪ F_PITCHACCESS(pressure.ptr, pressure.pitch, coordinate.x + xShift,
        ↪ coordinate.y + yShift); // Copy from the cell determined by the shifts.
    }
    else { // These are boundary cells with 2 edges - take the average of the 2
    ↪ cells with the boundary.
        F_PITCHACCESS(pressure.ptr, pressure.pitch, coordinate.x, coordinate.y) =
        ↪ (F_PITCHACCESS(pressure.ptr, pressure.pitch, coordinate.x + xShift,
        ↪ coordinate.y) + F_PITCHACCESS(pressure.ptr, pressure.pitch,
        ↪ coordinate.x, coordinate.y + yShift)) / (REAL)2; // Take the average of
        ↪ the one above/below and the one left/right by only using one shift for
        ↪ each of the field accesses.
    }
}

// Could implement this using cuda graphs
int Poisson(cudaStream_t* streams, dim3 threadsPerBlock, PointerWithPitch<REAL>
↪ pressure, PointerWithPitch<REAL> RHS, PointerWithPitch<BYTE> flags, uint2*
↪ coordinates, int coordinatesLength, int numFluidCells, int iMax, int jMax, int
↪ numColours, REAL delX, REAL delY, REAL residualTolerance, int minIterations, int
↪ maxIterations, REAL omega, REAL* residualNorm) {
    cudaError_t retVal;
    int numIterations = 0;
    REAL boundaryFraction = omega / ((2 / square(delX)) + (2 / square(delY))); //
    ↪ Only executed once so easier to execute on CPU and transfer.

    dim3 numBlocks(INT_DIVIDE_ROUND_UP(iMax, threadsPerBlock.x),
    ↪ INT_DIVIDE_ROUND_UP(jMax / numColours, threadsPerBlock.y)); // Number of
    ↪ blocks for an iMax x jMax launch.
```

```
int threadsPerBlockFlattened = threadsPerBlock.x * threadsPerBlock.y;
int numBlocksIMax = INT_DIVIDE_ROUND_UP(iMax, threadsPerBlockFlattened);
int numBlocksJMax = INT_DIVIDE_ROUND_UP(jMax, threadsPerBlockFlattened);

PointerWithPitch<REAL> residualArray = PointerWithPitch<REAL>(); // Create
↪  pointers and set them to null
REAL* d_residualNorm = nullptr; // Create a device version of residualNorm

retVal = cudaMallocPitch(&residualArray.ptr, &residualArray.pitch, jMax *
↪  sizeof(REAL), iMax); // Create residualArray with size iMax * jMax
if (retVal != cudaSuccess) goto free;

retVal = cudaMalloc(&d_residualNorm, sizeof(REAL)); // Allocate one REAL's worth
↪  of memory
if (retVal != cudaSuccess) goto free;


*residualNorm = 0; // Set both host and device residual norms to 0.
retVal = cudaMemset(d_residualNorm, 0, sizeof(REAL));
do {
    if (retVal != cudaSuccess) goto free;

    for (int colourNum = 0; colourNum < numColours; colourNum++) { // Loop
    ↪  through however many colours and perform SOR.
        SingleColourSOR KERNEL_ARGS(numBlocks, threadsPerBlock, 0, streams[0])
        ↪  (numColours, colourNum, pressure, RHS, flags, residualArray, iMax,
        ↪  jMax, delX, delY, omega, boundaryFraction);
    }

    retVal = cudaStreamSynchronize(streams[0]);
    if (retVal != cudaSuccess) goto free;

    // Copy the boundary cell pressures all in different streams
    CopyHorizontalPressures KERNEL_ARGS(numBlocksIMax, threadsPerBlockFlattened,
    ↪  0, streams[0]) (pressure, iMax, jMax);
    CopyVerticalPressures KERNEL_ARGS(numBlocksJMax, threadsPerBlockFlattened,
    ↪  0, streams[1]) (pressure, iMax, jMax);
    CopyBoundaryPressures KERNEL_ARGS(numBlocks, threadsPerBlock, 0, streams[2])
    ↪  (pressure, coordinates, coordinatesLength, flags, iMax, jMax);

    if (numIterations % 10 == 0) { // Only calculate the residual every 10
    ↪  iterations
        retVal = FieldSum(d_residualNorm, streams[0], residualArray, iMax,
        ↪  jMax);
        if (retVal != cudaSuccess) goto free;
        retVal = cudaMemcpy(residualNorm, d_residualNorm, sizeof(REAL),
        ↪  cudaMemcpyDeviceToHost); // Copy residual norm to host for
        ↪  processing and use in condition
        if (retVal != cudaSuccess) goto free;

        *residualNorm = sqrt(*residualNorm / numFluidCells);
    }
```

```
        numIterations++;
    } while ((numIterations < maxIterations && *residualNorm > residualTolerance) ||
     ↪  numIterations < minIterations);

free:
    cudaFree(residualArray.ptr);
    cudaFree(d_residualNorm);
    return retVal == cudaSuccess ? numIterations : 0; // Return 0 if there was an
     ↪  error, otherwise the number of iterations.
}
```

PressureComputation.cuh

```
#ifndef PRESSURE_COMPUTATION_CUH

#include "Definitions.cuh"

/// <summary>
/// Performs SOR iterations to solve the pressure poisson equation. Handles kernel
↪  launching internally. Requires 4 streams.
/// </summary>
/// <param name="coordinates">The coordinates of the boundary cells.</param>
/// <param name="coordinatesLength">The length of the coordinates array.</param>
/// <param name="omega">Relaxation between 0 and 2.</param>
/// <param name="residualNorm">The residual norm of the final iteration. This is an
↪  output variable, and does not need to be allocated.</param>
int Poisson(cudaStream_t* streams, dim3 threadsPerBlock, PointerWithPitch<REAL>
↪  pressure, PointerWithPitch<REAL> RHS, PointerWithPitch<BYTE> flags, uint2*
↪  coordinates, int coordinatesLength, int numFluidCells, int iMax, int jMax, int
↪  numColours, REAL delX, REAL delY, REAL residualTolerance, int minIterations, int
↪  maxIterations, REAL omega, REAL* residualNorm);

#endif // !PRESSURE_COMPUTATION_CUH
```

ReductionKernels.cu

```
#include "ReductionKernels.cuh"
#include <cmath>

#ifdef __INTELLISENSE__ // Allow intellisense to recognise cooperative groups
#define __CUDACC__
#endif // __INTELLISENSE__
#include <cooperative_groups.h>
#ifdef __INTELLISENSE__
#undef __CUDACC__
#endif // __INTELLISENSE__

namespace cg = cooperative_groups;

/// <summary>
/// Computes the max of the elements in <paramref name="sharedArray" />. Processes
↪  the number of elements equal to <paramref name="group" />'s size.
/// </summary>
/// <param name="group">The thread group of which the calling thread is a
↪  member.</param>
/// <param name="sharedArray">The array, in shared memory, to find the maximum
↪  of.</param>
__device__ void GroupMax(cg::thread_group group, volatile REAL* sharedArray) {
    int index = group.thread_rank();
    REAL val = sharedArray[index];
    for (int indexThreshold = group.size() / 2; indexThreshold > 0; indexThreshold
    ↪  /= 2) {
        if (index < indexThreshold) { // Halve the number of threads each iteration
            val = fmax(val, sharedArray[index + indexThreshold]); // Get the max of
            ↪  the thread's own value and the one at index + indexThreshold
            sharedArray[index] = val; // Store the max into the shared array at the
            ↪  current index
        }
        group.sync();
    }
}
/// <summary>
/// Computes the maximum of each column of a field. Requires xLength blocks, each of
↪  <c>field.pitch / sizeof(REAL)</c> threads, and 1 REAL's worth of shared memory
↪  per thread.
/// </summary>
/// <param name="partialMaxes">An array of length equal to the number of rows, for
↪  outputting the maxes of each column.</param>
/// <param name="field">The input field.</param>
/// <param name="yLength">The length of a column.</param>
__global__ void ComputePartialMaxes(REAL* partialMaxes, PointerWithPitch<REAL>
↪  field, int yLength) {
    cg::thread_block threadBlock = cg::this_thread_block();
    REAL* colBase = (REAL*)((char*)field.ptr + blockIdx.x * field.pitch);

    // Perform copy to shared memory.
    // Put a 0 in shared if current index is greater than yLength (this catches
    ↪  index in pitch padding, or index > size of a row)
    extern __shared__ REAL sharedArray[];

    if (threadIdx.x < yLength) { // the index of the thread is greater than the
    ↪  length of a column.
        sharedArray[threadIdx.x] = *(colBase + threadIdx.x);
    }
    else {
        sharedArray[threadIdx.x] = (REAL)0;
    }
    threadBlock.sync();

    GroupMax(threadBlock, sharedArray);

    if (threadIdx.x == 0) { // If the thread is the 0th in the block, store its
    ↪  result to global memory.
        partialMaxes[blockIdx.x] = sharedArray[0];
    }
}
```

```
__global__ void ComputePartialMaxes(REAL* partialMaxes, REAL* array, int
↪  arrayLength) {
    cg::thread_block threadBlock = cg::this_thread_block();
    REAL* startIndex = array + blockIdx.x * blockDim.x; // Start of this thread
     ↪  block's memory allocation

    // Perform copy to shared memory.
    // Put a 0 in shared if current index is greater than yLength (this catches
     ↪  index in pitch padding, or index > size of a row)
    extern __shared__ REAL sharedArray[];

    if (blockIdx.x * blockDim.x + threadIdx.x < arrayLength) { // the index of the
     ↪  thread is greater than the length of a column.
        sharedArray[threadIdx.x] = *(startIndex + threadIdx.x);
    }
    else {
        sharedArray[threadIdx.x] = (REAL)0;
    }
    threadBlock.sync();

    GroupMax(threadBlock, sharedArray);

    if (threadIdx.x == 0) { // If the thread is the 0th in the block, store its
     ↪  result to global memory.
        partialMaxes[blockIdx.x] = sharedArray[0];
    }
}

__global__ void ComputeFinalMax(REAL* max, REAL* partialMaxes, int xLength)
{
    cg::thread_block threadBlock = cg::this_thread_block();

    extern __shared__ REAL sharedMem[];

    // Copy to shared memory again
    if (threadIdx.x < xLength) {
        sharedMem[threadIdx.x] = partialMaxes[threadIdx.x];
    }
    else {
        sharedMem[threadIdx.x] = (REAL)0;
    }
    threadBlock.sync();

    GroupMax(threadBlock, sharedMem);
    if (threadIdx.x == 0) { // Thread 0 stores the final element.
        *max = sharedMem[0];
    }
}

/// <summary>
/// Computes the sum of the elements in <paramref name="sharedArray" />. Processes
↪  the number of elements equal to <paramref name="group" />'s size.
```

```
/// </summary>
/// <param name="group">The thread group of which the calling thread is a
↪   member.</param>
/// <param name="sharedArray">The array, in shared memory, to find the sum
↪   of.</param>
__device__ void GroupSum(cg::thread_group group, volatile REAL* sharedArray) {
    int index = group.thread_rank();
    for (int indexThreshold = group.size() / 2; indexThreshold > 0; indexThreshold
    ↪   /= 2) {
        if (index < indexThreshold) { // Halve the number of threads each iteration
            sharedArray[index] += sharedArray[index + indexThreshold]; // Add the
            ↪   value at index + indexThreshold to the value at the current index.
        }
        group.sync();
    }
}

/// <summary>
/// Computes the sum of each column of a field. Requires xLength blocks, each of
↪   <c>field.pitch / sizeof(REAL)</c> threads, and 1 REAL's worth of shared memory
↪   per thread.
/// </summary>
/// <param name="partialSums">An array of length equal to the number of rows, for
↪   outputting the sums of each column.</param>
/// <param name="field">The input field.</param>
/// <param name="yLength">The length of a column.</param>
__global__ void ComputePartialSums(REAL* partialSums, PointerWithPitch<REAL> field,
↪   int yLength) {
    cg::thread_block threadBlock = cg::this_thread_block();
    REAL* colBase = (REAL*)((char*)field.ptr + blockIdx.x * field.pitch);

    // Perform copy to shared memory.
    // Put a 0 in shared if current index is greater than yLength (this catches
    ↪   index in pitch padding, or index > size of a row)
    extern __shared__ REAL sharedArray[];

    if (threadIdx.x < yLength) { // the index of the thread is greater than the
    ↪   length of a column.
        sharedArray[threadIdx.x] = *(colBase + threadIdx.x);
    }
    else {
        sharedArray[threadIdx.x] = (REAL)0;
    }
    threadBlock.sync();

    GroupSum(threadBlock, sharedArray);

    if (threadIdx.x == 0) { // If the thread is the 0th in the block, store its
    ↪   result to global memory.
        partialSums[blockIdx.x] = sharedArray[0];
    }
}
```

```
/// <summary>
/// Computes partial sums of an array. Requires 1 REAL's worth of shared memory per
↪   thread.
/// </summary>
/// <param name="partialMaxes">The output array, length equal to the number of
↪   blocks spawned.</param>
/// <param name="array">The array to calculate the sum of.</param>
/// <param name="arrayLength">The length of the array.</param>
__global__ void ComputePartialSums(REAL* partialMaxes, REAL* array, int arrayLength)
↪   {
    cg::thread_block threadBlock = cg::this_thread_block();
    REAL* startIndex = array + blockIdx.x * blockDim.x; // Start of this thread
    ↪   block's memory allocation

    // Perform copy to shared memory.
    // Put a 0 in shared if current index is greater than yLength (this catches
    ↪   index in pitch padding, or index > size of a row)
    extern __shared__ REAL sharedArray[];

    if (blockIdx.x * blockDim.x + threadIdx.x < arrayLength) { // the index of the
    ↪   thread is greater than the length of a column.
        sharedArray[threadIdx.x] = *(startIndex + threadIdx.x);
    }
    else {
        sharedArray[threadIdx.x] = (REAL)0;
    }
    threadBlock.sync();

    GroupSum(threadBlock, sharedArray);

    if (threadIdx.x == 0) { // If the thread is the 0th in the block, store its
    ↪   result to global memory.
        partialMaxes[blockIdx.x] = sharedArray[0];
    }
}

/// <summary>
/// Computes the final sum from a given array of partial sums. Requires 1 block of
↪   <paramref name="xLength" /> threads, and 1 REAL's worth of shared memory per
↪   thread.
/// </summary>
/// <param name="sum">The location to place the output.</param>
/// <param name="partialSums">An array of partial sums, of size <paramref
↪   name="xLength" />.</param>
__global__ void ComputeFinalSum(REAL* sum, REAL* partialSums, int xLength)
{
    cg::thread_block threadBlock = cg::this_thread_block();

    extern __shared__ REAL sharedMem[];

    // Copy to shared memory again
    if (threadIdx.x < xLength) {
        sharedMem[threadIdx.x] = partialSums[threadIdx.x];
    }
    else {
        sharedMem[threadIdx.x] = (REAL)0;
    }
    threadBlock.sync();

    GroupSum(threadBlock, sharedMem);
    if (threadIdx.x == 0) { // Thread 0 stores the final element.
        *sum = sharedMem[0];
    }
}

cudaError_t FieldMax(REAL* max, cudaStream_t streamToUse, PointerWithPitch<REAL>
↪   field, int xLength, int yLength) {
    cudaError_t retVal;

    REAL* partialMaxes;
    retVal = cudaMalloc(&partialMaxes, xLength * sizeof(REAL));
    if (retVal != cudaSuccess) { // Return if there was an error in allocation
        return retVal;
    }

    // Run the GPU kernel:
    ComputePartialMaxes KERNEL_ARGS(xLength, (unsigned int)field.pitch /
    ↪   sizeof(REAL), field.pitch, streamToUse) (partialMaxes, field, yLength); // 1
    ↪   block per row. Number of threads is equal to column pitch, and each thread
    ↪   has 1 REAL worth of shared memory.
    retVal = cudaStreamSynchronize(streamToUse);
    if (retVal != cudaSuccess) { // Skip the rest of the computation if there was an
    ↪   error
        goto free;
    }

    ComputeFinalMax KERNEL_ARGS(1, xLength, xLength * sizeof(REAL), streamToUse)
    ↪   (max, partialMaxes, xLength); // 1 block to process all of the partial
    ↪   maxes, number of threads equal to number of partial maxes (xLength is also
    ↪   this)
    retVal = cudaStreamSynchronize(streamToUse);


free:
    cudaFree(partialMaxes);
    return retVal;
}

cudaError_t FieldSum(REAL* sum, cudaStream_t streamToUse, PointerWithPitch<REAL>
↪   field, int xLength, int yLength) {
    cudaError_t retVal;

    REAL* partialSums;
    retVal = cudaMalloc(&partialSums, xLength * sizeof(REAL));
    if (retVal != cudaSuccess) { // Return if there was an error in allocation
        return retVal;
    }

    // Run the GPU kernel:
    ComputePartialSums KERNEL_ARGS(xLength, (unsigned int)field.pitch /
     ↪  sizeof(REAL), field.pitch, streamToUse) (partialSums, field, yLength); // 1
     ↪  block per row. Number of threads is equal to column pitch, and each thread
     ↪  has 1 REAL worth of shared memory.
    retVal = cudaStreamSynchronize(streamToUse);
    if (retVal != cudaSuccess) { // Skip the rest of the computation if there was an
     ↪  error
        goto free;
    }

    ComputeFinalSum KERNEL_ARGS(1, xLength, xLength * sizeof(REAL), streamToUse)
     ↪  (sum, partialSums, xLength); // 1 block to process all of the partial sums,
     ↪  number of threads equal to number of partial sums (xLength is also this)
    retVal = cudaStreamSynchronize(streamToUse);

free:
    cudaFree(partialSums);
    return retVal;
}
```

ReductionKernels.cuh

```
#ifndef REDUCTION_KERNELS_CUH

#include "Definitions.cuh"

/// <summary>
/// Computes partial maxes of an array. Requires 1 REAL's worth of shared memory per
↪  thread.
/// </summary>
/// <param name="partialMaxes">The output array, length equal to the number of
↪  blocks spawned.</param>
/// <param name="array">The array to calculate the max of.</param>
/// <param name="arrayLength">The length of the array.</param>
__global__ void ComputePartialMaxes(REAL* partialMaxes, REAL* array, int
↪  arrayLength);

/// <summary>
/// Computes the final max from a given array of partial maxes. Requires 1 block of
↪  <paramref name="xLength" /> threads, and 1 REAL's worth of shared memory per
↪  thread.
/// </summary>
/// <param name="max">The location to place the output.</param>
/// <param name="partialMaxes">An array of partial maxes, of size <paramref
↪  name="xLength" />.</param>
__global__ void ComputeFinalMax(REAL* max, REAL* partialMaxes, int xLength);

/// <summary>
/// Computes the max of a given field.The field's width and height must each be no
↪  larger than the max number of threads per block.
/// </summary>
/// <param name="max">The location to place the output</param>
/// <returns>An error code, or <c>cudaSuccess</c>.</returns>
cudaError_t FieldMax(REAL* max, cudaStream_t streamToUse, PointerWithPitch<REAL>
↪  field, int xLength, int yLength);

/// <summary>
/// Computes partial sums of an array. Requires 1 REAL's worth of shared memory per
↪  thread.
/// </summary>
/// <param name="partialMaxes">The output array, length equal to the number of
↪  blocks spawned.</param>
/// <param name="array">The array to calculate the sum of.</param>
/// <param name="arrayLength">The length of the array.</param>
__global__ void ComputePartialSums(REAL* partialMaxes, REAL* array, int
↪  arrayLength);

/// <summary>
/// Computes the final sum from a given array of partial sums. Requires 1 block of
↪  <paramref name="xLength" /> threads, and 1 REAL's worth of shared memory per
↪  thread.
/// </summary>
/// <param name="sum">The location to place the output.</param>
/// <param name="partialSums">An array of partial sums, of size <paramref
↪  name="xLength" />.</param>
__global__ void ComputeFinalSum(REAL* sum, REAL* partialSums, int xLength);

/// <summary>
/// Computes the sum of a given field.The field's width and height must each be no
↪  larger than the max number of threads per block.
/// </summary>
/// <param name="sum">The location to place the output</param>
/// <returns>An error code, or <c>cudaSuccess</c>.</returns>
cudaError_t FieldSum(REAL* sum, cudaStream_t streamToUse, PointerWithPitch<REAL>
↪  field, int xLength, int yLength);

#endif // !REDUCTION_KERNELS_CUH
```

App.xaml

```
<Application x:Class="UserInterface.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:UserInterface"
             Startup="Start"
             ShutdownMode="OnMainWindowClose"
             Exit="Application_Exit">
    <Application.Resources>
        <Style TargetType="Label">
            <Setter Property="FontSize" Value="16" />
        </Style>
        <Style TargetType="Button">
            <Setter Property="FontSize" Value="16" />
        </Style>
        <Style TargetType="ComboBoxItem">
```

```
            <Setter Property="FontSize" Value="16" />
        </Style>
        <Style TargetType="TextBlock">
            <Setter Property="FontSize" Value="16" />
        </Style>
        <Style TargetType="ComboBox">
            <Setter Property="VerticalContentAlignment" Value="Center" />
            <Setter Property="FontSize" Value="16" />
        </Style>
    </Application.Resources>
</Application>
```

App.xaml.cs

```
using System;
using System.Threading;
using System.Windows;
using System.Windows.Controls;
using UserInterface.HelperClasses;
using UserInterface.ViewModels;
using UserInterface.Views;

#pragma warning disable CS8618 // Compiler doesn't understand that Start() is
↪   functionally the constructor for this class.

namespace UserInterface
{
    /// <summary>
    /// Interaction logic for App.xaml
    /// </summary>
    public partial class App : Application
    {
        private const int POPUP_HEIGHT = 400;
        private const int POPUP_WIDTH = 700;
        private UserControl currentUserControl;
        private UserControl? currentPopup;
        private MainWindow fullScreenWindowContainer; // 2 different container
        ↪   windows to allow for usercontrols to either be popups (that don't take
        ↪   up the whole screen), or fullscreen
        private PopupWindow popupWindowContainer;

        private ParameterHolder parameterHolder;
        private UnitHolder unitHolder;
        private ObstacleHolder obstacleHolder;

        public static event EventHandler<UserControlChangeEventArgs>?
        ↪   UserControlChanged;
        public static event EventHandler<UserControlChangeEventArgs>? PopupCreated;
        public static event EventHandler<EventArgs>? PopupDeleted;

        private void ChangeUserControl(object? sender, UserControlChangeEventArgs e)
        {
```

```
        currentUserControl =
        ↪  (UserControl)Activator.CreateInstance(e.NewUserControlType,
        ↪  [parameterHolder, unitHolder, obstacleHolder]); // Use the Type
        ↪  parameter to create a new instance

        fullScreenWindowContainer.Content = currentUserControl;
    }

    private void CreatePopup(object? sender, UserControlChangeEventArgs e)
    {
        currentPopup =
        ↪  (UserControl)Activator.CreateInstance(e.NewUserControlType,
        ↪  [parameterHolder, unitHolder, obstacleHolder]);
        popupWindowContainer.Content = currentPopup;

        popupWindowContainer.ShowDialog();
    }

    private void DeletePopup(object? sender, EventArgs e)
    {
        currentPopup = null;
        popupWindowContainer.Content = currentPopup;

        popupWindowContainer.Hide();
    }

    // Static methods for other classes to invoke events without the App
    ↪  instance.
    public static void RaiseUserControlChanged(object? sender,
    ↪  UserControlChangeEventArgs e)
    {
        UserControlChanged.Invoke(sender, e);
    }

    public static void RaisePopupCreated(object? sender,
    ↪  UserControlChangeEventArgs e)
    {
        PopupCreated.Invoke(sender, e);
    }

    public static void RaisePopupDeleted(object? sender, EventArgs e)
    {
        PopupDeleted.Invoke(sender, e);
    }

    public void Start(object Sender, StartupEventArgs e)
    {
        fullScreenWindowContainer = new MainWindow(); // Initialise container
        ↪  windows
        popupWindowContainer = new PopupWindow
        {
            Height = POPUP_HEIGHT,
            Width = POPUP_WIDTH
        };

        parameterHolder = new(DefaultParameters.WIDTH, DefaultParameters.HEIGHT,
            ↪  DefaultParameters.TIMESTEP_SAFETY_FACTOR,
            ↪  DefaultParameters.RELAXATION_PARAMETER,
            ↪  DefaultParameters.PRESSURE_RESIDUAL_TOLERANCE,
            ↪  DefaultParameters.PRESSURE_MAX_ITERATIONS,
            ↪  DefaultParameters.REYNOLDS_NUMBER,
            ↪  DefaultParameters.FLUID_VISCOSITY, DefaultParameters.FLUID_VELOCITY,
            ↪  DefaultParameters.FLUID_DENSITY, DefaultParameters.SURFACE_FRICTION,
            ↪  new FieldParameters(), DefaultParameters.DRAW_CONTOURS,
            ↪  DefaultParameters.CONTOUR_TOLERANCE,
            ↪  DefaultParameters.NUM_CONTOURS); // Use the defaults from
            ↪  DefaultParameters constant holder
        unitHolder = new UnitHolder();
        obstacleHolder = new ObstacleHolder(null, true);

        currentUserControl = new ConfigScreen(parameterHolder, unitHolder,
         ↪  obstacleHolder);
        fullScreenWindowContainer.Content = currentUserControl;
        fullScreenWindowContainer.Show();

        UserControlChanged += ChangeUserControl;
        PopupCreated += CreatePopup;
        PopupDeleted += DeletePopup;
    }

    private void Application_Exit(object sender, ExitEventArgs e)
    {
        if (currentUserControl is SimulationScreen simulationScreen) // Close
            ↪  the backend if it is running when application exits (current screen
            ↪  will be SimulationScreen).
        {
            SimulationScreenVM viewModel = simulationScreen.ViewModel;
            if (viewModel.BackendStatus == BackendStatus.Running)
            {
                viewModel.BackendCommand.Execute(null);
            }
            Thread.Sleep(100); // Give the backend time to stop
            simulationScreen.ViewModel.CloseBackend();
        }
    }
}

public class UserControlChangeEventArgs : EventArgs // EventArgs derivative
 ↪  containing the typename of the new user control
{
    public Type NewUserControlType { get; }
    public UserControlChangeEventArgs(Type newUserControlType) : base()
    {
        NewUserControlType = newUserControlType;
    }
}
```

```
}
```

AssemblyInfo.cs

```
using System.Windows;

[assembly: ThemeInfo(
    ResourceDictionaryLocation.None, // where theme specific resource dictionaries
     ↪  are located
                                     // (used if a resource is not found in the
                                      ↪  page,
                                     // or application resource dictionaries)
    ResourceDictionaryLocation.SourceAssembly // where the generic resource
     ↪  dictionary is located
                                              // (used if a resource is not found in
                                               ↪  the page,
                                              // app, or any theme specific resource
                                               ↪  dictionaries)
)]
```

BoolToTickPlacement.cs

```
using System;
using System.Windows.Controls.Primitives;
using System.Windows.Data;

namespace UserInterface.Converters
{
    [ValueConversion(typeof(bool), typeof(TickPlacement))]
    public class BoolToTickPlacement : IValueConverter
    {
        public object Convert(object value, Type targetType, object parameter,
         ↪  System.Globalization.CultureInfo culture)
        {
            if (value is not bool)
            {
                return TickPlacement.None;
            }
            if ((bool)value)
            {
                return TickPlacement.BottomRight;
            }
            return TickPlacement.None;
        }

        public object ConvertBack(object value, Type targetType, object parameter,
         ↪  System.Globalization.CultureInfo culture)
        {
            throw new InvalidOperationException("Conversion not allowed");
        }
    }
}
```

PolarListToRectList.cs

```
using System;
using System.Collections.ObjectModel;
using System.Globalization;
using System.Windows;
using System.Windows.Data;
using System.Windows.Media;
using UserInterface.HelperClasses;

namespace UserInterface.Converters
{
    public class PolarListToRectList : IMultiValueConverter
    {
        /// <summary>
        /// Converts a list of polar coordinates to a list of rectangular
        ↪ coordinates with a specified origin.
        /// </summary>
        /// <param name="values">An array of polar point observable collection, and
        ↪ origin (as fractions of the canvas size. Either a <see cref="Point"/> or
        ↪ <see cref="string"/> that can be converted to a point).</param>
        /// <returns>A <see cref="PointCollection"/> of rectangular
        ↪ points.</returns>
        /// <exception cref="NotImplementedException"></exception>
        public object Convert(object[] values, Type targetType, object parameter,
        ↪ CultureInfo culture)
        {
            RectangularToPolar RectToPolConverter = new RectangularToPolar();

            if (values[0] is not ObservableCollection<PolarPoint>)
            {
                return DependencyProperty.UnsetValue;
            }
            ObservableCollection<PolarPoint> polarPoints =
            ↪ (ObservableCollection<PolarPoint>)values[0];
            object origin = values[1]; // Allow RectToPolConverter to do the
            ↪ conversion

            PointCollection points = new PointCollection();
            foreach (PolarPoint point in polarPoints)
            {
                Point rectangularPoint =
                ↪ (Point)RectToPolConverter.ConvertBack(point, targetType, origin,
                ↪ culture);
                points.Add(new Point(rectangularPoint.X, 100 - rectangularPoint.Y));
                ↪ // Flip the y coordinates.
            }
            return points;
        }

        public object[] ConvertBack(object value, Type[] targetTypes, object
        ↪ parameter, CultureInfo culture)
        {
            throw new InvalidOperationException("Conversion not allowed.");
        }
    }
}
```

RectangularToPolar.cs

```
using System;
using System.Globalization;
using System.Linq;
using System.Windows;
using System.Windows.Data;
using UserInterface.HelperClasses;

namespace UserInterface.Converters
{
    /// <summary>
    /// Converter class that can convert rectangular to polar and back again with
    ↪  respect to a given pole.
    /// </summary>
    public class RectangularToPolar : IValueConverter
    {
        /// <summary>
        /// Converts a rectangular coordinate to a polar coordinate, with the pole
        ↪  to use optionally specified in rectangular coordinates in <paramref
        ↪  name="parameter"/>
        /// </summary>
        /// <param name="value">The rectangular coordinate, as a <see
        ↪  cref="Point"/>.</param>
        /// <param name="parameter">The pole to use in the conversion (as a
        ↪  rectangular <see cref="Point"/> or <see cref="string"/>), or <see
        ↪  cref="null"/> to use (0, 0).</param>
        /// <returns>A <see cref="PolarPoint"/> representing the converted
        ↪  coordinate.</returns>
        public object Convert(object value, Type targetType, object parameter,
        ↪  CultureInfo culture)
        {
            if (value is not Point || !(parameter is Point || parameter is string ||
            ↪  parameter is null))
            {
                return DependencyProperty.UnsetValue;
            }

            Point point = (Point)value;
            Point origin;
            if (parameter is Point)
            {
                origin = (Point)parameter;
            }
            else if (parameter is string)
            {
                string parameterNoSpaces =
                ↪  new(((string)parameter).ToCharArray().Where(c =>
                ↪  !char.IsWhiteSpace(c)).ToArray());
                string[] parameters = parameterNoSpaces.Split(',');
                if (!double.TryParse(parameters[0], out double x) ||
                 ↪   !double.TryParse(parameters[1], out double y))
                {
                    return DependencyProperty.UnsetValue;
                }
                origin = new Point(x, y);
            }
            else // parameter is null
            {
                origin = new Point(0, 0); // Origin not specified - use default.
            }

            Vector distFromOrigin = point - origin;

            double angle = Math.Atan2(distFromOrigin.Y, distFromOrigin.X); // Range
             ↪   -pi to pi.
            if (angle < 0) // Make the range 0 to 2 pi
            {
                angle += 2 * Math.PI;
            }

            return new PolarPoint(distFromOrigin.Length, angle); // This is the only
                ↪   line that actually converts a rectangular coordinate to a polar
                ↪   one.
        }

        /// <summary>
        /// Converts a polar coordinate to a rectangular coordinate, with the pole
            ↪   to use optionally specified in rectangular coordinates in <paramref
            ↪   name="parameter"/>
        /// </summary>
        /// <param name="value">The polar coordinate, as a <see
         ↪   cref="PolarPoint"/>.</param>
        /// <param name="parameter">The pole to use in the conversion (as a
            ↪   rectangular <see cref="Point"/> or <see cref="string"/>), or <see
            ↪   cref="null"/> to use (0, 0).</param>
        /// <returns>A <see cref="Point"/> representing the converted
         ↪   coordinate.</returns>
        public object ConvertBack(object value, Type targetType, object parameter,
         ↪   CultureInfo culture)
        {
            if (value is not PolarPoint || !(parameter is Point || parameter is
             ↪   string || parameter is null))
            {
                return DependencyProperty.UnsetValue;
            }

            PolarPoint point = (PolarPoint)value;
            Point origin;
            if (parameter is Point)
            {
                origin = (Point)parameter;
            }
            else if (parameter is string)
            {
                string parameterNoSpaces =
                ↪   new(((string)parameter).ToCharArray().Where(c =>
                ↪   !char.IsWhiteSpace(c)).ToArray());
                string[] parameters = parameterNoSpaces.Split(',');

                if (!double.TryParse(parameters[0], out double x) ||
                ↪   !double.TryParse(parameters[1], out double y))
                {
                    return DependencyProperty.UnsetValue;
                }
                origin = new Point(x, y);
            }
            else // parameter is null
            {
                origin = new Point(0, 0); // Origin not specified - use default.
            }

            Vector distanceFromOrigin = new Vector(point.Radius *
            ↪   Math.Cos(point.Angle), point.Radius * Math.Sin(point.Angle)); //
            ↪   Convert the polar coordinate to a rectangular vector.

            return origin + distanceFromOrigin; // Translate the origin by the
            ↪   vector to get the final rectangular point.
        }
    }
}
```

SignificantFigures.cs

```
using System;
using System.Globalization;
using System.Windows.Data;

namespace UserInterface.Converters
{
    [ValueConversion(typeof(float), typeof(string))]
    public class SignificantFigures : IValueConverter
    {
        /// <summary>
        /// Rounds an input value to a number of significant figures, returning a
        ↪   string to be displayed.
        /// </summary>
        /// <param name="value">The value, as a <see cref="float"/>, to
        ↪   round.</param>
        /// <param name="parameter">The number of significant figures to round to,
        ↪   as a <see cref="string"/> representation of an int.</param>
        /// <returns>The rounded value, cast from a <see cref="string"/> to an <see
        ↪   cref="object"/>.</returns>
        public object Convert(object value, Type targetType, object parameter,
        ↪   CultureInfo culture)
        {
            // Input validation
            if (!int.TryParse((string)parameter, out int iParameter))
            {
                return "";
            }
            float fValue;
            if (value is double dValue)
            {
                fValue = (float)dValue;
            }
            else if (value is float)
            {
                fValue = (float)value;
            }
            else
            {
                return "";
            }


            return fValue.ToString($"G{iParameter}"); // Use the ToString method
             ↪  with the number of SF as the parameter to it.
        }

        public object ConvertBack(object value, Type targetType, object parameter,
         ↪  CultureInfo culture)
        {
            throw new InvalidOperationException("Values cannot be converted back
             ↪  once they have been rounded.");
        }
    }
}
```

UnitClasses.cs

```
using System;

namespace UserInterface.Converters
{
    /// <summary>
    /// Static constant-containing class that defines the conversion ratios to
     ↪  convert from the non-SI unit to the corresponding SI unit. For example, to
     ↪  convert a value in feet to a value in metres, multiply the value in feet by
     ↪  <see cref="LengthMultipliers.Feet"/>.
    /// </summary>
    public class UnitClasses // All values to 6 sig fig
    {
        public enum UnitSystem
        {
            SI,
            CGS,
            DefaultImperial,
            OtherMetric,
            OtherImperial
```

```
}

/// <summary>
/// Multipliers to convert length units to metres.
/// </summary>
private static class LengthMultipliers
{
    public static readonly double Feet = 0.304800;
    public static readonly double Inches = 0.025400;
    public static readonly double Centimetres = 0.010000;
    public static readonly double Millimetres = 0.001000;
}
/// <summary>
/// Multipliers to convert speed units to metres per second.
/// </summary>
private static class SpeedMultipliers
{
    public static readonly double MilesPerHour = 0.447040;
    public static readonly double KilometresPerHour = 0.277778;
    public static readonly double CentimetresPerSecond =
     ↪  LengthMultipliers.Centimetres;
    public static readonly double FeetPerSecond = LengthMultipliers.Feet;
}
/// <summary>
/// Multipliers to convert time units to seconds.
/// </summary>
private static class TimeMultipliers
{
    public static readonly double Milliseconds = 0.001;
    public static readonly double Minutes = 60;
    public static readonly double Hours = 3600;
}
/// <summary>
/// Multipliers to convert density units to kilograms per cubic metre.
/// </summary>
private static class DensityMultipliers
{
    public static readonly double GramsPerCubicCentimetre = 1000.00;
    public static readonly double PoundsPerCubicInch = 27679.90;
    public static readonly double PoundsPerCubicFoot = 16.01846;
}
/// <summary>
/// Multipliers to convert viscosity units to kilograms per metre per
 ↪  second.
/// </summary>
private static class ViscosityMultipliers
{
    public static readonly double GramsPerCentimetrePerSecond = 0.1000000;
    public static readonly double PoundSecondsPerSquareFoot = 47.800000;
}

public abstract class Unit
{
    /// <summary>
    /// Multiplier when converting from this unit to the corresponding SI
     ↪  unit.
    /// </summary>
    protected readonly double conversionRatio;
    protected readonly string longName;
    protected readonly string shortName;

    public double ConversionRatio => conversionRatio;
    public string LongName => longName;
    public string ShortName => shortName;
    public Unit(double conversionRatio, string longName, string shortName)
    {
        this.conversionRatio = conversionRatio;
        this.longName = longName;
        this.shortName = shortName;
    }
}

public class Dimensionless() : Unit(1, "", "");

public abstract class LengthUnit(double conversionRatio, string longName,
 ↪  string shortName) : Unit(conversionRatio, longName, shortName) { }
public abstract class SpeedUnit(double conversionRatio, string longName,
 ↪  string shortName) : Unit(conversionRatio, longName, shortName) { }
public abstract class TimeUnit(double conversionRatio, string longName,
 ↪  string shortName) : Unit(conversionRatio, longName, shortName) { }
public abstract class DensityUnit(double conversionRatio, string longName,
 ↪  string shortName) : Unit(conversionRatio, longName, shortName) { }
public abstract class ViscosityUnit(double conversionRatio, string longName,
 ↪  string shortName) : Unit(conversionRatio, longName, shortName) { }

public class Metre() : LengthUnit(1, "Metres", "m");
public class Foot() : LengthUnit(LengthMultipliers.Feet, "Feet", "Ft");
public class Inch() : LengthUnit(LengthMultipliers.Inches, "Inches", "In");
public class Centimetre() : LengthUnit(LengthMultipliers.Centimetres,
 ↪  "Centimetres", "cm");
public class Millimetre() : LengthUnit(LengthMultipliers.Millimetres,
 ↪  "Millimetres", "mm");

public class MetrePerSecond() : SpeedUnit(1, "Metres per second", "m/s");
public class CentimetrePerSecond() :
    ↪  SpeedUnit(SpeedMultipliers.CentimetresPerSecond, "Centimetres per
    ↪  second", "cm/s");
public class MilePerHour() : SpeedUnit(SpeedMultipliers.MilesPerHour, "Miles
 ↪  per hour", "mph");
public class KilometrePerHour() :
    ↪  SpeedUnit(SpeedMultipliers.KilometresPerHour, "Kilometres per hour",
    ↪  "km/h");
public class FootPerSecond() : SpeedUnit(SpeedMultipliers.FeetPerSecond,
 ↪  "Feet per second", "Ft/s");

public class Second() : TimeUnit(1, "Seconds", "s");
```

```
        public class Millisecond() : TimeUnit(TimeMultipliers.Milliseconds,
        ↪   "Milliseconds", "ms");
        public class Minute() : TimeUnit(TimeMultipliers.Minutes, "Minutes", "min");
        public class Hour() : TimeUnit(TimeMultipliers.Hours, "Hours", "hr");

        public class KilogramPerCubicMetre() : DensityUnit(1, "Kilograms per cubic
        ↪   metre", "kg/m³");
        public class GramPerCubicCentimetre() :
        ↪   DensityUnit(DensityMultipliers.GramsPerCubicCentimetre, "Grams per cubic
        ↪   centimetre", "g/cm³");
        public class PoundPerCubicInch() :
        ↪   DensityUnit(DensityMultipliers.PoundsPerCubicInch, "Pounds per cubic
        ↪   inch", "lb/in³");
        public class PoundPerCubicFoot() :
        ↪   DensityUnit(DensityMultipliers.PoundsPerCubicFoot, "Pounds per cubic
        ↪   foot", "lb/ft³");

        public class KilogramPerMetrePerSecond() : ViscosityUnit(1, "Kilograms per
        ↪   metre per second", "kg/m s");
        public class GramPerCentimetrePerSecond() :
        ↪   ViscosityUnit(ViscosityMultipliers.GramsPerCentimetrePerSecond, "Grams
        ↪   per centimetre per second", "g/cm s");
        public class PoundSecondPerSquareFoot() :
        ↪   ViscosityUnit(ViscosityMultipliers.PoundSecondsPerSquareFoot,
        ↪   "Pound-seconds per square foot", "lb·s/ft²");

        public class UnitSystemList
        {
            public readonly UnitSystem unitSystem;
            public readonly LengthUnit lengthUnit;
            public readonly SpeedUnit speedUnit;
            public readonly TimeUnit timeUnit;
            public readonly DensityUnit densityUnit;
            public readonly ViscosityUnit viscosityUnit;

            public UnitSystemList(UnitSystem unitSystem, LengthUnit lengthUnit,
            ↪   SpeedUnit speedUnit, TimeUnit timeUnit, DensityUnit densityUnit,
            ↪   ViscosityUnit viscosityUnit)
            {
                this.unitSystem = unitSystem;
                this.lengthUnit = lengthUnit;
                this.speedUnit = speedUnit;
                this.timeUnit = timeUnit;
                this.densityUnit = densityUnit;
                this.viscosityUnit = viscosityUnit;
            }
        }
    }
}
```

Units.cs

```
using System;
using System.Globalization;
using System.Windows;
using System.Windows.Data;

namespace UserInterface.Converters
{
    public class Units : IValueConverter
    {
        private readonly UnitClasses.Unit? unit;

        private UnitClasses.Unit Unit { get => unit ?? new
        ↪  UnitClasses.Dimensionless(); }

        public Units(UnitClasses.Unit? unit)
        {
            this.unit = unit;
        }

        /// <summary>
        /// Converts a non-SI unit to the corresponding SI unit.
        /// </summary>
        /// <param name="value">The value in the converter's non-SI unit.</param>
        /// <returns><paramref name="value"/>, converted to the corresponding SI
        ↪  unit.</returns>
        public object Convert(object value, Type targetType, object parameter,
        ↪  CultureInfo culture)
        {
            if (value is not double nonSIValue)
            {
                return DependencyProperty.UnsetValue;
            }

            return nonSIValue * Unit.ConversionRatio;
        }

        /// <summary>
        /// Converts an SI unit to a specified non-SI unit.
        /// </summary>
        /// <param name="value">The value in the SI unit.</param>
        /// <returns><paramref name="value"/>, converted to the converter's non-SI
        ↪  unit.</returns>
        /// <exception cref="NotImplementedException"></exception>
        public object ConvertBack(object value, Type targetType, object parameter,
        ↪  CultureInfo culture)
        {
            if (value is not double SIValue)
            {
                return DependencyProperty.UnsetValue;
            }

            return SIValue / Unit.ConversionRatio;
        }
    }
}
```

BackendManager.cs

```
#if DEBUG
//#define NO_GPU_BACKEND
#endif

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Diagnostics;
using System.IO;
using System.Threading;
using System.Windows;


namespace UserInterface.HelperClasses
{
    public enum BackendStatus
    {
        /// <summary>
        /// Process created but not yet executing.
        /// </summary>
        NotStarted,

        /// <summary>
        /// Currently executing
        /// </summary>
        Running,

        /// <summary>
        /// Not executing, but in a paused state.
        /// </summary>
        Stopped,

        /// <summary>
        /// Not executing and the process has been destroyed or not yet created.
        /// </summary>
        Closed
    }

    /// <summary>
    /// Handler class for dealing with the backend
    /// </summary>
    public class BackendManager : INotifyPropertyChanged
    {
        private Process? backendProcess;
        private string filePath;
        private PipeManager? pipeManager;
        private int iMax;
        private int jMax;

        private BackendStatus backendStatus;

        private float[][]? fields;
        private FieldType[]? namedFields;

        private float frameTime;
        private float dragCoefficient;
        private Stopwatch frameTimer;

        private ResizableLinearQueue<ParameterChangedEventArgs> parameterSendQueue;
        private ParameterHolder parameterHolder;

        private readonly string pipeName = "NEAFluidDynamicsPipe";
        private readonly bool createNoWindow = true;

        public int FieldLength { get => iMax * jMax; }
        public int IMax { get => iMax; set => iMax = value; }
        public int JMax { get => jMax; set => jMax = value; }

        public float FrameTime
        {
            get => frameTime;
            private set
            {
                frameTime = value;
                PropertyChanged?.Invoke(this, new
                 ↪  PropertyChangedEventArgs(nameof(FrameTime)));
            }
        }

        public float DragCoefficient
        {
            get => dragCoefficient;
            private set
            {
                dragCoefficient = value;
                PropertyChanged?.Invoke(this, new
                 ↪  PropertyChangedEventArgs(nameof(DragCoefficient)));
            }
        }

        public BackendStatus BackendStatus
        {
            get => backendStatus;
            private set
            {
                backendStatus = value;
                PropertyChanged?.Invoke(value, new
                 ↪  PropertyChangedEventArgs(nameof(BackendStatus)));
            }
        }

        public event PropertyChangedEventHandler? PropertyChanged;

        private bool CreateBackend(int iMax = 0, int jMax = 0)
        {
            try
            {
                backendProcess = new Process();
                backendProcess.StartInfo.FileName = filePath;
                backendProcess.StartInfo.ArgumentList.Add(pipeName);
                backendProcess.StartInfo.ArgumentList.Add(iMax.ToString());
                backendProcess.StartInfo.ArgumentList.Add(jMax.ToString());
                backendProcess.StartInfo.CreateNoWindow = createNoWindow;
                backendProcess.Start();
                BackendStatus = BackendStatus.NotStarted;
                return true;
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message, "ERROR!");
                return false;
            }
        }

        private bool PipeHandshake()
        {
            pipeManager = new PipeManager(pipeName);
            pipeManager.WaitForConnection();
            (iMax, jMax) = pipeManager.Handshake();
            return iMax > 0 && jMax > 0; // (0,0) is the error condition
        }

        private bool SendControlByte(byte controlByte)
        {
            return pipeManager.WriteByte(controlByte);
        }

        /// <summary>
        /// Initialises field arrays and constructs a request byte based on the
         ↪  null-ness of the field arguments.
        /// </summary>
        private byte CheckFieldParameters(float[]? horizontalVelocity, float[]?
         ↪  verticalVelocity, float[]? pressure, float[]? streamFunction)
        {
            if (pipeManager == null)
            {
                throw new InvalidOperationException("Cannot get data when pipe has
                 ↪  not been opened");
            }

            int requestedFields = horizontalVelocity == null ? 0 : 1; // Sum up how
             ↪  many fields are not null
            requestedFields += verticalVelocity == null ? 0 : 1;
            requestedFields += pressure == null ? 0 : 1;
            requestedFields += streamFunction == null ? 0 : 1;

            if (requestedFields == 0)
            {
```

```
        throw new InvalidOperationException("No fields have been provided,
        ↪  cannot execute");
    }

    byte requestByte = PipeConstants.Request.CONTREQ;
    fields = new float[requestedFields][]; // A container for references to
    ↪  all the different fields
    int fieldNumber = 0;
    List<FieldType> namedFieldsList = new List<FieldType>();

    if (horizontalVelocity != null)
    {
        if (horizontalVelocity.Length < FieldLength)
        {
            throw new InvalidOperationException("Field array is too small");
        }
        requestByte += PipeConstants.Request.HVEL;
        fields[fieldNumber] = horizontalVelocity;
        namedFieldsList.Add(FieldType.HorizontalVelocity);
        fieldNumber++;
    }
    if (verticalVelocity != null)
    {
        if (verticalVelocity.Length < FieldLength)
        {
            throw new InvalidOperationException("Field array is too small");
        }
        requestByte += PipeConstants.Request.VVEL;
        fields[fieldNumber] = verticalVelocity;
        namedFieldsList.Add(FieldType.VerticalVelocity);
        fieldNumber++;
    }
    if (pressure != null)
    {
        if (pressure.Length < FieldLength)
        {
            throw new InvalidOperationException("Field array is too small");
        }
        requestByte += PipeConstants.Request.PRES;
        fields[fieldNumber] = pressure;
        namedFieldsList.Add(FieldType.Pressure);
        fieldNumber++;
    }
    if (streamFunction != null)
    {
        if (streamFunction.Length < FieldLength)
        {
            throw new InvalidOperationException("Field array is too small");
        }
        requestByte += PipeConstants.Request.STRM;
        fields[fieldNumber] = streamFunction;
        namedFieldsList.Add(FieldType.StreamFunction);
    }
```

```
        namedFields = namedFieldsList.ToArray();
        return requestByte;
    }

    private async void SendParameters()
    {
        while (!parameterSendQueue.IsEmpty)
        {
            ParameterChangedEventArgs args = parameterSendQueue.Dequeue();
            string parameterName = args.PropertyName;
            float parameterValue = args.NewValue;
            byte parameterBits = parameterName switch
            {
                "Width" => PipeConstants.Marker.WIDTH,
                "Height" => PipeConstants.Marker.HEIGHT,
                "TimeStepSafetyFactor" => PipeConstants.Marker.TAU,
                "RelaxationParameter" => PipeConstants.Marker.OMEGA,
                "PressureResidualTolerance" => PipeConstants.Marker.RMAX,
                "PressureMaxIterations" => PipeConstants.Marker.ITERMAX,
                "ReynoldsNumber" => PipeConstants.Marker.REYNOLDS,
                "InflowVelocity" => PipeConstants.Marker.INVEL,
                "SurfaceFriction" => PipeConstants.Marker.CHI,
                "FluidViscosity" => PipeConstants.Marker.MU,
                "FluidDensity" => PipeConstants.Marker.DENSITY,
                _ => 0,
            };

            if (parameterBits == 0) // Error case
            {
                throw new InvalidOperationException("Parameter in queue was not
                 ↪ recognised");
            }

            if (parameterBits == PipeConstants.Marker.ITERMAX) // Itermax is the
            ↪ only parameter that is an integer so needs special treatment
            {
                pipeManager.SendParameter((int)parameterValue, parameterBits);
            }
            else
            {
                pipeManager.SendParameter(parameterValue, parameterBits);
            }
            if (await pipeManager.ReadAsync() != PipeConstants.Status.OK)
            {
                throw new IOException("Backend did not read parameters
                 ↪ correctly");
            }
        }
    }

    private void HandleParameterChanged(object? sender, PropertyChangedEventArgs
    ↪ args)
    {
            parameterSendQueue.Enqueue((ParameterChangedEventArgs)args);
        }

        public BackendManager(ParameterHolder parameterHolder)
        {
            this.parameterHolder = parameterHolder;
            parameterHolder.PropertyChanged += HandleParameterChanged;

            fields = null;
            namedFields = null;

            parameterSendQueue = new();

            frameTimer = new Stopwatch();

#if NO_GPU_BACKEND
            if (File.Exists(".\\CPUBackend.exe"))
            {
                filePath = ".\\CPUBackend.exe"; // Look for CPUBackend in same
                ↪ directory...
            }
            else if (File.Exists("..\\..\\..\\..\\x64\\Debug\\CPUBackend.exe"))
            {
                filePath = "..\\..\\..\\..\\x64\\Debug\\CPUBackend.exe"; // ...then
                ↪ look in debug directory.
            }
            else
            {
                MessageBox.Show("Could not find backend executable. Make sure that
                ↪ CPUBackend.exe exists in the same folder as UserInterface.exe",
                ↪ "ERROR: Could not find backend executable.");
                throw new FileNotFoundException("Backend executable could not be
                ↪ found");
            }
#else // ^^ NO_GPU_BACKEND ^^ / vv !NO_GPU_BACKEND vv
            if (File.Exists(".\\GPUBackend.exe"))
            {
                filePath = ".\\GPUBackend.exe"; // First try to find GPU backend in
                ↪ same directory...
            }
            else if (File.Exists(".\\CPUBackend.exe"))
            {
                filePath = ".\\CPUBackend.exe"; // ...then look for CPU backend in
                ↪ same directory.
            }
            else if (File.Exists("..\\..\\..\\..\\x64\\Debug\\GPUBackend.exe"))
            {
                filePath = "..\\..\\..\\..\\x64\\Debug\\GPUBackend.exe"; // When
                ↪ debugging, backend executables are here. Try GPU backend
                ↪ first...
            }
            else if (File.Exists("..\\..\\..\\..\\x64\\Debug\\CPUBackend.exe"))
            {
                filePath = "..\\..\\..\\..\\x64\\Debug\\CPUBackend.exe"; // ...then
                 ↪  try CPU backend.
            }
            else
            {
                MessageBox.Show("Could not find backend executable. Make sure that
                 ↪  either GPUBackend.exe or CPUBackend.exe exists in the same
                 ↪  folder as UserInterface.exe", "ERROR: Could not find backend
                 ↪  executable.");
                throw new FileNotFoundException("Backend executable could not be
                 ↪  found");
            }
#endif // !NO_GPU_BACKEND

            BackendStatus = BackendStatus.Closed;
        }

        /// <summary>
        /// Method to start and connect to the backend process
        /// </summary>
        /// <returns>Boolean result indicating whether the connection was
        ↪  successful</returns>
        public bool ConnectBackend()
        {
            return CreateBackend() && PipeHandshake(); // Return true only if both
             ↪  were successful. Also doesn't attempt handshake if backend did not
             ↪  start correctly
        }

        public bool ConnectBackend(int iMax, int jMax)
        {
            return CreateBackend(iMax, jMax) && PipeHandshake();
        }

        public async void SendAllParameters()
        {
            pipeManager.SendParameter(parameterHolder.Width.Value,
             ↪  PipeConstants.Marker.WIDTH);
            if (await pipeManager.ReadAsync() != PipeConstants.Status.OK) throw new
             ↪  IOException("Backend did not read parameters correctly");

            pipeManager.SendParameter(parameterHolder.Height.Value,
             ↪  PipeConstants.Marker.HEIGHT);
            if (await pipeManager.ReadAsync() != PipeConstants.Status.OK) throw new
             ↪  IOException("Backend did not read parameters correctly");

            pipeManager.SendParameter(parameterHolder.TimeStepSafetyFactor.Value,
             ↪  PipeConstants.Marker.TAU);
            if (await pipeManager.ReadAsync() != PipeConstants.Status.OK) throw new
             ↪  IOException("Backend did not read parameters correctly");

            pipeManager.SendParameter(parameterHolder.RelaxationParameter.Value,
             ↪  PipeConstants.Marker.OMEGA);
            if (await pipeManager.ReadAsync() != PipeConstants.Status.OK) throw new
             ↪  IOException("Backend did not read parameters correctly");


                ↪  pipeManager.SendParameter(parameterHolder.PressureResidualTolerance.Value,
                ↪  PipeConstants.Marker.RMAX);
            if (await pipeManager.ReadAsync() != PipeConstants.Status.OK) throw new
             ↪  IOException("Backend did not read parameters correctly");


                ↪  pipeManager.SendParameter((int)parameterHolder.PressureMaxIterations.Value,
                ↪  PipeConstants.Marker.ITERMAX);
            if (await pipeManager.ReadAsync() != PipeConstants.Status.OK) throw new
             ↪  IOException("Backend did not read parameters correctly");

            pipeManager.SendParameter(parameterHolder.ReynoldsNumber.Value,
             ↪  PipeConstants.Marker.REYNOLDS);
            if (await pipeManager.ReadAsync() != PipeConstants.Status.OK) throw new
             ↪  IOException("Backend did not read parameters correctly");

            pipeManager.SendParameter(parameterHolder.InflowVelocity.Value,
             ↪  PipeConstants.Marker.INVEL);
            if (await pipeManager.ReadAsync() != PipeConstants.Status.OK) throw new
             ↪  IOException("Backend did not read parameters correctly");

            pipeManager.SendParameter(parameterHolder.SurfaceFriction.Value,
             ↪  PipeConstants.Marker.CHI);
            if (await pipeManager.ReadAsync() != PipeConstants.Status.OK) throw new
             ↪  IOException("Backend did not read parameters correctly");

            pipeManager.SendParameter(parameterHolder.FluidDensity.Value,
             ↪  PipeConstants.Marker.DENSITY);
            if (await pipeManager.ReadAsync() != PipeConstants.Status.OK) throw new
             ↪  IOException("Backend did not read parameters correctly");

            pipeManager.SendParameter(parameterHolder.FluidViscosity.Value,
             ↪  PipeConstants.Marker.MU);
            if (await pipeManager.ReadAsync() != PipeConstants.Status.OK) throw new
             ↪  IOException("Backend did not read parameters correctly");
        }

        private async void ReceiveParameter(byte parameterBits)
        {
            if (parameterBits == PipeConstants.Marker.DRAGCOEF)
            {
                DragCoefficient = await pipeManager.ReadParameterAsync();
            }
        }

        /// <summary>
        /// Asynchronous method to repeatedly receive fields from the backend, for
         ↪  visualisation
        /// </summary>
```

```
/// <param name="horizontalVelocity">Array to store horizontal velocity
↪   data</param>
/// <param name="verticalVelocity">Array to store vertical velocity
↪   data</param>
/// <param name="pressure">Array to store pressure data</param>
/// <param name="streamFunction">Array to store stream function
↪   data</param>
/// <param name="token">A cancellation token to stop the method and
↪   backend</param>
/// <exception cref="InvalidOperationException">Thrown when parameters are
↪   invalid</exception>
/// <exception cref="IOException">Thrown when backend does not respond as
↪   expected</exception>
public async void GetFieldStreamsAsync(float[]? horizontalVelocity, float[]?
↪   verticalVelocity, float[]? pressure, float[]? streamFunction,
↪   CancellationToken token)
{
    switch (BackendStatus)
    {
        case BackendStatus.NotStarted:
            byte requestByte = CheckFieldParameters(horizontalVelocity,
            ↪   verticalVelocity, pressure, streamFunction); // Abstract the
            ↪   parameter checking into its own function

            SendParameters(); // Send the parameters that were set before
            ↪   the simulation started

            SendControlByte(requestByte); // Start the backend executing
            byte receivedByte = await pipeManager.ReadAsync();
            if (receivedByte != PipeConstants.Status.OK) // Should receive
            ↪   OK, then the backend will start executing
            {
                if ((receivedByte & PipeConstants.CATEGORYMASK) ==
                ↪   PipeConstants.Error.GENERIC) // Throw an exception with
                ↪   the provided error code
                {
                    throw new IOException($"Backend did not receive data
                    ↪   correctly. Exception code {receivedByte}.");
                }
                throw new IOException("Result from backend not understood");
                ↪   // Throw a generic error if it was not understood at
                ↪   all
            }

            break;

        case BackendStatus.Stopped: // Resuming from a paused state
            if (parameterSendQueue.IsEmpty)
            {
                SendControlByte(PipeConstants.Status.OK);
            }
            else
            {
                SendParameters();
                SendControlByte(PipeConstants.Status.OK);
            }
            break;
        case BackendStatus.Closed:
            throw new IOException("Backend must be created and connected
            ↪  before calling GetFieldStreamsAsync.");
        default:
            break;
    }

    byte[] tmpByteBuffer = new byte[FieldLength * sizeof(float)]; //
    ↪  Temporary buffer for pipe output

    frameTimer.Start(); // Start the timer and create a variable to hold the
    ↪  previous time.
    TimeSpan iterationStartTime = frameTimer.Elapsed;

    bool cancellationRequested = token.IsCancellationRequested;
    BackendStatus = BackendStatus.Running;

    while (!cancellationRequested) // Repeat until the task is cancelled
    {
        if (await pipeManager.ReadAsync() != PipeConstants.Marker.ITERSTART)
        ↪  throw new IOException("Backend did not send data correctly"); //
        ↪  Each timestep iteration should start with an ITERSTART

        for (int fieldNum = 0; fieldNum < fields.Length; fieldNum++)
        {
            byte fieldBits = (byte)namedFields[fieldNum];
            byte startMarker = await pipeManager.ReadAsync();
            if (startMarker != (PipeConstants.Marker.FLDSTART | fieldBits))
            ↪  throw new IOException($"Backend did not send data correctly.
            ↪  Bits were {startMarker}"); // Each field should start with a
            ↪  FLDSTART with the relevant field bits

            await pipeManager.ReadAsync(tmpByteBuffer, FieldLength *
            ↪  sizeof(float)); // Read the stream of bytes into the
            ↪  temporary buffer
            Buffer.BlockCopy(tmpByteBuffer, 0, fields[fieldNum], 0,
            ↪  FieldLength * sizeof(float)); // Copy the bytes from the
            ↪  temporary buffer into the double array
            if (await pipeManager.ReadAsync() !=
            ↪  (PipeConstants.Marker.FLDEND | fieldBits))  throw new
            ↪  IOException("Backend did not send data correctly"); // Each
            ↪  field should start with a FLDEND with the relevant field
            ↪  bits
        }
        byte nextByte = await pipeManager.ReadAsync();
        while ((nextByte & ~PipeConstants.Marker.PRMMASK) ==
        ↪  PipeConstants.Marker.PRMSTART)
        {
            byte parameterBits = (byte)(nextByte &
            ↪  PipeConstants.Marker.PRMMASK);
```

```
                ReceiveParameter(parameterBits);
                if (await pipeManager.ReadAsync() !=
                ↪   (PipeConstants.Marker.PRMEND | parameterBits)) throw new
                ↪   IOException("Backend did not send data correctly");
                nextByte = await pipeManager.ReadAsync();
            }

            if (nextByte != PipeConstants.Marker.ITEREND) throw new
            ↪   IOException("Backend did not send data correctly"); // Each
            ↪   timestep iteration should end with an ITEREND

            if (token.IsCancellationRequested)
            {
                cancellationRequested = true;
            }
            else if (parameterSendQueue.IsEmpty)
            {
                SendControlByte(PipeConstants.Status.OK);
            }
            else
            {
                SendParameters();
                SendControlByte(PipeConstants.Status.OK);
            }
            TimeSpan iterationLength = frameTimer.Elapsed - iterationStartTime;
            FrameTime = (float)iterationLength.TotalSeconds;

            iterationStartTime = frameTimer.Elapsed; // Set the new iteration
            ↪   start time once FPS processing is done.
        }

        SendControlByte(PipeConstants.Status.STOP); // Upon cancellation, stop
        ↪   (pause) the backend.
        BackendStatus = BackendStatus.Stopped;

        if (await pipeManager.ReadAsync() != PipeConstants.Status.OK) throw new
        ↪   IOException("Backend did not stop correctly");
    }

    public bool SendObstacles(bool[] obstacles)
    {
        return pipeManager.SendObstacles(obstacles);
    }

    public bool CloseBackend()
    {
        SendControlByte(PipeConstants.Status.CLOSE);
        if (pipeManager.AttemptRead().data[0] != PipeConstants.Status.OK)
        {
            return false;
        }
        SendControlByte(PipeConstants.Status.CLOSE);
        if (pipeManager.AttemptRead().data[0] != PipeConstants.Status.OK)
```

```
            {
                return false;
            }
            if (!backendProcess.HasExited)
            {
                return false;
            }
            backendProcess.Close();
            pipeManager.CloseConnection();
            return true;
        }

        public void ForceCloseBackend()
        {
            backendProcess.Kill();
            pipeManager.CloseConnection();
        }
    }
}
```

CircularQueue.cs

```
using System;

namespace UserInterface.HelperClasses
{
    public class CircularQueue<T> : Queue<T>
    {
        protected int count;

        public CircularQueue(int length) : base(length)
        {
            count = 0;
        }

        public override void Enqueue(T value)
        {
            if (IsFull)
            {
                throw new InvalidOperationException("Queue was full when Enqueue was
                ↪  called");
            }

            array[back] = value;
            back++;
            if (back >= length)
            {
                back = 0;
            }
            count++;
        }

        public override T Dequeue()
        {
            if (IsEmpty)
            {
                throw new InvalidOperationException("Queue was empty when Dequeue
                ↪   was called");
            }

            T removedItem = array[front];
            front++;
            if (front >= length)
            {
                front = 0;
            }
            count--;

            return removedItem;
        }

        public override bool IsEmpty
        {
            get { return count == 0; }
        }

        public override bool IsFull
        {
            get { return count == length; }
        }

        public override int Count => count;
    }
}
```

Commands.cs

```
using Microsoft.Win32;
using System;
using System.ComponentModel;
using System.Diagnostics;
using System.Threading;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Input;
using UserInterface.ViewModels;
using UserInterface.Views;

namespace UserInterface.HelperClasses
{
    public class Commands
    {
        /// <summary>
        /// Base class for commands that are related to a specific ViewModel using
        ↪   dependency injection. Abstractly implements <see cref="ICommand"/>.
        /// </summary>
        /// <typeparam name="VMType">The type of the ViewModel that will be used
        ↪   with the Command.</typeparam>
```

```
public abstract class VMCommandBase<VMType> : ICommand
{
    protected VMType parentViewModel;

    public event EventHandler? CanExecuteChanged;

    public virtual void OnCanExecuteChanged(object? sender, EventArgs e)
    {
        CanExecuteChanged?.Invoke(sender, e);
    }

    public virtual bool CanExecute(object? parameter) { return true; }

    public abstract void Execute(object? parameter);

    public VMCommandBase(VMType parentViewModel)
    {
        this.parentViewModel = parentViewModel;
    }
}

/// <summary>
/// Base class for commands that deal with parameters, again using
 ↪  dependency injection to get the <see cref="ParameterHolder" />.
/// </summary>
/// <typeparam name="VMType">The type fo the ViewModel that will be used
 ↪  with the Command.</typeparam>
public abstract class ParameterCommandBase<VMType> : VMCommandBase<VMType>
{
    protected ParameterHolder parameterHolder;

    public ParameterCommandBase(VMType parentViewModel, ParameterHolder
     ↪  parameterHolder) : base(parentViewModel)
    {
        this.parameterHolder = parameterHolder;
    }
}

public class SetAirParameters : VMCommandBase<ConfigScreenVM>
{
    public override void Execute(object? parameter)
    {
        parentViewModel.ReynoldsNo = DefaultParameters.REYNOLDS_NUMBER;
        parentViewModel.Viscosity = DefaultParameters.FLUID_VISCOSITY;
        parentViewModel.Density = DefaultParameters.FLUID_DENSITY;
    }

    public SetAirParameters(ConfigScreenVM parentViewModel) :
     ↪  base(parentViewModel) { }
}

public class AdvancedParametersReset :
 ↪  ParameterCommandBase<AdvancedParametersVM>
```

```
{
    public override void Execute(object? parameter)
    {
        parameterHolder.TimeStepSafetyFactor.Reset();
        parentViewModel.Tau =
         ↪  parameterHolder.TimeStepSafetyFactor.DefaultValue;

        parameterHolder.RelaxationParameter.Reset();
        parentViewModel.Omega =
         ↪  parameterHolder.RelaxationParameter.DefaultValue;

        parameterHolder.PressureResidualTolerance.Reset();
        parentViewModel.RMax =
         ↪  parameterHolder.PressureResidualTolerance.DefaultValue;

        parameterHolder.PressureMaxIterations.Reset();
        parentViewModel.IterMax =
         ↪  parameterHolder.PressureMaxIterations.DefaultValue;
    }

    public AdvancedParametersReset(AdvancedParametersVM parentViewModel,
     ↪  ParameterHolder parameterHolder) : base(parentViewModel,
     ↪  parameterHolder) { }
}

public class ConfigScreenReset : ParameterCommandBase<ConfigScreenVM>
{
    public override void Execute(object? parameter)
    {
        parameterHolder.InflowVelocity.Reset();
        parentViewModel.InVel = parameterHolder.InflowVelocity.DefaultValue;

        parameterHolder.SurfaceFriction.Reset();
        parentViewModel.Chi = parameterHolder.SurfaceFriction.DefaultValue;

        parameterHolder.Width.Reset();
        parentViewModel.Width = parameterHolder.Width.DefaultValue;

        parameterHolder.Height.Reset();
        parentViewModel.Height = parameterHolder.Height.DefaultValue;
    }

    public ConfigScreenReset(ConfigScreenVM parentViewModel, ParameterHolder
     ↪  parameterHolder) : base(parentViewModel, parameterHolder) { }
}

public class SaveParameters : ParameterCommandBase<AdvancedParametersVM>
{
    private ParameterStruct<T> ModifyParameterValue<T>(ParameterStruct<T>
     ↪  parameterStruct, T newValue)
    {
        parameterStruct.Value = newValue;
        return parameterStruct;
        }

        public override void Execute(object? parameter)
        {
            parameterHolder.TimeStepSafetyFactor =
             ↪  ModifyParameterValue(parameterHolder.TimeStepSafetyFactor,
             ↪  parentViewModel.Tau);
            parameterHolder.RelaxationParameter =
             ↪  ModifyParameterValue(parameterHolder.RelaxationParameter,
             ↪  parentViewModel.Omega);
            parameterHolder.PressureResidualTolerance =
             ↪  ModifyParameterValue(parameterHolder.PressureResidualTolerance,
             ↪  parentViewModel.RMax);
            parameterHolder.PressureMaxIterations =
             ↪  ModifyParameterValue(parameterHolder.PressureMaxIterations,
             ↪  parentViewModel.IterMax);

            App.RaisePopupDeleted(this, new EventArgs());
        }

        public SaveParameters(AdvancedParametersVM parentViewModel,
         ↪  ParameterHolder parameterHolder, ChangeWindow changeWindowCommand) :
         ↪  base(parentViewModel, parameterHolder) { }
    }

    public class SwitchPanel : VMCommandBase<SimulationScreenVM>
    {
        public override void Execute(object? parameter)
        {
            string name = ((FrameworkElement)parameter).Name;
            if (name == parentViewModel.CurrentButton) // If the button of the
             ↪  currently open panel is clicked, set the current button to null
             ↪  to close all panels (toggle functionality).
            {
                parentViewModel.CurrentButton = null;
            }
            else
            {
                parentViewModel.CurrentButton = name; // If any other panel is
                 ↪  open, or no panel is open, open the one corresponding to the
                 ↪  button.
            }
        }

        public SwitchPanel(SimulationScreenVM parentViewModel) :
         ↪  base(parentViewModel) { }
    }

    public class ChangeWindow : ICommand
    {
        public event EventHandler? CanExecuteChanged;

        public bool CanExecute(object? parameter) { return true; } // Unless app
         ↪  logic changes, this command can always execute.
        public void Execute(object? parameter)
        {
            if (parameter == null) { return; }
            App.RaiseUserControlChanged(this, new
            ↪  UserControlChangeEventArgs((Type)parameter));
        }
    }

    public class CreatePopup : ICommand
    {
        public event EventHandler? CanExecuteChanged;

        public bool CanExecute(object? parameter) { return true; }

        public void Execute(object? parameter)
        {
            if (parameter == null) return;
            App.RaisePopupCreated(this, new
            ↪  UserControlChangeEventArgs((Type)parameter));
        }
    }

    public class PauseResumeBackend : VMCommandBase<SimulationScreenVM>
    {
        public override bool CanExecute(object? parameter)
        {
            return !parentViewModel.EditingObstacles; // Cannot execute when
            ↪  editing obstacles.
        }

        public override void Execute(object? parameter)
        {
            switch (parentViewModel.BackendStatus)
            {
                case BackendStatus.Running:
                    parentViewModel.BackendCTS.Cancel(); // Pause the backend.
                    break;
                case BackendStatus.Stopped:
                    Task.Run(parentViewModel.StartComputation); // Resume the
                    ↪  backend computation.
                    break;
                default:
                    break;
            }
        }
        public PauseResumeBackend(SimulationScreenVM parentViewModel) :
        ↪  base(parentViewModel)
        {
            parentViewModel.PropertyChanged += VMPropertyChanged;
        }

        private void VMPropertyChanged(object? sender, PropertyChangedEventArgs
        ↪  e)
        {
            if (e.PropertyName == nameof(parentViewModel.EditingObstacles))
            {
                OnCanExecuteChanged(sender, e);
            }
        }
    }

    public class EditObstacles : VMCommandBase<SimulationScreenVM>
    {
        PauseResumeBackend BackendCommand;

        public override bool CanExecute(object? parameter)
        {
            return !parentViewModel.ObstacleHolder.UsingObstacleFile;
        }

        public override void Execute(object? parameter)
        {
            if (parentViewModel.EditingObstacles) // Obstacle editing is
             ↪  finished, need to embed obstacles and start backend executing.
            {
                parentViewModel.EditingObstacles = false;
                parentViewModel.EmbedObstacles();
                BackendCommand.Execute(null);
            }
            else // Obstacle editing has started, need to stop backend and allow
             ↪  obstacles to be edited.
            {
                BackendCommand.Execute(null);
                parentViewModel.EditingObstacles = true;
            }
        }
        public EditObstacles(SimulationScreenVM parentViewModel) :
         ↪  base(parentViewModel)
        {
            BackendCommand = new PauseResumeBackend(parentViewModel);
        }
    }

    public class SelectObstacleFile : VMCommandBase<ConfigScreenVM>
    {
        public override bool CanExecute(object? parameter)
        {
            return parentViewModel.UsingObstacleFile;
        }
        public override void Execute(object? parameter)
        {
            OpenFileDialog openDialog = new();
            openDialog.Title = "Open Obstacle File";
            openDialog.Filter = "Obstacle Files (*.simobst)|*.simobst|Binary
             ↪  Files (*.bin)|*.bin|All Files|*.*";
            if (openDialog.ShowDialog() == true) // OK pressed rather than
             ↪  cancel
            {
                parentViewModel.FileName = openDialog.FileName;
            }
        }

        public SelectObstacleFile(ConfigScreenVM parentViewModel) :
        ↪   base(parentViewModel)
        {
            parentViewModel.PropertyChanged += VMPropertyChanged;
        }

        private void VMPropertyChanged(object? sender, PropertyChangedEventArgs
        ↪   e)
        {
            if (e.PropertyName == nameof(parentViewModel.UsingObstacleFile))
            {
                OnCanExecuteChanged(sender, e);
            }
        }
    }

    public class TrySimulate : VMCommandBase<ConfigScreenVM>
    {
        private ChangeWindow ChangeWindowCommand;
        public override void Execute(object? parameter)
        {
            if (parentViewModel.UsingObstacleFile && parentViewModel.FileName is
            ↪   null)
            {
                MessageBox.Show("You must select a file with obstacles, or
                ↪   deselect the checkbox.", "ERROR: No file selected");
            }
            else
            {
                ChangeWindowCommand.Execute(typeof(SimulationScreen));
            }
        }

        public TrySimulate(ConfigScreenVM parentViewModel) :
        ↪   base(parentViewModel)
        {
            ChangeWindowCommand = new();
        }
    }

    public class SimScreenBack : VMCommandBase<SimulationScreenVM>
    {
        private readonly PauseResumeBackend PauseResumeBackendCommand;
        private readonly ChangeWindow ChangeWindowCommand;
        public override bool CanExecute(object? parameter)
        {
            return !parentViewModel.EditingObstacles; // Cannot execute when
            ↪   editing obstacles.
            }

            public override void Execute(object? parameter)
            {
                if (parentViewModel.BackendStatus == BackendStatus.Running)
                {
                    PauseResumeBackendCommand.Execute(null);
                }
                Thread.Sleep(100);
                parentViewModel.CloseBackend();
                ChangeWindowCommand.Execute(typeof(ConfigScreen));
            }

            public SimScreenBack(SimulationScreenVM parentViewModel) :
             ↪  base(parentViewModel)
            {
                PauseResumeBackendCommand = new PauseResumeBackend(parentViewModel);
                ChangeWindowCommand = new ChangeWindow();
                parentViewModel.PropertyChanged += VMPropertyChanged;
            }

            private void VMPropertyChanged(object? sender, PropertyChangedEventArgs
             ↪  e)
            {
                if (e.PropertyName == nameof(parentViewModel.EditingObstacles))
                 ↪  OnCanExecuteChanged(sender, e);
            }
        }
    }
}
```

DefaultParameters.cs

```
namespace UserInterface.HelperClasses
{
    public static class DefaultParameters
    {
        public static readonly float WIDTH = 1f;
        public static readonly float HEIGHT = 1f;
        public static readonly float TIMESTEP_SAFETY_FACTOR = 0.8f;
        public static readonly float RELAXATION_PARAMETER = 1.7f;
        public static readonly float PRESSURE_RESIDUAL_TOLERANCE = 2f;
        public static readonly float PRESSURE_MAX_ITERATIONS = 1000f;
        public static readonly float REYNOLDS_NUMBER = 2000f;
        public static readonly float FLUID_VISCOSITY = 1.983E-5f;
        public static readonly float FLUID_VELOCITY = 5f;
        public static readonly float FLUID_DENSITY = 1.293f;
        public static readonly float SURFACE_FRICTION = 0f;

        public static readonly bool DRAW_CONTOURS = true;
        public static readonly float CONTOUR_TOLERANCE = 0.01f;
        public static readonly float NUM_CONTOURS = 15f;
        public static readonly int FPS_WINDOW_SIZE = 500;
        public static readonly int DRAG_COEF_WINDOW_SIZE = 10;
        public static readonly float VELOCITY_MIN = 0f;
        public static readonly float VELOCITY_MAX = 18f;
        public static readonly float PRESSURE_MIN = 1000f;
        public static readonly float PRESSURE_MAX = 100000f;
    }
}
```

MovingAverage.cs

```
using System;
using System.Numerics;
namespace UserInterface.HelperClasses
{
    public class MovingAverage<T> where T : INumber<T>
    {
        private CircularQueue<T> dataPoints;
        private readonly int windowSize;

        private T currentSum = default; // Contains the sum of all the current data
        ↪  points

        public T Average { get; private set; } = default;

        public MovingAverage(int windowSize)
        {
            this.windowSize = windowSize;
            dataPoints = new CircularQueue<T>(windowSize);
        }

        public void UpdateAverage(T newValue)
        {
            if (dataPoints.Count == windowSize)
            {
                currentSum -= dataPoints.Dequeue(); // Take the first item off the
                ↪  sum (discarding it)
            }

            currentSum += newValue;
            dataPoints.Enqueue(newValue);

            Average = currentSum / (T)Convert.ChangeType(dataPoints.Count,
            ↪  typeof(T)); // Divide the current sum by the number of data points.
            ↪  Conversion between int and generic T had to use ChangeType
        }
    }
}
```

ObstacleHolder.cs

```
using System.IO;

namespace UserInterface.HelperClasses
{
```

```
public class ObstacleHolder
{
    private string? fileName;
    private bool usingObstacleFile;

    private int dataWidth;
    private int dataHeight;

    private bool[]? obstacleData;

    public string? FileName { get => fileName; set => fileName = value; }
    public bool UsingObstacleFile { get => usingObstacleFile; set =>
    ↪  usingObstacleFile = value; }
    public int DataWidth { get => dataWidth; set => dataWidth = value; }
    public int DataHeight { get => dataHeight; set => dataHeight = value; }
    public bool[]? ObstacleData { get => obstacleData; set => obstacleData =
    ↪  value; }

    public ObstacleHolder(string? fileName, bool usingObstacleFile)
    {
        this.fileName = fileName;
        this.usingObstacleFile = usingObstacleFile;
        dataWidth = 128;
        dataHeight = 256;
    }

    /// <summary>
    /// Reads in an obstacle file, sets the data width and height, and returns
    ↪  the contents, formatted into a boolean array.
    /// </summary>
    /// <returns>The contents of the file, formatted into a flattened boolean
    ↪  array.</returns>
    /// <exception cref="FileNotFoundException">Thrown when the file is not
    ↪  found or, more likely, the file cannot be accessed with current
    ↪  permissions.</exception>
    /// <exception cref="FileFormatException">Thrown when the file is not in the
    ↪  correct format.</exception>
    public void ReadObstacleFile()
    {
        if (!File.Exists(fileName))
        {
            throw new FileNotFoundException("Specified file was not found. Check
            ↪  the correct permissions exist to access it, and that it has not
            ↪  been deleted.");
        }

        byte[] obstacleBuffer;
        using (Stream stream = File.OpenRead(fileName))
        {
            using BinaryReader reader = new BinaryReader(stream);
            try
            {
                dataWidth = reader.ReadInt32(); // Read iMax and jMax
                dataHeight = reader.ReadInt32();
                int fieldLength = (dataWidth + 2) * (dataHeight + 2);
                int obstacleDataLength = fieldLength / 8 + (fieldLength % 8 == 0
                 ↪  ? 0 : 1); // Field length divided by 8 plus an extra byte
                 ↪  for any remaining bits
                obstacleBuffer = reader.ReadBytes(obstacleDataLength);
            }
            catch (IOException e)
            {
                string exceptionMessage = e.Message.Length > 0 ? e.Message :
                 ↪  "[no internal error message]";
                throw new FileFormatException($"Input file was malformed, and an
                 ↪  error occurred while parsing: {exceptionMessage}.");
            }
        }

        obstacleData = new bool[(dataWidth + 2) * (dataHeight + 2)];
        int byteNumber = 0;
        for (int i = 0; i < obstacleData.Length; i++)
        {
            obstacleData[byteNumber * 8 + (i % 8)] =
             ↪  ((obstacleBuffer[byteNumber] >> (i % 8)) & 1) != 0; // Due to
             ↪  the way bits are shifted into the bytes in the file, they must
             ↪  be shifted off in the opposite order hence the complicated
             ↪  expression for obstacles[...]. Right shift and AND with 1 takes
             ↪  that bit only

            if (i % 8 == 7)
            {
                byteNumber++;
            }
        }
    }
  }
}
```

ParameterChangedEventArgs.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace UserInterface.HelperClasses
{
    class ParameterChangedEventArgs : PropertyChangedEventArgs
    {
        public float NewValue { get; private set; }

        public ParameterChangedEventArgs(string propertyName, float newValue) :
         ↪  base(propertyName)
        {
            NewValue = newValue;
        }
    }
}
```

ParameterHolder.cs

```
using System;
using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace UserInterface.HelperClasses
{
    public enum ParameterUsage
    {
        Backend,
        Visualisation
    }


    public struct ParameterStruct<T>
    {
        /// <summary>
        /// Initialises a parameter struct with a default value separate to its
        ↪ initial value.
        /// </summary>
        /// <param name="defaultValue">The default value for the parameter.</param>
        /// <param name="value">The initial value for the paramter.</param>
        /// <param name="usage">Where in the program the parameter is used.</param>
        /// <param name="canChangeWhileRunning">A <c>bool</c> to indicate whether
        ↪ the parameter can change while the simulation is running.</param>
        public ParameterStruct(T defaultValue, T value, ParameterUsage usage, bool
        ↪ canChangeWhileRunning)
        {
            DefaultValue = defaultValue;
            Value = value;
            Usage = usage;
            CanChangeWhileRunning = canChangeWhileRunning;
        }

        /// <summary>
        /// Initialises a parameter struct with its default value.
        /// </summary>
        /// <param name="value">The default value for the parameter, to be used as
        ↪ its initial value also.</param>
        /// <param name="usage">Where in the program the parameter is used.</param>
        /// <param name="canChangeWhileRunning">A <c>bool</c> to indicate whether
        ↪ the parameter can change while the simulation is running.</param>
        public ParameterStruct(T value, ParameterUsage usage, bool
        ↪ canChangeWhileRunning)
        {
            DefaultValue = value;
            Value = DefaultValue;
            Usage = usage;
            CanChangeWhileRunning = canChangeWhileRunning;
        }

        public T DefaultValue { get; }
        public T Value { get; set; }
        public ParameterUsage Usage { get; }
        public bool CanChangeWhileRunning { get; }

        public void Reset()
        {
            Value = DefaultValue;
        }
    }

    public struct FieldParameters
    {
        public float[] field;
        public float min;
        public float max;
    }

    public class ParameterHolder : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler? PropertyChanged;

        // Backend parameters
        private ParameterStruct<float> width;
        private ParameterStruct<float> height;
        private ParameterStruct<float> timeStepSafetyFactor;
        private ParameterStruct<float> relaxationParameter;
        private ParameterStruct<float> pressureResidualTolerance;
        private ParameterStruct<float> pressureMaxIterations;
        private ParameterStruct<float> reynoldsNumber;
        private ParameterStruct<float> fluidViscosity;
        private ParameterStruct<float> fluidVelocity;
        private ParameterStruct<float> fluidDensity;
        private ParameterStruct<float> surfaceFriction;

        // Visualisation parameters
        private ParameterStruct<FieldParameters> fieldParameters;
        private ParameterStruct<bool> drawContours;
        private ParameterStruct<float> contourTolerance;
        private ParameterStruct<float> numContours;

        #region Properties
        public ParameterStruct<float> Width
        {
            get => width;

            set
            {
                width = value;
                OnPropertyChanged(width.Value);
            }
        }
        public ParameterStruct<float> Height
        {
            get => height;

            set
            {
                height = value;
                OnPropertyChanged(height.Value);
            }
        }
        public ParameterStruct<float> TimeStepSafetyFactor
        {
            get => timeStepSafetyFactor;

            set
            {
                timeStepSafetyFactor = value;
                OnPropertyChanged(TimeStepSafetyFactor.Value);
            }
        }
        public ParameterStruct<float> RelaxationParameter
        {
            get => relaxationParameter;

            set
            {
                relaxationParameter = value;
                OnPropertyChanged(relaxationParameter.Value);
            }
        }
        public ParameterStruct<float> PressureResidualTolerance
        {
            get => pressureResidualTolerance;

            set
            {
                pressureResidualTolerance = value;
                OnPropertyChanged(pressureResidualTolerance.Value);
            }
        }
        public ParameterStruct<float> PressureMaxIterations
        {
            get => pressureMaxIterations;

            set
            {
                pressureMaxIterations = value;
                OnPropertyChanged(pressureMaxIterations.Value);
            }
        }
        public ParameterStruct<float> ReynoldsNumber
```

```
        {
            get => reynoldsNumber;

            set
            {
                reynoldsNumber = value;
                OnPropertyChanged(reynoldsNumber.Value);
            }
        }

        public ParameterStruct<float> FluidViscosity
        {
            get => fluidViscosity;
            set
            {
                fluidViscosity = value;
                OnPropertyChanged(fluidViscosity.Value);
            }
        }

        public ParameterStruct<float> InflowVelocity
        {
            get => fluidVelocity;

            set
            {
                fluidVelocity = value;
                OnPropertyChanged(fluidVelocity.Value);
            }
        }

        public ParameterStruct<float> FluidDensity
        {
            get => fluidDensity;
            set
            {
                fluidDensity = value;
                OnPropertyChanged(FluidDensity.Value);
            }
        }

        public ParameterStruct<float> SurfaceFriction
        {
            get => surfaceFriction;

            set
            {
                surfaceFriction = value;
                OnPropertyChanged(surfaceFriction.Value);
            }
        }
        public ParameterStruct<FieldParameters> FieldParameters
        {
```

```
        get => fieldParameters;

        set
        {
            fieldParameters = value;
        }
    }
    public ParameterStruct<float> ContourTolerance
    {
        get => contourTolerance;

        set
        {
            contourTolerance = value;
        }
    }
    public ParameterStruct<float> NumContours
    {
        get => numContours;

        set
        {
            numContours = value;
        }
    }
    public ParameterStruct<bool> DrawContours
    {
        get => drawContours;

        set
        {
            drawContours = value;
        }
    }
    #endregion

    public ParameterHolder(float width, float height, float
    ↪   timeStepSafetyFactor, float relaxationParameter, float
    ↪   pressureResidualTolerance, float pressureMaxIterations, float
    ↪   reynoldsNumber, float fluidViscosity, float fluidVelocity, float
    ↪   fluidDensity, float surfaceFriction, FieldParameters fieldParameters,
    ↪   bool drawContours, float contourTolerance, float numContours)
    {

        this.width = new ParameterStruct<float>(width, ParameterUsage.Backend,
         ↪  false);
        this.height = new ParameterStruct<float>(height, ParameterUsage.Backend,
         ↪  false);
        this.timeStepSafetyFactor = new
         ↪  ParameterStruct<float>(timeStepSafetyFactor, ParameterUsage.Backend,
         ↪  true);
        this.relaxationParameter = new
         ↪  ParameterStruct<float>(relaxationParameter, ParameterUsage.Backend,
         ↪  false);
```

```
            this.pressureResidualTolerance = new
             ↪  ParameterStruct<float>(pressureResidualTolerance,
             ↪  ParameterUsage.Backend, true);
            this.pressureMaxIterations = new
             ↪  ParameterStruct<float>(pressureMaxIterations,
             ↪  ParameterUsage.Backend, true);
            this.reynoldsNumber = new ParameterStruct<float>(reynoldsNumber,
             ↪  ParameterUsage.Backend, false);
            this.fluidViscosity = new ParameterStruct<float>(fluidViscosity,
             ↪  ParameterUsage.Backend, false);
            this.fluidVelocity = new ParameterStruct<float>(fluidVelocity,
             ↪  ParameterUsage.Backend, true);
            this.fluidDensity = new ParameterStruct<float>(fluidDensity,
             ↪  ParameterUsage.Backend, false);
            this.surfaceFriction = new ParameterStruct<float>(surfaceFriction,
             ↪  ParameterUsage.Backend, true);
            this.fieldParameters = new
             ↪  ParameterStruct<FieldParameters>(fieldParameters,
             ↪  ParameterUsage.Visualisation, true);
            this.drawContours = new ParameterStruct<bool>(drawContours,
             ↪  ParameterUsage.Visualisation, true);
            this.contourTolerance = new ParameterStruct<float>(contourTolerance,
             ↪  ParameterUsage.Visualisation, true);
            this.numContours = new ParameterStruct<float>(numContours,
             ↪  ParameterUsage.Visualisation, true);
        }

        private void OnPropertyChanged(float value, [CallerMemberName] string name =
         ↪  "")
        {
            PropertyChanged?.Invoke(this, new ParameterChangedEventArgs(name,
             ↪  value));
        }

        public void ReadParameters(string fileName)
        {
            throw new NotImplementedException("ReadParameters not yet implemented");
        }
    }
}
```

PipeConstants.cs

```
namespace UserInterface.HelperClasses
{
    /// <summary>
    /// Constants for pipe communication, containing all the control bytes as
     ↪  defined in Documentation D.3 Precise Specification
    /// </summary>
    internal static class PipeConstants
    {
        public static readonly byte NULL = 0;
        public static readonly byte CATEGORYMASK = 0b11000000;
```

```
/// <summary>
/// STATUS bytes, providing information to the client or commands to do with
↪   program state
/// </summary>
public static class Status
{
    public static readonly byte GENERIC = 0b00000000;
    public static readonly byte HELLO = 0b00001000;
    public static readonly byte BUSY = 0b00010000;
    public static readonly byte OK = 0b00011000;
    public static readonly byte STOP = 0b00100000;
    public static readonly byte CLOSE = 0b00101000;

    public static readonly byte PARAMMASK = 0b00000111;
}

/// <summary>
/// REQUEST bytes, to request data to be calculated and sent by the client
/// </summary>
public static class Request
{
    public static readonly byte GENERIC = 0b01000000;
    public static readonly byte FIXLENREQ = 0b01000000;
    public static readonly byte CONTREQ = 0b01100000;

    public static readonly byte PARAMMASK = 0b00011111;

    public static readonly byte HVEL = 0b00010000;
    public static readonly byte VVEL = 0b00001000;
    public static readonly byte PRES = 0b00000100;
    public static readonly byte STRM = 0b00000010;
}

/// <summary>
/// MARKER bytes, to denote start and end of fields, timestep iterations, or
↪   parameters
/// </summary>
public static class Marker
{
    public static readonly byte GENERIC = 0b10000000;
    public static readonly byte ITERSTART = 0b10000000;
    public static readonly byte ITEREND = 0b10001000;
    public static readonly byte FLDSTART = 0b10010000;
    public static readonly byte FLDEND = 0b10011000;

    public static readonly byte ITERPRMMASK = 0b00000111;

    public static readonly byte HVEL = 0b00000001;
    public static readonly byte VVEL = 0b00000010;
    public static readonly byte PRES = 0b00000011;
    public static readonly byte STRM = 0b00000100;
    public static readonly byte OBST = 0b00000101;
```

```
            public static readonly byte PRMSTART = 0b10100000;
            public static readonly byte PRMEND = 0b10101000;

            public static readonly byte PRMMASK = 0b00001111;

            public static readonly byte IMAX = 0b00000001;
            public static readonly byte JMAX = 0b00000010;
            public static readonly byte WIDTH = 0b00000011;
            public static readonly byte HEIGHT = 0b00000100;
            public static readonly byte TAU = 0b00000101;
            public static readonly byte OMEGA = 0b00000110;
            public static readonly byte RMAX = 0b00000111;
            public static readonly byte ITERMAX = 0b00001000;
            public static readonly byte REYNOLDS = 0b00001001;
            public static readonly byte INVEL = 0b00001010;
            public static readonly byte CHI = 0b00001011;
            public static readonly byte MU = 0b00001100;
            public static readonly byte DENSITY = 0b00001101;
            public static readonly byte DRAGCOEF = 0b00001110;
        }

        /// <summary>
        /// ERROR bytes, sent due to errors in data or internal stop codes
        /// </summary>
        public static class Error
        {
            public static readonly byte GENERIC = 0b11000000;
            public static readonly byte BADREQ = 0b11000001;
            public static readonly byte BADPARAM = 0b11000010;
            public static readonly byte INTERNAL = 0b11000011;
            public static readonly byte TIMEOUT = 0b11000100;
            public static readonly byte BADTYPE = 0b11000101;
            public static readonly byte BADLEN = 0b11000110;
        }
    }
}
```

PipeManager.cs

```
using System;
using System.Diagnostics;
using System.IO.Pipes;
using System.Threading.Tasks;

namespace UserInterface.HelperClasses
{
    /// <summary>
    /// Struct enclosing a bool, specifying whether a read operation happened, and a
    ↪  buffer for the read operation output (if applicable)
    /// </summary>
    public struct ReadResults
    {
        public bool anythingRead;
        public byte[] data;
```

```
}

public enum FieldType
{
    HorizontalVelocity = 1,
    VerticalVelocity = 2,
    Pressure = 3,
    StreamFunction = 4
}

/// <summary>
/// Helper class for managing the pipe communication with the C++ backend
/// </summary>
public class PipeManager
{
    private readonly NamedPipeServerStream pipeStream;

    public PipeManager(string pipeName)
    {
        pipeStream = new NamedPipeServerStream(pipeName);
    }

    /// <summary>
    /// Serialises an integer into part of a buffer.
    /// </summary>
    /// <param name="buffer">The <c>byte[] to store the result in.</c></param>
    /// <param name="offset">The index in which to store the first
    ↪   element.</param>
    /// <param name="datum">The datum to store.</param>
    private static void SerialisePrimitive(byte[] buffer, int offset, int datum)
    {
        for (int i = 0; i < sizeof(int); i++)
        {
            buffer[i + offset] = (byte)(datum >> i * 8);
        }
    }

    /// <summary>
    /// Serialises a float into part of a buffer.
    /// </summary>
    /// <param name="buffer">The <c>byte[] to store the result in.</c></param>
    /// <param name="offset">The index in which to store the first
    ↪   element.</param>
    /// <param name="datum">The datum to store.</param>
    private static void SerialisePrimitive(byte[] buffer, int offset, float
    ↪   datum)
    {
        byte[] serialisedPrimitive = BitConverter.GetBytes(datum);
        Buffer.BlockCopy(serialisedPrimitive, 0, buffer, offset, sizeof(float));
    }

    /// <summary>
    /// Reads one byte asynchronously
    /// </summary>
    /// <returns>A task to read the byte from the pipe, when one is
    ↪ available</returns>
    public Task<byte> ReadAsync()
    {
        TaskCompletionSource<byte> taskCompletionSource = new
        ↪ TaskCompletionSource<byte>();

        byte[] buffer = new byte[1];
        pipeStream.Read(buffer, 0, 1); // Read one byte. ReadByte method is not
            ↪ used because that returns -1 if there is nothing to read, whereas we
            ↪ want to wait until there is data available which Read does

        taskCompletionSource.SetResult(buffer[0]);
        return taskCompletionSource.Task;
    }

    public Task<bool> ReadAsync(byte[] buffer, int count)
    {
        TaskCompletionSource<bool> taskCompletionSource = new
        ↪ TaskCompletionSource<bool>();
        pipeStream.Read(buffer, 0, count);
        taskCompletionSource.SetResult(true);
        return taskCompletionSource.Task;
    }

    /// <summary>
    /// Attempts a read operation of the pipe stream
    /// </summary>
    /// <returns>A ReadResults struct, including whether any data was read and
    ↪ the data (if applicable)</returns>
    public ReadResults AttemptRead()
    {
        byte[] buffer = new byte[1024]; // Start by reading 1kiB of the pipe
        int bytesRead = pipeStream.Read(buffer, 0, buffer.Length);
        if (bytesRead == 0)
        {
            return new ReadResults { anythingRead = false, data = new byte[1] };
        }
        int offset = 1;
        while (bytesRead == 1024) // While the buffer gets filled
        {
            Array.Resize(ref buffer, 1024 * (offset + 1)); // Resize the buffer
            ↪ by 1kiB
            bytesRead = pipeStream.Read(buffer, offset * 1024, 1024); // Read
            ↪ the next 1k bytes
            offset++;
        }
        Array.Resize(ref buffer, (offset - 1) * 1024 + bytesRead); // Resize the
        ↪ buffer to the actual length of data
        return new ReadResults { anythingRead = true, data = buffer };
    }
```

```
public async Task<ReadResults> ReadFieldAsync(FieldType field, int
↪  fieldLength)
{
    ReadResults readResults = new ReadResults();
    byte[] buffer = new byte[fieldLength];
    if (await ReadAsync() != (PipeConstants.Marker.GENERIC | (byte)field))
    ↪  // If the received byte is not a marker with the correct field
    {
        readResults.anythingRead = false;
        return readResults;
    }

    pipeStream.Read(buffer, 0, fieldLength);
    readResults.anythingRead = true;
    readResults.data = buffer;

    return readResults;
}

/// <summary>
/// Writes a single byte to the pipe
/// </summary>
/// <param name="b">The byte to write</param>
/// <returns></returns>
public bool WriteByte(byte b)
{
    try
    {
        pipeStream.WriteByte(b);
        return true;
    }
    catch (Exception e)
    {
        Trace.WriteLine(e.Message);
        return false;
    }
}

/// <summary>
/// Performs a handshake with the client where server dictates the field
↪  length
/// </summary>
/// <param name="fieldLength">The size of the simulation domain</param>
/// <returns>true if successful, false if handshake failed</returns>
public bool Handshake(int iMax, int jMax)
{
    byte[] buffer = new byte[12];
    WriteByte(PipeConstants.Status.HELLO); // Send a HELLO byte
    if (AttemptRead().data[0] != PipeConstants.Status.HELLO) // Handshake
    ↪  not completed
    {
        return false;
    }
```

```
        buffer[0] = (byte)(PipeConstants.Marker.PRMSTART |
        ↪  PipeConstants.Marker.IMAX); // Send PRMSTART with iMax
        SerialisePrimitive(buffer, 1, iMax);
        buffer[5] = (byte)(PipeConstants.Marker.PRMEND |
        ↪  PipeConstants.Marker.IMAX); // Send corresponding PRMEND

        buffer[6] = (byte)(PipeConstants.Marker.PRMSTART |
        ↪  PipeConstants.Marker.JMAX); // Send PRMSTART with jMax
        SerialisePrimitive(buffer, 7, jMax);
        buffer[11] = (byte)(PipeConstants.Marker.PRMEND |
        ↪  PipeConstants.Marker.JMAX); // Send PRMEND

        pipeStream.Write(new ReadOnlySpan<byte>(buffer));

        ReadResults readResults = AttemptRead();
        if (readResults.anythingRead == false || readResults.data[0] !=
        ↪  PipeConstants.Status.OK) // If nothing was read or no OK byte, param
        ↪  read was unsuccessful
        {
            return false;
        }
        return true;

    }
    /// <summary>
    /// Performs a handshake with the client where the client dictates the field
    ↪  length
    /// </summary>
    /// <returns>The field length, or 0 if handshake failed</returns>
    public (int, int) Handshake()
    {
        pipeStream.WriteByte(PipeConstants.Status.HELLO); // Write a HELLO byte,
        ↪  backend dictates field dimensions
        pipeStream.WaitForPipeDrain();

        ReadResults readResults = AttemptRead();
        if (!readResults.anythingRead || readResults.data[0] !=
        ↪  PipeConstants.Status.HELLO)
        {
            return (0, 0); // Error case
        }

        if (readResults.data[1] != (PipeConstants.Marker.PRMSTART |
        ↪  PipeConstants.Marker.IMAX)) { return (0, 0); } // Should start with
        ↪  PRMSTART
        int iMax = BitConverter.ToInt32(readResults.data, 2);
        if (readResults.data[6] != (PipeConstants.Marker.PRMEND |
        ↪  PipeConstants.Marker.IMAX)) { return (0, 0); } // Should end with
        ↪  PRMEND

        if (readResults.data[7] != (PipeConstants.Marker.PRMSTART |
        ↪  PipeConstants.Marker.JMAX)) { return (0, 0); }
        int jMax = BitConverter.ToInt32(readResults.data, 8);
        if (readResults.data[12] != (PipeConstants.Marker.PRMEND |
        ↪  PipeConstants.Marker.JMAX)) { return (0, 0); }

        WriteByte(PipeConstants.Status.OK); // Send an OK byte to show the
        ↪  transmission was successful

        return (iMax, jMax);
    }

    /// <summary>
    /// Wrapper method that waits for the backend to connect to the pipe.
    /// </summary>
    public void WaitForConnection()
    {
        pipeStream.WaitForConnection();
    }

    public void CloseConnection()
    {
        pipeStream.Close();
    }

    /// <summary>
    /// Sends a parameter to the backend
    /// </summary>
    /// <param name="parameter">The value of the parameter to send</param>
    /// <param name="bits">The bits corresponding to the parameter, as read from
    ↪  <c>PipeConstants</c></param>
    public void SendParameter(float parameter, byte bits)
    {
        byte[] buffer = new byte[6];
        buffer[0] = (byte)(PipeConstants.Marker.PRMSTART | bits);
        SerialisePrimitive(buffer, 1, parameter);
        buffer[5] = (byte)(PipeConstants.Marker.PRMEND | bits);
        pipeStream.Write(buffer, 0, buffer.Length);
    }

    /// <summary>
    /// Sends a parameter to the backend
    /// </summary>
    /// <param name="parameter">The value of the parameter to send</param>
    /// <param name="bits">The bits corresponding to the parameter, as read from
    ↪  <c>PipeConstants</c></param>
    public void SendParameter(int parameter, byte bits)
    {
        byte[] buffer = new byte[2 + sizeof(float)];
        buffer[0] = (byte)(PipeConstants.Marker.PRMSTART | bits);
        SerialisePrimitive(buffer, 1, parameter);
        buffer[1 + sizeof(float)] = (byte)(PipeConstants.Marker.PRMEND | bits);
        pipeStream.Write(buffer, 0, buffer.Length);
    }
```

```
        public async Task<float> ReadParameterAsync()
        {
            byte[] buffer = new byte[sizeof(float)];
            await ReadAsync(buffer, sizeof(float));
            return BitConverter.ToSingle(buffer, 0);
        }

        /// <summary>
        /// Serialises and sends obstacle data through the pipe.
        /// </summary>
        /// <param name="obstacles">A boolean array indicating whether each cell is
        ↪   an obstacle cell or fluid cell.</param>
        /// <returns>A boolean indicating whether the transmission was
        ↪   successful.</returns>
        public bool SendObstacles(bool[] obstacles)
        {
            byte[] buffer = new byte[obstacles.Length / 8 + (obstacles.Length % 8 ==
            ↪   0 ? 0 : 1) + 1]; // Divide the length by 8 and add one if the length
            ↪   does not divide evenly. Also add 1 byte for FLDEND
            WriteByte((byte)(PipeConstants.Marker.FLDSTART |
            ↪   PipeConstants.Marker.OBST)); // Put a FLDSTART marker at the start
            int index = 0;
            for (int i = 0; i < obstacles.Length; i++)
            {
                buffer[index] |= (byte)((obstacles[i] ? 1 : 0) << i % 8); // Convert
                ↪   the bool to 1 or 0, shift it left the relevant amount of times
                ↪   and OR it with the current value in the buffer
                if (i % 8 == 7) // Add one to the index if the byte is full
                {
                    index++;
                }
            }
            buffer[^1] = (byte)(PipeConstants.Marker.FLDEND |
            ↪   PipeConstants.Marker.OBST); // And put a FLDEND at the end (^1 gets
            ↪   the last element of the array)

            pipeStream.Write(buffer, 0, buffer.Length);

            ReadResults readResults = AttemptRead();

            return readResults.anythingRead && readResults.data[0] ==
            ↪   PipeConstants.Status.OK;
        }

    }
}
```

PolarPoint.cs

```
using System;

namespace UserInterface.HelperClasses
{
    /// <summary>
    /// Represents a point defined by polar coordinates.
    /// </summary>
    /// <summary>
    /// Represents a point defined by polar coordinates.
    /// </summary>
    public class PolarPoint : IComparable<PolarPoint>, IEquatable<PolarPoint>
    {
        /// <summary>
        /// The distance from the origin to the point.
        /// </summary>
        public double Radius;

        /// <summary>
        /// The angle, in radians, with respect to a right-facing initial line.
        /// </summary>
        public double Angle;

        /// <summary>
        /// The angle, in degrees, with respect to a right-facing initial line.
        /// </summary>
        public double DegreesAngle { get => Angle * 180 / Math.PI; }

        /// <summary>
        /// Creates a polar point with a given radius and angle.
        /// </summary>
        /// <param name="radius">The distance from the origin to the point.</param>
        /// <param name="angle">The angle, in radians, with respect to a
         ↪  right-facing initial line.</param>
        public PolarPoint(double radius, double angle)
        {
            Radius = radius;
            Angle = angle;
        }

        public int CompareTo(PolarPoint? other)
        {
            if (Angle == other?.Angle) // If same angle, sort on radius
            {
                return Radius.CompareTo(other.Radius);
            }
            return Angle.CompareTo(other?.Angle);
        }

        public bool Equals(PolarPoint? other)
        {
            return Radius == other?.Radius && Angle == other?.Angle;
        }

        public override string ToString()
        {
            return $"{Radius}, {Angle}";
        }
    }
```

```
}
```

PolarSplineCalculator.cs

```
using System;
using System.Collections.Generic;
using MathNet.Numerics.LinearAlgebra;

namespace UserInterface.HelperClasses
{
    public class PolarSplineCalculator
    {
        private List<PolarPoint> controlPoints;

        private Vector<double>? splineFunctionCoefficients;
        private bool isValidSpline;

        public bool IsValidSpline { get => isValidSpline; private set =>
        ↪  isValidSpline = value; }

        public PolarSplineCalculator()
        {
            controlPoints = new List<PolarPoint>();
            IsValidSpline = false;
        }

        /// <summary>
        /// Adds one to <paramref name="input"/>. If that is equal to <paramref
        ↪  name="comparison"/>, return 0.
        /// </summary>
        /// <param name="input">The input</param>
        /// <param name="comparison">The number that <paramref name="input"/> must
        ↪  be less than.</param>
        /// <returns><paramref name="input"/> + 1, or 0.</returns>
        private static int WrapAdd(int input, int comparison) => (input + 1) ==
        ↪  comparison ? 0 : (input + 1);

        private void CalculateSplineFunction()
        {
            int numSegments = controlPoints.Count; // n segments for n points,
            ↪  beacuse the final segment wraps back around to the first.
            // For each segment:
            // Eq0: passes ith control point.
            // Eq1: passes through (i + 1)th control point.
            // Eq2: Derivative of ith segment at (i + 1)th x coordinate is equal to
            ↪  (i + 1)th segment at that x coordinate
            // Eq3: As above but for second derivative.
            // Form of each cubic: ax^3 + bx^2 + cx + d.
            Matrix<double> cubicCoefficients = Matrix<double>.Build.Dense(4 *
            ↪  numSegments, 4 * numSegments); // Create a new matrix for
            ↪  coefficients with size 4 * numSegments, because there are 4
            ↪  equations and 4 coefficients (it's a cubic) per segment.
            Vector<double> rhsValues = Vector<double>.Build.Dense(4 * numSegments);
            for (int segmentNo = 0; segmentNo < numSegments; segmentNo++)
```

```
{
    PolarPoint startPoint = controlPoints[segmentNo];
    PolarPoint endPoint;

    if (segmentNo < numSegments - 1)
    {
        endPoint = controlPoints[segmentNo + 1];
    }
    else // Last segment needs to wrap around and add 2 pi.
    {
        endPoint = new PolarPoint(controlPoints[0].Radius,
         ↪  controlPoints[0].Angle + 2 * Math.PI);
    }

    // Eq0: substitute angle of start point into cubic and equate it to
    ↪  radius of point.
    cubicCoefficients[segmentNo * 4 + 0, segmentNo * 4 + 0] =
    ↪  Math.Pow(startPoint.Angle, 3);
    cubicCoefficients[segmentNo * 4 + 0, segmentNo * 4 + 1] =
    ↪  Math.Pow(startPoint.Angle, 2);
    cubicCoefficients[segmentNo * 4 + 0, segmentNo * 4 + 2] =
    ↪  startPoint.Angle;
    cubicCoefficients[segmentNo * 4 + 0, segmentNo * 4 + 3] = 1;
    rhsValues[segmentNo * 4 + 0] = startPoint.Radius;

    // Eq1: substitute angle of end point into cubic.
    cubicCoefficients[segmentNo * 4 + 1, segmentNo * 4 + 0] =
    ↪  Math.Pow(endPoint.Angle, 3);
    cubicCoefficients[segmentNo * 4 + 1, segmentNo * 4 + 1] =
    ↪  Math.Pow(endPoint.Angle, 2);
    cubicCoefficients[segmentNo * 4 + 1, segmentNo * 4 + 2] =
    ↪  endPoint.Angle;
    cubicCoefficients[segmentNo * 4 + 1, segmentNo * 4 + 3] = 1;
    rhsValues[segmentNo * 4 + 1] = endPoint.Radius;

    // Eq2: derivatives match at end point
    cubicCoefficients[segmentNo * 4 + 2, segmentNo * 4 + 0] = 3 *
    ↪  Math.Pow(endPoint.Angle, 2);
    cubicCoefficients[segmentNo * 4 + 2, segmentNo * 4 + 1] = 2 *
    ↪  endPoint.Angle;
    cubicCoefficients[segmentNo * 4 + 2, segmentNo * 4 + 2] = 1;
    cubicCoefficients[segmentNo * 4 + 2, WrapAdd(segmentNo, numSegments)
    ↪  * 4 + 0] = -3 * Math.Pow(endPoint.Angle, 2);
    cubicCoefficients[segmentNo * 4 + 2, WrapAdd(segmentNo, numSegments)
    ↪  * 4 + 1] = -2 * endPoint.Angle;
    cubicCoefficients[segmentNo * 4 + 2, WrapAdd(segmentNo, numSegments)
    ↪  * 4 + 2] = -1;
    // RHS is 0.

    // Eq3: second derivatives match at end point
    cubicCoefficients[segmentNo * 4 + 3, segmentNo * 4 + 0] = 6 *
    ↪  endPoint.Angle;
    cubicCoefficients[segmentNo * 4 + 3, segmentNo * 4 + 1] = 2;
```

```
            cubicCoefficients[segmentNo * 4 + 3, WrapAdd(segmentNo, numSegments)
            ↪  * 4 + 0] = -6 * endPoint.Angle;
            cubicCoefficients[segmentNo * 4 + 3, WrapAdd(segmentNo, numSegments)
            ↪  * 4 + 1] = -2;
            // RHS is 0.
        }

        splineFunctionCoefficients = cubicCoefficients.Solve(rhsValues);
    }

    /// <summary>
    /// Adds a new control point.
    /// </summary>
    /// <param name="point">The point to add.</param>
    public void AddControlPoint(PolarPoint point)
    {

        controlPoints.Add(point);
        controlPoints.Sort();
        if (controlPoints.Count >= 3)
        {
            CalculateSplineFunction();
        }
    }

    public void ModifyControlPoint(PolarPoint oldPoint, PolarPoint newPoint)
    {
        controlPoints.Remove(oldPoint);
        controlPoints.Add(newPoint);
        controlPoints.Sort();
        if (controlPoints.Count >= 3)
        {
            CalculateSplineFunction();
        }
    }

    /// <summary>
    /// Removes a control point. If the control point does not exist, does
    ↪  nothing.
    /// </summary>
    /// <param name="point">The point to remove.</param>
    /// <exception cref="InvalidOperationException">Thrown if there were fewer
    ↪  than 3 points when the method was called.</exception>
    public void RemoveControlPoint(PolarPoint point)
    {
        if (controlPoints.Count < 3)
        {
            throw new InvalidOperationException("A spline must have at least 3
            ↪  points.");
        }
        controlPoints.Remove(point);
        CalculateSplineFunction();
    }
```

```
/// <summary>
/// Uses the calculated spline function to return a radius for a given
↪  angle.
/// </summary>
/// <param name="theta">The angle of the point.</param>
/// <returns>The calculated radius of a point at the supplied
↪  angle.</returns>
/// <exception cref="InvalidOperationException">Thrown if there are fewer
↪  than 3 coordinates supplied.</exception>
/// <exception cref="ArgumentOutOfRangeException">Thrown if <paramref
↪  name="theta"/> is not between 0 and 2 pi.</exception>
public double CalculatePoint(double theta)
{
    if (splineFunctionCoefficients is null)
    {
        throw new InvalidOperationException("CalculatePoint cannot be called
        ↪  when there are fewer than 3 coordinates supplied.");
    }
    if (theta < 0 || theta > 2 * Math.PI)
    {
        throw new ArgumentOutOfRangeException(nameof(theta), "Supplied angle
        ↪  must be between 0 and 2 pi.");
    }

    IsValidSpline = true;

    if (theta < controlPoints[0].Angle) theta += 2 * Math.PI; // If theta is
    ↪  before the first control point, it is in the last segment so add 2
    ↪  pi to it so it conforms to the bounds of the last segment.

    int segmentNo = controlPoints.Count - 1; // If theta is less than none
    ↪  of the coordinates then it must be in the last segment. segmentNo
    ↪  starts as this in case none of the conditions in the loop are met.

    for (int i = 0; i < controlPoints.Count - 1; i++)
    {
        if (theta < controlPoints[i + 1].Angle)
        {
            segmentNo = i;
            break;
        }
    }
    double radius = splineFunctionCoefficients[4 * segmentNo + 0] *
    ↪  Math.Pow(theta, 3)
    + splineFunctionCoefficients[4 * segmentNo + 1] * Math.Pow(theta, 2)
    + splineFunctionCoefficients[4 * segmentNo + 2] * theta
    + splineFunctionCoefficients[4 * segmentNo + 3];

    if (radius > 0)
    {
        return radius;
    }
```

```
            else
            {
                IsValidSpline = false;
                return 0;
            }
        }
    }
}
```

Queue.cs

```
namespace UserInterface.HelperClasses
{
    /// <summary>
    /// An abstract class to represent the methods for a queue.
    /// </summary>
    /// <typeparam name="T">The type of objects that will be stored in the
    ↪   queue.</typeparam>
    public abstract class Queue<T>
    {
        protected T[] array;

        protected int front;
        protected int back;
        protected int length;

        /// <summary>
        /// Initialises a queue with length <paramref name="length"/>
        /// </summary>
        /// <param name="length">The length of the queue</param>
        public Queue(int length)
        {
            array = new T[length];
            this.length = length;
        }

        /// <summary>
        /// Adds <paramref name="item"/> to the back of the queue.
        /// </summary>
        /// <param name="item">The item to add to the queue.</param>
        public abstract void Enqueue(T item);

        /// <summary>
        /// Removes one item from the front of the queue, and returns it.
        /// </summary>
        /// <returns>The item that used to be at the front of the queue.</returns>
        public abstract T Dequeue();

        /// <summary>
        /// Returns whether the queue is full.
        /// </summary>
        public abstract bool IsFull { get; }

        /// <summary>
        /// Returns whether the queue is empty.
        /// </summary>
        public abstract bool IsEmpty { get; }

        /// <summary>
        /// Returns the number of items in the queue.
        /// </summary>
        public abstract int Count { get; }
    }
}
```

ResizableLinearQueue.cs

```
using System;

namespace UserInterface.HelperClasses
{
    /// <summary>
    /// A memory-efficient linear queue implementation.
    /// </summary>
    /// <typeparam name="T">The type of the objects that will be stored in the
     ↪  queue.</typeparam>
    public class ResizableLinearQueue<T> : Queue<T>
    {

        /// <summary>
        /// Initialises the linear queue with a known start length.
        /// </summary>
        /// <param name="startLength">The initial length of the queue.</param>
        public ResizableLinearQueue(int startLength) : base(startLength)
        {
            back = 0;
            front = 0;
        }

        /// <summary>
        /// Initialises the linear queue with a default initial length of 1.
        /// </summary>
        public ResizableLinearQueue() : this(1) { } // If no length is specified,
         ↪  start at 1.

        /// <summary>
        /// Moves all the elements forwards in the array such the front of the queue
         ↪  is at position 0 and the rest are stored contiguously.
        /// </summary>
        private void Reshuffle()
        {
            for (int i = front; i < length; i++)
            {
                array[i - front] = array[i]; // Reshuffle the array by copying each
                 ↪  item back a certain number of spaces
            }
            back -= front; // Reshuffle the pointers
            front = 0;
        }

        public override void Enqueue(T value)
        {
            if (back == length)
            {
                if (front > 0) // If the queue is full and the front is not at 0,
                ↪  there is unused space at the start of the array
                {
                    Reshuffle();
                }
                else // If the queue is completely full (front pointer at 0), make
                ↪  the queue 2x longer (don't reshuffle because this will have no
                ↪  effect)
                {
                    length *= 2;
                    Array.Resize(ref array, length);
                }
            }
            array[back] = value; // Put an item at the back and increase the back
            ↪  pointer
            back++;
        }

        public override T Dequeue()
        {
            if (IsEmpty)
            {
                throw new InvalidOperationException("CircularQueue was empty when
                ↪  Dequeue was called");
            }
            T removedItem = array[front];
            front++;

            if (back - front < length / 4) // If only 1/4 of the array now is used,
            ↪  reshuffle it and halve the size
            {
                Reshuffle();
                length /= 2;
                Array.Resize(ref array, length);
            }
            return removedItem; // Return the item that has been "removed"
        }

        public override bool IsEmpty => front == back;

        public override bool IsFull => false; // As the queue is resizable, it is
        ↪  never full. This is here for inheritance reasons.

        public override int Count => front - back;
    }
}
```

UnitChangedEventArgs.cs

```
using System.ComponentModel;
using UserInterface.Converters;

namespace UserInterface.HelperClasses
{
    public class UnitChangedEventArgs : PropertyChangedEventArgs
    {
        public UnitClasses.Unit NewValue { get; private set; }

        public UnitChangedEventArgs(string propertyName, UnitClasses.Unit newValue)
         ↪   : base(propertyName)
        {
            NewValue = newValue;
        }
    }
}
```

UnitHolder.cs

```
using System;
using System.ComponentModel;
using System.Runtime.CompilerServices;
using UserInterface.Converters;

namespace UserInterface.HelperClasses
{
    public class UnitHolder : INotifyPropertyChanged
    {
        private UnitClasses.LengthUnit lengthUnit;
        private UnitClasses.SpeedUnit speedUnit;
        private UnitClasses.TimeUnit timeUnit;
        private UnitClasses.DensityUnit densityUnit;
        private UnitClasses.ViscosityUnit viscosityUnit;

        private readonly UnitClasses.UnitSystemList SIList;
        private readonly UnitClasses.UnitSystemList CGSList;
        private readonly UnitClasses.UnitSystemList DefaultImperialList;

        public UnitClasses.LengthUnit LengthUnit
        {
            get => lengthUnit;
            set
            {
                lengthUnit = value;
                OnPropertyChanged(LengthUnit);
            }
        }
        public UnitClasses.SpeedUnit SpeedUnit
        {
            get => speedUnit;
            set
            {
```

```
            speedUnit = value;
            OnPropertyChanged(SpeedUnit);
        }
    }
    public UnitClasses.TimeUnit TimeUnit
    {
        get => timeUnit;
        set
        {
            timeUnit = value;
            OnPropertyChanged(TimeUnit);
        }
    }
    public UnitClasses.DensityUnit DensityUnit
    {
        get => densityUnit;
        set
        {
            densityUnit = value;
            OnPropertyChanged(DensityUnit);
        }
    }
    public UnitClasses.ViscosityUnit ViscosityUnit
    {
        get => viscosityUnit;
        set
        {
            viscosityUnit = value;
            OnPropertyChanged(ViscosityUnit);
        }
    }

    public event PropertyChangedEventHandler? PropertyChanged;

    public UnitHolder()
    {
        lengthUnit = new UnitClasses.Metre();
        speedUnit = new UnitClasses.MetrePerSecond();
        timeUnit = new UnitClasses.Second();
        densityUnit = new UnitClasses.KilogramPerCubicMetre();
        viscosityUnit = new UnitClasses.KilogramPerMetrePerSecond();

        SIList = new UnitClasses.UnitSystemList(UnitClasses.UnitSystem.SI, new
        ↪   UnitClasses.Metre(), new UnitClasses.MetrePerSecond(), new
        ↪   UnitClasses.Second(), new UnitClasses.KilogramPerCubicMetre(), new
        ↪   UnitClasses.KilogramPerMetrePerSecond());
        CGSList = new UnitClasses.UnitSystemList(UnitClasses.UnitSystem.CGS, new
        ↪   UnitClasses.Centimetre(), new UnitClasses.CentimetrePerSecond(), new
        ↪   UnitClasses.Second(), new UnitClasses.GramPerCubicCentimetre(), new
        ↪   UnitClasses.GramPerCentimetrePerSecond());
```

```
            DefaultImperialList = new
            ↪  UnitClasses.UnitSystemList(UnitClasses.UnitSystem.SI, new
            ↪  UnitClasses.Foot(), new UnitClasses.FootPerSecond(), new
            ↪  UnitClasses.Second(), new UnitClasses.PoundPerCubicFoot(), new
            ↪  UnitClasses.PoundSecondPerSquareFoot());
        }

        private void OnPropertyChanged(UnitClasses.Unit value, [CallerMemberName]
        ↪  string name = "")
        {
            PropertyChanged?.Invoke(this, new UnitChangedEventArgs(name, value));
        }

        public void SetUnitSystem(UnitClasses.UnitSystem unitSystem)
        {
            UnitClasses.UnitSystemList unitSystemList = unitSystem switch
            {
                UnitClasses.UnitSystem.SI => SIList,
                UnitClasses.UnitSystem.CGS => CGSList,
                UnitClasses.UnitSystem.DefaultImperial => DefaultImperialList,
                _ => throw new ArgumentException("Unit system parameter was not a
                ↪  valid unit system.", nameof(unitSystem))
            };

            LengthUnit = unitSystemList.lengthUnit;
            SpeedUnit = unitSystemList.speedUnit;
            TimeUnit = unitSystemList.timeUnit;
            DensityUnit = unitSystemList.densityUnit;
            ViscosityUnit = unitSystemList.viscosityUnit;
        }
    }
}
```

ObstacleCell.cs

```
namespace UserInterface.HelperControls
{
    public class ObstacleCell
    {
        public double X { get; set; }
        public double Y { get; set; }
        public double Width { get; set; }
        public double Height { get; set; }
    }
}
```

ResizableCentredTextBox.xaml

```
<UserControl x:Class="UserInterface.HelperControls.ResizableCentredTextBox"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             mc:Ignorable="d"
             d:DesignHeight="450" d:DesignWidth="800">
    <Viewbox HorizontalAlignment="Center" VerticalAlignment="Center"
     ↪  x:Name="LayoutRoot">
        <TextBlock Margin="10 0 10 0" Text="{Binding Text}" />
    </Viewbox>
</UserControl>
```

ResizableCentredTextBox.xaml.cs

```
using System.Windows;
using System.Windows.Controls;

namespace UserInterface.HelperControls
{
    /// <summary>
    /// Interaction logic for ResizableCentredTextBox.xaml
    /// </summary>
    public partial class ResizableCentredTextBox : UserControl
    {
        public string Text
        {
            get { return (string)GetValue(TextProperty); }
            set { SetValue(TextProperty, value); }
        }

        // Using a DependencyProperty as the backing store for Text.  This enables
        ↪  animation, styling, binding, etc...
        public static readonly DependencyProperty TextProperty =
            DependencyProperty.Register("Text", typeof(string),
            ↪  typeof(ResizableCentredTextBox), new PropertyMetadata("Text not
            ↪  bound."));

        public ResizableCentredTextBox()
        {
            InitializeComponent();
            LayoutRoot.DataContext = this;
        }
    }
}
```

SliderWithValue.xaml

```
<UserControl x:Class="UserInterface.HelperControls.SliderWithValue"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             xmlns:diag="clr-namespace:System.Diagnostics;assembly=WindowsBase"
             xmlns:converters="clr-namespace:UserInterface.Converters"
             mc:Ignorable="d"
             d:DesignHeight="450" d:DesignWidth="800">
    <UserControl.Resources>
        <ResourceDictionary>
            <converters:BoolToTickPlacement x:Key="BoolToTickPlacementConverter" />
            <converters:SignificantFigures x:Key="SignificantFiguresConverter" />
        </ResourceDictionary>
    </UserControl.Resources>
    <Grid x:Name="LayoutRoot">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="Auto" />
        </Grid.ColumnDefinitions>
        <Label Grid.Row="0" Grid.Column="0" Content="{Binding ConvertedMinimum,
        ↪   Converter={StaticResource SignificantFiguresConverter},
        ↪   ConverterParameter=3}" d:Content="0" />
        <Label Grid.Row="0" Grid.Column="2" Content="{Binding ConvertedMaximum,
        ↪   Converter={StaticResource SignificantFiguresConverter},
        ↪   ConverterParameter=3}" d:Content="1" />
        <Label Grid.Row="0" Grid.Column="3" Content="{Binding UnitShortName}"
        ↪   d:Content="m" />
        <Slider Grid.Row="1" Grid.Column="0" Grid.ColumnSpan="3" Minimum="{Binding
        ↪   ConvertedMinimum}" Maximum="{Binding ConvertedMaximum}" Value="{Binding
        ↪   ConvertedValue}"  x:Name="slider" Margin="10 0 10 0" Ticks="{Binding
        ↪   ForceIntegers, Converter={StaticResource BoolToTickPlacementConverter},
        ↪   Mode=OneWay}" TickFrequency="1" IsSnapToTickEnabled="{Binding
        ↪   ForceIntegers}" />
        <TextBox Grid.Row="1" Grid.Column="3" Text="{Binding ConvertedValue,
        ↪   UpdateSourceTrigger=PropertyChanged}" TextAlignment="Right" Width="40"
        ↪   />
    </Grid>
</UserControl>
```

SliderWithValue.xaml.cs

```
using System.Collections.Generic;
using System.ComponentModel;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Controls.Primitives;
using UserInterface.Converters;
using UserInterface.HelperClasses;

namespace UserInterface.HelperControls
{
    /// <summary>
    /// Interaction logic for SliderWithValue.xaml
    /// </summary>
    public partial class SliderWithValue : UserControl, INotifyPropertyChanged
    {

        // Dependency properties are used extensively here to allow for bindings on
        ↪   Value, Minimum and Maximum.
```

```
public double Value
{
    get { return (double)GetValue(ValueProperty); }
    set { SetValue(ValueProperty, value); }
}
public static readonly DependencyProperty ValueProperty =
    DependencyProperty.Register("Value", typeof(double),
    ↪  typeof(SliderWithValue), new FrameworkPropertyMetadata(0.0,
    ↪  FrameworkPropertyMetadataOptions.BindsTwoWayByDefault,
    ↪  OnValueChanged)); // Value is two-way: changes to the slider must be
    ↪  passed to the source that uses this UserControl.
public double ConvertedValue
{
    get
    {
        Units UnitConverter = new Units(Unit);
        return (double)UnitConverter.ConvertBack(Value, typeof(double), 0,
        ↪  System.Globalization.CultureInfo.CurrentCulture);
    }
    set
    {
        Units UnitConverter = new Units(Unit);
        Value = (double)UnitConverter.Convert(value, typeof(double), 0,
        ↪  System.Globalization.CultureInfo.CurrentCulture);
        OnPropertyChanged(nameof(ConvertedValue));
    }
}
private static void OnValueChanged(DependencyObject d,
↪  DependencyPropertyChangedEventArgs e)
{
    if (d is SliderWithValue slider)
    {
        slider.OnPropertyChanged(nameof(slider.ConvertedValue));
    }
}

public double Minimum
{
    get { return (double)GetValue(MinimumProperty); }
    set { SetValue(MinimumProperty, value); }
}
public static readonly DependencyProperty MinimumProperty =
    DependencyProperty.Register("Minimum", typeof(double),
    ↪  typeof(SliderWithValue), new PropertyMetadata(0d,
    ↪  OnMinimumChanged));
public double ConvertedMinimum
{
    get
    {
        Units UnitConverter = new Units(Unit);
        return (double)UnitConverter.ConvertBack(Minimum, typeof(double), 0,
        ↪  System.Globalization.CultureInfo.CurrentCulture);
    }
    set
    {
        Units UnitConverter = new Units(Unit);
        Minimum = (double)UnitConverter.Convert(value, typeof(double), 0,
        ↪  System.Globalization.CultureInfo.CurrentCulture);
        OnPropertyChanged(nameof(ConvertedMinimum));
    }
}
private static void OnMinimumChanged(DependencyObject d,
↪  DependencyPropertyChangedEventArgs e)
{
    if (d is SliderWithValue slider)
    {
        slider.OnPropertyChanged(nameof(slider.ConvertedMinimum));
    }
}

public double Maximum
{
    get { return (double)GetValue(MaximumProperty); }
    set { SetValue(MaximumProperty, value); }
}
public static readonly DependencyProperty MaximumProperty =
    DependencyProperty.Register("Maximum", typeof(double),
     ↪  typeof(SliderWithValue), new PropertyMetadata(1d,
     ↪  OnMaximumChanged));
public double ConvertedMaximum
{
    get
    {
        Units UnitConverter = new Units(Unit);
        return (double)UnitConverter.ConvertBack(Maximum, typeof(double), 0,
        ↪  System.Globalization.CultureInfo.CurrentCulture);
    }
    set
    {
        Units UnitConverter = new Units(Unit);
        Maximum = (double)UnitConverter.Convert(value, typeof(double), 0,
        ↪  System.Globalization.CultureInfo.CurrentCulture);
        OnPropertyChanged(nameof(ConvertedMaximum));
    }
}
private static void OnMaximumChanged(DependencyObject d,
↪  DependencyPropertyChangedEventArgs e)
{
    if (d is SliderWithValue slider)
    {
        slider.OnPropertyChanged(nameof(slider.ConvertedMaximum));
    }
}

public string UnitShortName { get => Unit?.ShortName ?? ""; }
public UnitClasses.Unit Unit
{
    get { return (UnitClasses.Unit)GetValue(UnitProperty); }
    set { SetValue(UnitProperty, value); }
}

public static readonly DependencyProperty UnitProperty =
    DependencyProperty.Register(nameof(Unit), typeof(UnitClasses.Unit),
     ↪  typeof(SliderWithValue), new
     ↪  PropertyMetadata(OnUnitChangedCallBack));

public event PropertyChangedEventHandler? PropertyChanged;

public SliderWithValue()
{
    InitializeComponent();
    LayoutRoot.DataContext = this;
}

private static void OnUnitChangedCallBack(DependencyObject sender,
↪  DependencyPropertyChangedEventArgs e)
{
    if (sender is SliderWithValue sliderWithValue)
    {
        sliderWithValue.OnUnitChanged();
    }
}

public void OnUnitChanged()
{
    OnPropertiesChanged([nameof(ConvertedValue), nameof(ConvertedMinimum),
     ↪  nameof(ConvertedMaximum), nameof(UnitShortName)]);
}

private void OnPropertiesChanged(IEnumerable<string> properties)
{
    foreach (string property in properties)
    {
        OnPropertyChanged(property);
    }
}

private void OnPropertyChanged(string property)
{
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(property));
}

public bool ForceIntegers { get; set; } = false;

public TickPlacement TickPlacement { get; } = TickPlacement.None;
}
```

```
}
```

UnitConversionPanel.xaml

```
<UserControl x:Class="UserInterface.HelperControls.UnitConversionPanel"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             xmlns:local="clr-namespace:UserInterface.HelperControls"
             mc:Ignorable="d"
             d:DesignHeight="450" d:DesignWidth="800">

    <Grid x:Name="LayoutRoot">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>
        <Label Grid.Row="0" Grid.Column="0">Unit system:</Label>
        <ComboBox Grid.Row="0" Grid.Column="1" x:Name="UnitSystemComboBox"
        ↪   SelectionChanged="OnUnitSystemChanged">
            <ComboBoxItem Content="SI (metre, kilogram, second)" d:IsSelected="True"
            ↪   />
            <ComboBoxItem Content="CGS (centimetre, gram, second)" />
            <ComboBoxItem Content="Imperial (foot, pound, second)" />
        </ComboBox>
        <Label Grid.Row="1" Grid.Column="0">Length units:</Label>
        <ComboBox Grid.Row="1" Grid.Column="1" x:Name="LengthComboBox"
        ↪   SelectionChanged="OnSelectionChanged">
            <ComboBoxItem Content="Metres" d:IsSelected="True" />
            <ComboBoxItem Content="Feet" />
            <ComboBoxItem Content="Inches" />
            <ComboBoxItem Content="Centimetres" />
            <ComboBoxItem Content="Millimetres" />
        </ComboBox>
        <Label Grid.Row="2" Grid.Column="0">Speed units:</Label>
        <ComboBox Grid.Row="2" Grid.Column="1" x:Name="SpeedComboBox"
        ↪   SelectionChanged="OnSelectionChanged">
            <ComboBoxItem Content="Metres per second" d:IsSelected="True" />
            <ComboBoxItem Content="Centimetres per second" />
            <ComboBoxItem Content="Miles per hour" />
            <ComboBoxItem Content="Kilometres per hour" />
            <ComboBoxItem Content="Feet per second" />
        </ComboBox>
        <Label Grid.Row="3" Grid.Column="0">Time units:</Label>
        <ComboBox Grid.Row="3" Grid.Column="1" x:Name="TimeComboBox"
        ↪   SelectionChanged="OnSelectionChanged">
            <ComboBoxItem Content="Seconds" d:IsSelected="True" />
            <ComboBoxItem Content="Milliseconds" />
            <ComboBoxItem Content="Minutes" />
            <ComboBoxItem Content="Hours" />
        </ComboBox>
        <Label Grid.Row="4" Grid.Column="0">Density units:</Label>
        <ComboBox Grid.Row="4" Grid.Column="1" x:Name="DensityComboBox"
        ↪   SelectionChanged="OnSelectionChanged">
            <ComboBoxItem Content="Kilograms per cubic metre" d:IsSelected="True" />
            <ComboBoxItem Content="Grams per cubic centimetre" />
            <ComboBoxItem Content="Pounds per cubic inch" />
            <ComboBoxItem Content="Pounds per cubic foot" />
        </ComboBox>
        <Label Grid.Row="5" Grid.Column="0">Viscosity units:</Label>
        <ComboBox Grid.Row="5" Grid.Column="1" x:Name="ViscosityComboBox"
        ↪   SelectionChanged="OnSelectionChanged">
            <ComboBoxItem Content="Kilograms per metre per second"
            ↪   d:IsSelected="True" />
            <ComboBoxItem Content="Grams per centimetre per second" />
            <ComboBoxItem Content="Pound-seconds per square foot" />
        </ComboBox>
    </Grid>
</UserControl>
```

UnitConversionPanel.xaml.cs

```
using System;
using System.ComponentModel;
using System.Windows;
using System.Windows.Controls;
using UserInterface.Converters;
using UserInterface.HelperClasses;

namespace UserInterface.HelperControls
{
    /// <summary>
    /// Interaction logic for UnitConversionPanel.xaml
    /// </summary>
    public partial class UnitConversionPanel : UserControl
    {
        private readonly UnitHolder unitHolder;

        public UnitConversionPanel(UnitHolder unitHolder)
        {
            InitializeComponent();
            this.unitHolder = unitHolder;
            SetInitialComboBoxContents();
            unitHolder.PropertyChanged += OnUnitChanged;
        }

        private void SetInitialComboBoxContents()
        {
```

```
        SetComboBoxContents(unitHolder.LengthUnit, LengthComboBox);
        SetComboBoxContents(unitHolder.SpeedUnit, SpeedComboBox);
        SetComboBoxContents(unitHolder.TimeUnit, TimeComboBox);
        SetComboBoxContents(unitHolder.DensityUnit, DensityComboBox);
        SetComboBoxContents(unitHolder.ViscosityUnit, ViscosityComboBox);
    }

    private void OnUnitChanged(object? sender, PropertyChangedEventArgs e)
    {
        UnitChangedEventArgs args = e as UnitChangedEventArgs;
        ComboBox? relevantComboBox = args.PropertyName switch
        {
            nameof(unitHolder.LengthUnit) => LengthComboBox,
            nameof(unitHolder.SpeedUnit) => SpeedComboBox,
            nameof(unitHolder.TimeUnit) => TimeComboBox,
            nameof(unitHolder.DensityUnit) => DensityComboBox,
            nameof(unitHolder.ViscosityUnit) => ViscosityComboBox,
            _ => null
        };
        if (relevantComboBox is null) return;
        SetComboBoxContents(args.NewValue, relevantComboBox);
    }

    private void SetComboBoxContents(UnitClasses.Unit newUnit, ComboBox?
    ↪   relevantComboBox)
    {
        relevantComboBox.SelectionChanged -= OnSelectionChanged;

        foreach (var item in relevantComboBox.Items)
        {
            if (item is ComboBoxItem comboBoxItem &&
            ↪   (string)comboBoxItem.Content == newUnit.LongName)
            {
                comboBoxItem.IsSelected = true;
            }
        }
        relevantComboBox.SelectionChanged += OnSelectionChanged;
    }

    private void OnUnitSystemChanged(object sender, SelectionChangedEventArgs e)
    {
        int selectedIndex = (sender as ComboBox).SelectedIndex;
        if (selectedIndex == -1 || selectedIndex > 2) return;
        UnitClasses.UnitSystem unitSystem =
        ↪   (UnitClasses.UnitSystem)selectedIndex;
        unitHolder.SetUnitSystem(unitSystem);
    }

    private void OnSelectionChanged(object sender, SelectionChangedEventArgs e)
    {
        if (sender is not ComboBox relevantComboBox) return;
        UnitSystemComboBox.SelectedIndex = -1;
        unitHolder.PropertyChanged -= OnUnitChanged;
```

```
switch (relevantComboBox.Name)
{
    case "LengthComboBox":
        unitHolder.LengthUnit = relevantComboBox.SelectedIndex switch
        {
            0 => new UnitClasses.Metre(),
            1 => new UnitClasses.Foot(),
            2 => new UnitClasses.Inch(),
            3 => new UnitClasses.Centimetre(),
            4 => new UnitClasses.Millimetre(),
            _ => null
        };
        break;
    case "SpeedComboBox":
        unitHolder.SpeedUnit = relevantComboBox.SelectedIndex switch
        {
            0 => new UnitClasses.MetrePerSecond(),
            1 => new UnitClasses.CentimetrePerSecond(),
            2 => new UnitClasses.MilePerHour(),
            3 => new UnitClasses.KilometrePerHour(),
            4 => new UnitClasses.FootPerSecond(),
            _ => null
        };
        break;
    case "TimeComboBox":
        unitHolder.TimeUnit = relevantComboBox.SelectedIndex switch
        {
            0 => new UnitClasses.Second(),
            1 => new UnitClasses.Millisecond(),
            2 => new UnitClasses.Minute(),
            3 => new UnitClasses.Hour(),
            _ => null
        };
        break;
    case "DensityComboBox":
        unitHolder.DensityUnit = relevantComboBox.SelectedIndex switch
        {
            0 => new UnitClasses.KilogramPerCubicMetre(),
            1 => new UnitClasses.GramPerCubicCentimetre(),
            2 => new UnitClasses.PoundPerCubicInch(),
            3 => new UnitClasses.PoundPerCubicFoot(),
            _ => null
        };
        break;
    case "ViscosityComboBox":
        unitHolder.ViscosityUnit = relevantComboBox.SelectedIndex switch
        {
            0 => new UnitClasses.KilogramPerMetrePerSecond(),
            1 => new UnitClasses.GramPerCentimetrePerSecond(),
            2 => new UnitClasses.PoundSecondPerSquareFoot(),
            _ => null
        };
        break;
```

```
            }
            unitHolder.PropertyChanged += OnUnitChanged;
        }
    }
}
```

VisualPoint.cs

```
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Shapes;

namespace UserInterface.HelperControls
{
    public class VisualPoint : Shape
    {
        private const int defaultCircleRadiusRatio = 3;
        private const int hoverCircleRadiusRatio = 2;
        private const int size = 1;
        private const float aspectRatio = (float)9/16;
        private readonly DrawingBrush defaultFill;
        private readonly DrawingBrush mouseHoverFill;

        private Point point;
        private bool isDragged;

        protected override Geometry DefiningGeometry => new EllipseGeometry(new
        ↪   Point(0, 0), size * aspectRatio, size);

        public Point Point
        {
            get => point;
            set
            {
                point = value;
                Canvas.SetLeft(this, point.X - (size * aspectRatio) / 2);
                Canvas.SetBottom(this, point.Y - size);
            }
        }

        public bool IsDragged
        {
            get => isDragged;
            set
            {
                isDragged = value;
                OnIsDraggedChanged();
            }
        }

        public VisualPoint(Point point)
        {
```

```
        defaultFill = new DrawingBrush(new DrawingGroup
        {
            Children =
            [
                new GeometryDrawing // Outer circle
                {
                    Brush = Brushes.DarkGreen,
                    Geometry = new EllipseGeometry(new Point(0, 0), aspectRatio
                    ↪  * defaultCircleRadiusRatio, defaultCircleRadiusRatio)
                },
                new GeometryDrawing // Inner, darker circle
                {
                    Brush = Brushes.LightGreen,
                    Geometry = new EllipseGeometry(new Point(0, 0), aspectRatio,
                    ↪  1)
                }
            ]
        });

        mouseHoverFill = new DrawingBrush(new DrawingGroup
        {
            Children =
            [
                new GeometryDrawing // Outer circle
                {
                    Brush = Brushes.DarkGreen,
                    Geometry = new EllipseGeometry(new Point(0, 0), aspectRatio
                    ↪  * hoverCircleRadiusRatio, hoverCircleRadiusRatio)
                },
                new GeometryDrawing // Inner, darker circle
                {
                    Brush = Brushes.LightGreen,
                    Geometry = new EllipseGeometry(new Point(0, 0), aspectRatio,
                    ↪  1)
                }
            ]
        });

        isDragged = false;

        MouseEnter += OnMouseEnter;
        MouseLeave += OnMouseLeave;

        Fill = defaultFill;
        Point = point;
    }

    public VisualPoint() : this(new Point(0, 0)) { }

    public VisualPoint(double x, double y) : this(new Point(x, y)) { }

    private void OnIsDraggedChanged()
    {
            if (isDragged) // Dragging has just started
            {
                MouseEnter -= OnMouseEnter;
                MouseLeave -= OnMouseLeave;
                Fill = mouseHoverFill;
            }
            else // Dragging has just ended
            {
                MouseEnter += OnMouseEnter;
                MouseLeave += OnMouseLeave;
                Fill = defaultFill;
            }
        }

        private void OnMouseEnter(object sender, MouseEventArgs e) => Fill =
        ↪   mouseHoverFill;

        private void OnMouseLeave(object sender, MouseEventArgs e) => Fill =
        ↪   defaultFill;

        public override string ToString()
        {
            return point.ToString();
        }
    }
}
```

AdvancedParametersVM.cs

```
using System.Windows.Navigation;
using UserInterface.HelperClasses;

namespace UserInterface.ViewModels
{
    public class AdvancedParametersVM : ViewModel
    {
        #region Field and Properties
        private float tau;
        private float omega;
        private float rMax;
        private float iterMax;

        public float Tau
        {
            get => tau;
            set { tau = value; OnPropertyChanged(this, nameof(Tau)); }
        }

        public float Omega {
            get => omega;
            set { omega = value; OnPropertyChanged(this, nameof(Omega)); }
        }

        public float RMax
        {
            get => rMax;
            set { rMax = value; OnPropertyChanged(this, nameof(RMax)); }
        }

        public float IterMax
        {
            get => iterMax;
            set { iterMax = value; OnPropertyChanged(this, nameof(IterMax)); }
        }

        public float DelX
        {
            get => parameterHolder.Width.Value / obstacleHolder.DataWidth;
            set
            {
                obstacleHolder.DataWidth = (int)(parameterHolder.Width.Value /
                ↪  value);
                OnPropertyChanged(this, nameof(DelX));
            }
        }
        public float DelY
        {
            get => parameterHolder.Height.Value / obstacleHolder.DataHeight;
            set
            {
                obstacleHolder.DataHeight = (int)(parameterHolder.Height.Value /
                ↪  value);
                OnPropertyChanged(this, nameof(DelY));
            }
        }

        public bool CanChangeGridSizes
        {
            get => !obstacleHolder.UsingObstacleFile;
        }

        public Commands.AdvancedParametersReset ResetCommand { get; set; }

        public Commands.SaveParameters SaveCommand { get; set; }
        #endregion

        public AdvancedParametersVM(ParameterHolder parameterHolder, UnitHolder
        ↪  unitHolder, ObstacleHolder obstacleHolder) : base(parameterHolder,
        ↪  unitHolder, obstacleHolder)
        {
            Tau = parameterHolder.TimeStepSafetyFactor.Value;
            Omega = parameterHolder.RelaxationParameter.Value;
            RMax = parameterHolder.PressureResidualTolerance.Value;
            IterMax = parameterHolder.PressureMaxIterations.Value;

            ResetCommand = new Commands.AdvancedParametersReset(this,
            ↪  parameterHolder);
            SaveCommand = new Commands.SaveParameters(this, parameterHolder, new
            ↪  Commands.ChangeWindow());
        }
    }
}
```

ConfigScreenVM.cs

```
using UserInterface.HelperClasses;
using UserInterface.HelperControls;

namespace UserInterface.ViewModels
{
    public class ConfigScreenVM : ViewModel
    {
        private readonly UnitConversionPanel unitsPanel;

        public float InVel
        {
            get => parameterHolder.InflowVelocity.Value;
            set
            {
                parameterHolder.InflowVelocity =
                ↪  ModifyParameterValue(parameterHolder.InflowVelocity, value);
                OnPropertyChanged(this, nameof(InVel));
            }
        }
        public float Chi
        {
            get => parameterHolder.SurfaceFriction.Value;
            set
            {
                parameterHolder.SurfaceFriction =
                ↪  ModifyParameterValue(parameterHolder.SurfaceFriction, value);
                OnPropertyChanged(this, nameof(Chi));
            }
        }
        public float Width
        {
            get => parameterHolder.Width.Value;
            set
            {
                parameterHolder.Width = ModifyParameterValue(parameterHolder.Width,
                ↪  value);
                OnPropertyChanged(this, nameof(Width));
            }
        }
        public float Height
        {
            get => parameterHolder.Height.Value;
            set
            {
                parameterHolder.Height =
                ↪  ModifyParameterValue(parameterHolder.Height, value);
                OnPropertyChanged(this, nameof(Height));
            }
        }
        public float ReynoldsNo
        {
            get => parameterHolder.ReynoldsNumber.Value;
            set
            {
                parameterHolder.ReynoldsNumber =
                ↪  ModifyParameterValue(parameterHolder.ReynoldsNumber, value);
                OnPropertyChanged(this, nameof(ReynoldsNo));
            }
        }
        public float Viscosity
        {
            get => parameterHolder.FluidViscosity.Value;
            set {
                parameterHolder.FluidViscosity =
                ↪  ModifyParameterValue(parameterHolder.FluidViscosity, value);
                OnPropertyChanged(this, nameof(Viscosity));
            }
        }
        public float Density
        {
            get => parameterHolder.FluidDensity.Value;
            set
            {
                parameterHolder.FluidDensity =
                ↪  ModifyParameterValue(parameterHolder.FluidDensity, value);
                OnPropertyChanged(this, nameof(Density));
            }
        }

        public string? FileName
        {
            get => obstacleHolder.FileName;
            set
            {
                obstacleHolder.FileName = value;
                OnPropertyChanged(this, nameof(FileName));
                OnPropertyChanged(this, nameof(DisplayFileName));
            }
        }
        public string DisplayFileName
        {
            get
            {
                if (obstacleHolder.UsingObstacleFile == false)
                {
                    return "Not using obstacle files.";
                }
                if (obstacleHolder.FileName == null)
                {
                    return "No File Selected.";
                }
                string[] fileParts = obstacleHolder.FileName.Split('\\');
                return $"File selected: {fileParts[^1]}";
            }
        }
        public bool UsingObstacleFile
        {
            get => obstacleHolder.UsingObstacleFile;
            set
            {
                obstacleHolder.UsingObstacleFile = value;
                OnPropertyChanged(this, nameof(UsingObstacleFile));
                OnPropertyChanged(this, nameof(DisplayFileName));
            }
        }

        public UnitConversionPanel UnitsPanel => unitsPanel;

        public Commands.ConfigScreenReset ResetCommand { get; set; }
        public Commands.SetAirParameters SetAirCommand { get; set; }
        public Commands.SelectObstacleFile SelectObstacleFileCommand { get; set; }
        public Commands.TrySimulate TrySimulateCommand { get; set; }
        public Commands.CreatePopup CreatePopupCommand { get; set; }

        public ConfigScreenVM(ParameterHolder parameterHolder, UnitHolder
            ↪  unitHolder, ObstacleHolder obstacleHolder) : base(parameterHolder,
            ↪  unitHolder, obstacleHolder)
        {
            InVel = parameterHolder.InflowVelocity.Value;
            Chi = parameterHolder.SurfaceFriction.Value;
            Width = parameterHolder.Width.Value;
            Height = parameterHolder.Height.Value;
            ReynoldsNo = parameterHolder.ReynoldsNumber.Value;
            Viscosity = parameterHolder.FluidViscosity.Value;
            Density = parameterHolder.FluidDensity.Value;

            unitsPanel = new UnitConversionPanel(unitHolder);
            ResetCommand = new Commands.ConfigScreenReset(this, parameterHolder);
            SetAirCommand = new Commands.SetAirParameters(this);
            SelectObstacleFileCommand = new Commands.SelectObstacleFile(this);
            TrySimulateCommand = new Commands.TrySimulate(this);
            CreatePopupCommand = new Commands.CreatePopup();
        }
    }
}
```

SimulationScreenVM.cs

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Collections.Specialized;
using System.ComponentModel;
using System.Diagnostics;
using System.IO;
using System.Threading;
using System.Threading.Tasks;
using System.Windows;
using UserInterface.Converters;
using UserInterface.HelperClasses;
using UserInterface.HelperControls;

namespace UserInterface.ViewModels
{
    public class SimulationScreenVM : ViewModel
    {
        private const int CANVAS_WIDTH = 100;
        private const int CANVAS_HEIGHT = 100;
        #region Fields, Properties and Enums
        private SidePanelButton? currentButton;

        private float visLowerBound;
        private float visUpperBound;

        private readonly float[] velocity;
        private readonly float[] pressure;
        private readonly float[] streamFunction;
        private FieldParameters pressureParameters;
        private FieldParameters velocityParameters;
        private SelectedField selectedField;

        private readonly BackendManager backendManager;
        private CancellationTokenSource backendCTS;
        private readonly UnitConversionPanel unitsPanel;
        private readonly VisualisationControl visualisationControl;
        private readonly MovingAverage<float> visFrameTimeAverage;
        private readonly MovingAverage<float> backFrameTimeAverage;
        private readonly MovingAverage<float> dragCoefficientAverage;
        private readonly RectangularToPolar RecToPolConverter;

        private readonly int dataWidth;
        private readonly int dataHeight;

        private readonly ObservableCollection<PolarPoint> obstaclePoints;
        private readonly ObservableCollection<PolarPoint> controlPoints;
        private readonly PolarSplineCalculator obstaclePointCalculator;
        private bool editingObstacles;
        private Point obstacleCentre;

        private readonly int numObstaclePoints = 80;


        public string? CurrentButton // Conversion between string and internal enum
        ↪   value done in property
        {
            get
            {
                if (currentButton == null) return null;
                return Enum.GetName(typeof(SidePanelButton), currentButton);
            }
            set
            {
                if (value == null)
                {
                    currentButton = null;
                }
                else
                {
                    currentButton =
                     ↪  (SidePanelButton)Enum.Parse(typeof(SidePanelButton), value);
                }
                OnPropertyChanged(this, nameof(currentButton));
            }
        }

        public float InVel
        {
            get => parameterHolder.InflowVelocity.Value;
            set
            {
                parameterHolder.InflowVelocity =
                 ↪  ModifyParameterValue(parameterHolder.InflowVelocity, value);
                OnPropertyChanged(this, nameof(InVel));
            }
        }
        public float Chi
        {
            get => parameterHolder.SurfaceFriction.Value;
            set
            {
                parameterHolder.SurfaceFriction =
                 ↪  ModifyParameterValue(parameterHolder.SurfaceFriction, value);
                OnPropertyChanged(this, nameof(Chi));
            }
        }
        public float VisMin
        {
            get => parameterHolder.FieldParameters.Value.min;
            set
            {
                parameterHolder.FieldParameters =
                 ↪  ModifyParameterValue(parameterHolder.FieldParameters,
                 ↪  ModifyFieldParameters(parameterHolder.FieldParameters.Value,
                 ↪  null, value, null));
                OnPropertyChanged(this, nameof(VisMin));
            }
        }
        public float VisMax
        {
```

```
        get => parameterHolder.FieldParameters.Value.max;
        set
        {
            parameterHolder.FieldParameters =
            ↪  ModifyParameterValue(parameterHolder.FieldParameters,
            ↪  ModifyFieldParameters(parameterHolder.FieldParameters.Value,
            ↪  null, null, value));
            OnPropertyChanged(this, nameof(VisMax));
        }
    }
    public float VisLowerBound
    {
        get => visLowerBound;
        private set
        {
            visLowerBound = value;
            OnPropertyChanged(this, nameof(VisLowerBound));
        }
    }
    public float VisUpperBound
    {
        get => visUpperBound;
        private set
        {
            visUpperBound = value;
            OnPropertyChanged(this, nameof(VisUpperBound));
        }
    }
    public float NumContours
    {
        get => parameterHolder.NumContours.Value;
        set
        {
            parameterHolder.NumContours =
            ↪  ModifyParameterValue(parameterHolder.NumContours, value);
            OnPropertyChanged(this, nameof(NumContours));
        }
    }
    public float ContourTolerance
    {
        get => parameterHolder.ContourTolerance.Value;
        set
        {
            parameterHolder.ContourTolerance =
            ↪  ModifyParameterValue(parameterHolder.ContourTolerance, value);
            OnPropertyChanged(this, nameof(ContourTolerance));
        }
    }
    public bool PressureChecked
    {
        get { return selectedField == SelectedField.Pressure; }
        set
        {
            if (value)
            {
                selectedField = SelectedField.Pressure;
            }
            else
            {
                selectedField = SelectedField.Velocity;
            }

            OnPropertyChanged(this, nameof(PressureChecked));
            SwitchFieldParameters();
        }
    }
    public bool VelocityChecked
    {
        get { return selectedField == SelectedField.Velocity; }
        set
        {
            if (value) selectedField = SelectedField.Velocity;
            else selectedField = SelectedField.Pressure;
            OnPropertyChanged(this, nameof(VelocityChecked));
            SwitchFieldParameters();
        }
    }
    public bool StreamlinesEnabled
    {
        get => parameterHolder.DrawContours.Value;
        set
        {
            parameterHolder.DrawContours =
            ↪  ModifyParameterValue(parameterHolder.DrawContours, value);
            OnPropertyChanged(this, nameof(StreamlinesEnabled));
        }
    }

    public bool EditingObstacles
    {
        get => editingObstacles;
        set
        {
            editingObstacles = value;
            OnPropertyChanged(this, nameof(EditingObstacles));
            OnPropertyChanged(this, nameof(EditObstaclesButtonText));
        }
    }
    public ObservableCollection<PolarPoint> ObstaclePoints { get =>
    ↪   obstaclePoints; }
    public ObservableCollection<PolarPoint> ControlPoints { get =>
    ↪   controlPoints; }
    public Point ObstacleCentre
    {
        get => obstacleCentre;
        set
        {
            obstacleCentre = value;
            OnPropertyChanged(this, nameof(ObstacleCentre));
        }
    }
    public ObservableCollection<ObstacleCell> ObstacleCells { get; private set;
    ↪  }

    public VisualisationControl VisualisationControl { get =>
     ↪  visualisationControl; }
    public UnitConversionPanel UnitsPanel { get => unitsPanel; }

    public float VisFPS { get => 1 / visFrameTimeAverage.Average; }
    public float BackFPS { get => 1 / backFrameTimeAverage.Average; }
    public float DragCoefficient { get => dragCoefficientAverage.Average; }

    public CancellationTokenSource BackendCTS { get => backendCTS; set =>
     ↪  backendCTS = value; }

    public string BackendButtonText
    {
        get
        {
            return BackendStatus switch
            {
                BackendStatus.Running => "Pause simulation",
                BackendStatus.Stopped => "Resume simulation",
                _ => string.Empty,
            };
        }
    }

    public string EditObstaclesButtonText
    {
        get
        {
            return EditingObstacles ? "Finish editing" : "Edit simulation
             ↪  obstacles";
        }
    }

    public BackendStatus BackendStatus
    {
        get => backendManager.BackendStatus;
    }

    public Commands.SwitchPanel SwitchPanelCommand { get; set; }
    public Commands.PauseResumeBackend BackendCommand { get; set; }
    public Commands.ChangeWindow ChangeWindowCommand { get; set; }
    public Commands.CreatePopup CreatePopupCommand { get; set; }
    public Commands.EditObstacles EditObstaclesCommand { get; set; }

    public Commands.SimScreenBack BackCommand { get; set; }
```

```
private enum SidePanelButton // Different side panels on SimluationScreen
{
    BtnParametersSelect,
    BtnUnitsSelect,
    BtnVisualisationSettingsSelect,
    BtnRecordingSelect
}
private enum SelectedField
{
    Pressure,
    Velocity
}

public event CancelEventHandler StopBackendExecuting;
#endregion

public SimulationScreenVM(ParameterHolder parameterHolder, UnitHolder
↪   unitHolder, ObstacleHolder obstacleHolder) : base(parameterHolder,
↪   unitHolder, obstacleHolder)
{
    #region Parameters related to View
    currentButton = null; // Initially no panel selected
    obstaclePoints = [];
    controlPoints = [];
    obstacleCentre = new Point(50, 50);
    obstaclePointCalculator = new PolarSplineCalculator();
    ObstacleCells = new ObservableCollection<ObstacleCell>();
    if (!obstacleHolder.UsingObstacleFile)
    {
        CreateDefaultObstacle();
    }

    controlPoints.CollectionChanged += OnControlPointsChanged;
    editingObstacles = false;
    InVel = parameterHolder.InflowVelocity.Value;
    Chi = parameterHolder.SurfaceFriction.Value;

    unitsPanel = new UnitConversionPanel(unitHolder);
    SwitchPanelCommand = new Commands.SwitchPanel(this);
    BackendCommand = new Commands.PauseResumeBackend(this);
    EditObstaclesCommand = new Commands.EditObstacles(this);
    BackCommand = new Commands.SimScreenBack(this);
    ChangeWindowCommand = new Commands.ChangeWindow();
    CreatePopupCommand = new Commands.CreatePopup();
    RecToPolConverter = new RectangularToPolar();
    #endregion

    #region Parameters related to Backend
    backendCTS = new CancellationTokenSource();
    StopBackendExecuting += (object? sender, CancelEventArgs e) =>
    ↪   backendCTS.Cancel();
```

```
if (obstacleHolder.UsingObstacleFile)
{
    try
    {
        obstacleHolder.ReadObstacleFile();
    }
    catch (FileNotFoundException e)
    {
        MessageBox.Show(e.Message + "Reverting to drawable obstacles.",
        ↪  "Error: obstacle file not found.");
        obstacleHolder.UsingObstacleFile = false;
    }
    catch (FileFormatException e)
    {
        MessageBox.Show(e.Message + "Reverting to drawable obstacles.",
        ↪  "Error: malformed obstacle file.");
        obstacleHolder.UsingObstacleFile = false;
    }
}

backendManager = new BackendManager(parameterHolder);
bool connectionSuccess =
↪  backendManager.ConnectBackend(obstacleHolder.DataWidth,
↪  obstacleHolder.DataHeight);
if (!connectionSuccess)
{
    MessageBox.Show("Backend did not connect properly.", "ERROR: Backend
    ↪  did not connect properly");
    throw new IOException("Backend did not connect properly.");
}

backendManager.SendAllParameters();
velocity = new float[backendManager.FieldLength];
pressure = new float[backendManager.FieldLength];
streamFunction = new float[backendManager.FieldLength];
dataWidth = backendManager.IMax;
dataHeight = backendManager.JMax;

if (obstacleHolder.UsingObstacleFile)
{
    _ = backendManager.SendObstacles(obstacleHolder.ObstacleData!);
    CoverObstacleCells();
}
else
{
    EmbedObstacles();
}

Task.Run(StartComputation);
backFrameTimeAverage = new
↪  MovingAverage<float>(DefaultParameters.FPS_WINDOW_SIZE);
dragCoefficientAverage = new
↪  MovingAverage<float>(DefaultParameters.DRAG_COEF_WINDOW_SIZE);
```

```
        backendManager.PropertyChanged += HandleBackendPropertyChanged;
        #endregion

        #region Parameters related to Visualisation
        SetFieldDefaults();
        selectedField = SelectedField.Velocity; // Velocity selected initially.
        SwitchFieldParameters();

        visualisationControl = new VisualisationControl(parameterHolder,
        ↪  streamFunction, dataWidth, dataHeight); // Content of
        ↪  VisualisationControlHolder is bound to this.
        visualisationControl.PropertyChanged += VisFPSUpdate; // FPS updating
        ↪  method
        visFrameTimeAverage = new
        ↪  MovingAverage<float>(DefaultParameters.FPS_WINDOW_SIZE);
        #endregion
    }

    private void OnControlPointsChanged(object? sender,
    ↪  NotifyCollectionChangedEventArgs e)
    {
        switch (e.Action)
        {
            case NotifyCollectionChangedAction.Add:
                foreach (object? point in e.NewItems)
                {
                    if (point is not PolarPoint polarPoint)
                    {
                        throw new ArgumentException("The item added to the
                        ↪  collection was not valid.");
                    }
                    obstaclePointCalculator.AddControlPoint(polarPoint);
                }
                break;

            case NotifyCollectionChangedAction.Remove:
                foreach (object? point in e.OldItems)
                {
                    if (point is not PolarPoint polarPoint)
                    {
                        throw new ArgumentException("The item removed from the
                        ↪  collection was not valid");
                    }
                    obstaclePointCalculator.RemoveControlPoint(polarPoint);
                }
                break;

            case NotifyCollectionChangedAction.Replace:
                if (e.OldItems[0] is not PolarPoint oldPoint || e.NewItems[0] is
                ↪  not PolarPoint newPoint)
                {
                    throw new ArgumentException("The item removed from the
                    ↪  collection was not valid");
                }
                obstaclePointCalculator.ModifyControlPoint(oldPoint, newPoint);
                ↪  // Check if NewItems contains the new item here.
                break;
            default:
                throw new InvalidOperationException("Only add, modify and remove
                ↪  are supported for obstacle points collection.");
        }
        // If control has reached this point, a valid modification has been made
        ↪  to the control points collection
        PlotObstaclePoints();
    }

    private void SetFieldDefaults()
    {
        velocityParameters.field = velocity;
        velocityParameters.min = DefaultParameters.VELOCITY_MIN;
        velocityParameters.max = DefaultParameters.VELOCITY_MAX;
        pressureParameters.field = pressure;
        pressureParameters.min = DefaultParameters.PRESSURE_MIN;
        pressureParameters.max = DefaultParameters.PRESSURE_MAX;
    }

    private void SwitchFieldParameters()
    {
        if (selectedField == SelectedField.Pressure)
        {
            parameterHolder.FieldParameters =
            ↪  ModifyParameterValue(parameterHolder.FieldParameters,
            ↪  pressureParameters);
            VisLowerBound = DefaultParameters.PRESSURE_MIN;
            VisUpperBound = DefaultParameters.PRESSURE_MAX;
        }
        else // Velocity selected
        {
            parameterHolder.FieldParameters =
            ↪  ModifyParameterValue(parameterHolder.FieldParameters,
            ↪  velocityParameters);
            VisLowerBound = DefaultParameters.VELOCITY_MIN;
            VisUpperBound = DefaultParameters.VELOCITY_MAX;
        }
        VisMin = parameterHolder.FieldParameters.Value.min;
        VisMax = parameterHolder.FieldParameters.Value.max;


    }

    private static FieldParameters ModifyFieldParameters(FieldParameters
    ↪  fieldParameters, float[]? newField, float? newMin, float? newMax)
    {
        if (newField is not null)
        {
            fieldParameters.field = newField;
        }
        if (newMin is not null)
        {
            fieldParameters.min = (float)newMin;
        }
        if (newMax is not null)
        {
            fieldParameters.max = (float)newMax;
        }
        return fieldParameters;
    }

    public void StartComputation()
    {
        try
        {
            backendCTS = new CancellationTokenSource();
            backendManager.GetFieldStreamsAsync(velocity, null, pressure,
            ↪  streamFunction, backendCTS.Token);
        }
        catch (IOException e)
        {
            MessageBox.Show(e.Message, "ERROR");
        }
        catch (Exception e)
        {
            MessageBox.Show($"Generic error: {e.Message}", "ERROR");
        }
    }

    private void CreateDefaultObstacle()
    {
        double scale = 10;

        // Define the bean.
        controlPoints.Add(new PolarPoint(scale, Math.PI / 4));
        controlPoints.Add(new PolarPoint(scale, 3 * Math.PI / 4));
        controlPoints.Add(new PolarPoint(scale, 5 * Math.PI / 4));
        controlPoints.Add(new PolarPoint(scale, 7 * Math.PI / 4));
        controlPoints.Add(new PolarPoint(scale / Math.Sqrt(2), Math.PI / 2));

        foreach (PolarPoint controlPoint in controlPoints)
        {
            obstaclePointCalculator.AddControlPoint(controlPoint);
        }

        PlotObstaclePoints();
    }

    /// <summary>
    /// Wipes the <see cref="ObstaclePoints"/> collection and plots new points
    ↪   based on the <see cref="obstaclePointCalculator"/> function.
    /// </summary>
```

```
private void PlotObstaclePoints()
{
    ObstaclePoints.Clear();
    for (double i = 0; i < numObstaclePoints; i++)
    {
        ObstaclePoints.Add(new
        ↪   PolarPoint(obstaclePointCalculator.CalculatePoint(i /
        ↪   numObstaclePoints * 2 * Math.PI), i / numObstaclePoints * 2 *
        ↪   Math.PI));
    }
    OnPropertyChanged(this, nameof(ObstaclePoints));
}

private void CoverObstacleCells()
{
    float cellWidth = (float)CANVAS_WIDTH / backendManager.IMax;
    float cellHeight = (float)CANVAS_HEIGHT / backendManager.JMax;
    for (int i = 1; i <= backendManager.IMax; i++)
    {
        for (int j = 1; j <= backendManager.JMax; j++)
        {
            if (!obstacleHolder.ObstacleData[i * (obstacleHolder.DataHeight
            ↪   + 2) + j]) // Obstacle cells
            {
                ObstacleCells.Add(new ObstacleCell
                {
                    X = (i - 1) * cellWidth,
                    Y = (j - 1) * cellHeight,
                    Width = cellWidth,
                    Height = cellHeight
                });
            }
        }
    }
}

public void EmbedObstacles()
{
    bool[] obstacles = new bool[(dataWidth + 2) * (dataHeight + 2)];
    for (int i = 1; i <= dataWidth; i++)
    {
        for (int j = 1; j <= dataHeight; j++)
        {
            obstacles[i * (dataHeight + 2) + j] = true; // Set cells to
            ↪   fluid
        }
    }

    for (int i = 1; i <= dataWidth; i++)
    {
        for (int j = 1; j <= dataHeight; j++)
        {
            float screenX = i * (float)CANVAS_WIDTH / dataWidth;
```

```
                float screenY = j * (float)CANVAS_HEIGHT / dataHeight;
                PolarPoint polarPoint =
                ↪   (PolarPoint)RecToPolConverter.Convert(new Point(screenX,
                ↪   screenY), typeof(PolarPoint), ObstacleCentre,
                ↪   System.Globalization.CultureInfo.CurrentCulture);
                if (polarPoint.Radius <
                ↪   obstaclePointCalculator.CalculatePoint(polarPoint.Angle)) //
                ↪   Within the obstacle
                {
                    obstacles[i * (dataHeight + 2) + j] = false; // Set cells to
                    ↪   obstacle
                }
            }
        }
        _ = backendManager.SendObstacles(obstacles);
    }

    public void CloseBackend()
    {
        if (!backendManager.CloseBackend())
        {
            backendManager.ForceCloseBackend();
        }
    }

    private void VisFPSUpdate(object? sender, PropertyChangedEventArgs e)
    {
        visFrameTimeAverage.UpdateAverage(visualisationControl.FrameTime);
        OnPropertyChanged(this, nameof(VisFPS));
    }

    private void BackFPSUpdate()
    {
        backFrameTimeAverage.UpdateAverage(backendManager.FrameTime);
        OnPropertyChanged(this, nameof(BackFPS));
    }

    private void DragCoefficientUpdate()
    {
        dragCoefficientAverage.UpdateAverage(backendManager.DragCoefficient);
        OnPropertyChanged(this, nameof(DragCoefficient));
    }

    private void HandleBackendPropertyChanged(object? sender,
    ↪   PropertyChangedEventArgs e)
    {
        if (e.PropertyName == nameof(backendManager.BackendStatus))
        {
            OnPropertyChanged(sender, nameof(BackendStatus));
            OnPropertyChanged(this, nameof(BackendButtonText));
        }
        else if (e.PropertyName == nameof(backendManager.FrameTime))
        {
                BackFPSUpdate();
            }
            else if (e.PropertyName == nameof(backendManager.DragCoefficient))
            {
                DragCoefficientUpdate();
            }
        }
    }
}
```

ViewModel.cs

```
using System.ComponentModel;
using UserInterface.Converters;
using UserInterface.HelperClasses;

namespace UserInterface.ViewModels
{
    /// <summary>
    /// Abstract parent class for view models, providing functionality common to all
    ↪  view models.
    /// </summary>
    public abstract class ViewModel : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler? PropertyChanged;

        protected void OnPropertyChanged(object? sender, string propertyName)
        {
            PropertyChanged?.Invoke(sender, new
            ↪  PropertyChangedEventArgs(propertyName));
        }

        protected ParameterHolder? parameterHolder;
        protected UnitHolder? unitHolder;
        protected ObstacleHolder? obstacleHolder;

        protected ParameterStruct<T> ModifyParameterValue<T>(ParameterStruct<T>
        ↪  parameterStruct, T newValue)
        {
            parameterStruct.Value = newValue;
            return parameterStruct;
        }

        public UnitHolder UnitHolder { get => unitHolder!; set => unitHolder =
        ↪  value; }
        public ParameterHolder ParameterHolder { get => parameterHolder!; set =>
        ↪  parameterHolder = value; }
        public ObstacleHolder ObstacleHolder { get => obstacleHolder!; set =>
        ↪  obstacleHolder = value; }

        public UnitClasses.Dimensionless DimensionlessUnit { get; } = new
        ↪  UnitClasses.Dimensionless();
        public UnitClasses.LengthUnit LengthUnit { get => UnitHolder.LengthUnit; }
        public UnitClasses.SpeedUnit SpeedUnit { get => UnitHolder.SpeedUnit; }
        public UnitClasses.TimeUnit TimeUnit { get => UnitHolder.TimeUnit; }
        public UnitClasses.DensityUnit DensityUnit { get => UnitHolder.DensityUnit;
        ↪  }
        public UnitClasses.ViscosityUnit ViscosityUnit { get =>
        ↪  UnitHolder.ViscosityUnit; }

        public ViewModel() { }

        public ViewModel(ParameterHolder parameterHolder, UnitHolder unitHolder,
        ↪  ObstacleHolder obstacleHolder)
        {
            this.parameterHolder = parameterHolder;
            this.unitHolder = unitHolder;
            this.obstacleHolder = obstacleHolder;
            unitHolder.PropertyChanged += OnUnitsChanged;
        }

        /// <summary>
        /// Handles property changed notifications generated by <see
        ↪  cref="UnitHolder"/>.
        /// </summary>
        private void OnUnitsChanged(object? sender, PropertyChangedEventArgs e)
        {
            switch (e.PropertyName)
            {
                case nameof(UnitHolder.LengthUnit):
                    OnPropertyChanged(this, nameof(LengthUnit));
                    break;
                case nameof(UnitHolder.SpeedUnit):
                    OnPropertyChanged(this, nameof(SpeedUnit));
                    break;
                case nameof(UnitHolder.TimeUnit):
                    OnPropertyChanged(this, nameof(TimeUnit));
                    break;
                case nameof(UnitHolder.DensityUnit):
                    OnPropertyChanged(this, nameof(DensityUnit));
                    break;
                case nameof(UnitHolder.ViscosityUnit):
                    OnPropertyChanged(this, nameof(ViscosityUnit));
                    break;
            }
        }
    }
}
```

AdvancedParameters.xaml

```
<UserControl x:Class="UserInterface.Views.AdvancedParameters"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             xmlns:local="clr-namespace:UserInterface.Views"
             xmlns:helpercontrols="clr-namespace:UserInterface.HelperControls"
```

```
            xmlns:viewModels="clr-namespace:UserInterface.ViewModels"
            mc:Ignorable="d"
            d:DataContext="{d:DesignInstance Type=viewModels:AdvancedParametersVM}"
            d:DesignHeight="400" d:DesignWidth="700">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>
        <helpercontrols:ResizableCentredTextBox Grid.Row="0" Grid.Column="0"
        ↪   Grid.ColumnSpan="2" Text="Advanced Parameters" />
        <Label Grid.Row="1" Grid.Column="0">Timestep Safety Factor &#964;</Label>
        <helpercontrols:SliderWithValue x:Name="sliderTau" Grid.Row="1"
        ↪   Grid.Column="1" Minimum="0" Maximum="1" Value="{Binding Tau}"
        ↪   Unit="{Binding DimensionlessUnit}"/>
        <Label Grid.Row="2" Grid.Column="0">SOR relaxation parameter &#969;</Label>
        <helpercontrols:SliderWithValue x:Name="sliderOmega" Grid.Row="2"
        ↪   Grid.Column="1" Minimum="0" Maximum="2" Value="{Binding Omega}"
        ↪   Unit="{Binding DimensionlessUnit}" />
        <Label Grid.Row="3" Grid.Column="0">Pressure residual tolerance r</Label>
        <helpercontrols:SliderWithValue x:Name="sliderRMax" Grid.Row="3"
        ↪   Grid.Column="1" Minimum="0" Maximum="100" Value="{Binding RMax}"
        ↪   Unit="{Binding DimensionlessUnit}" />
        <Label Grid.Row="4" Grid.Column="0">Maximum SOR iterations</Label>
        <helpercontrols:SliderWithValue x:Name="sliderIterMax" Grid.Row="4"
        ↪   Grid.Column="1" Minimum="0" Maximum="10000" ForceIntegers="True"
        ↪   Value="{Binding IterMax}" Unit="{Binding DimensionlessUnit}" />
        <Label Grid.Row="5" Grid.Column="0">Grid cell size (horizontal)</Label>
        <helpercontrols:SliderWithValue x:Name="sliderDelX" Grid.Row="5"
        ↪   Grid.Column="1" Minimum="0" Maximum="0.1" ForceIntegers="False"
        ↪   Value="{Binding DelX}" Unit="{Binding LengthUnit}" IsEnabled="{Binding
        ↪   CanChangeGridSizes, Mode=OneWay}" />
        <Label Grid.Row="6" Grid.Column="0">Grid cell size (vertical)</Label>
        <helpercontrols:SliderWithValue x:Name="sliderDelY" Grid.Row="6"
        ↪   Grid.Column="1" Minimum="0" Maximum="0.1" ForceIntegers="False"
        ↪   Value="{Binding DelY}" Unit="{Binding LengthUnit}" IsEnabled="{Binding
        ↪   CanChangeGridSizes, Mode=OneWay}" />
        <Button x:Name="ResetButton" Grid.Row="7" Grid.Column="0" FontSize="16"
        ↪   Command="{Binding ResetCommand}">Reset values</Button>
        <Button x:Name="SaveButton" Grid.Row="7" Grid.Column="1" FontSize="16"
        ↪   Command="{Binding SaveCommand}">Save and return</Button>
    </Grid>
</UserControl>
```

AdvancedParameters.xaml.cs

```
using System.Windows.Controls;
using UserInterface.HelperClasses;
using UserInterface.ViewModels;

namespace UserInterface.Views
{
    /// <summary>
    /// Interaction logic for AdvancedParameters.xaml
    /// </summary>
    public partial class AdvancedParameters : UserControl
    {
        public AdvancedParameters(ParameterHolder parameterHolder, UnitHolder
        ↪  unitHolder, ObstacleHolder obstacleHolder)
        {
            InitializeComponent();
            DataContext = new AdvancedParametersVM(parameterHolder, unitHolder,
            ↪  obstacleHolder);
        }
    }
}
```

ConfigScreen.xaml

```
<UserControl x:Class="UserInterface.Views.ConfigScreen"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             xmlns:local="clr-namespace:UserInterface.Views"
             xmlns:helpercontrols="clr-namespace:UserInterface.HelperControls"
             xmlns:viewmodels="clr-namespace:UserInterface.ViewModels"
             mc:Ignorable="d"
             d:DataContext="{d:DesignInstance Type=viewmodels:ConfigScreenVM}"
             d:DesignHeight="630" d:DesignWidth="1120">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="2*" MaxHeight="100" />
            <RowDefinition Height="*" MaxHeight="50" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" MaxHeight="50" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="0.1*" />
            <RowDefinition Height="*" MaxHeight="50" />
        </Grid.RowDefinitions>
```

```
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto" />
    <ColumnDefinition Width="6*"/>
    <ColumnDefinition Width="Auto" />
    <ColumnDefinition Width="*" />
</Grid.ColumnDefinitions>

<helpercontrols:ResizableCentredTextBox Grid.Row="0" Grid.Column="0"
↪   Grid.ColumnSpan="3" Text="Simulation configuration" />
<helpercontrols:ResizableCentredTextBox Grid.Row="1" Grid.Column="0"
↪   Grid.ColumnSpan="2" Text="Parameters" />
<helpercontrols:ResizableCentredTextBox Grid.Row="1" Grid.Column="2"
↪   Grid.ColumnSpan="2" Text="Units" />

<Label Grid.Row="2" Grid.Column="0">Flow velocity</Label>
<helpercontrols:SliderWithValue x:Name="SliderInVel" Grid.Row="2"
↪   Grid.Column="1" Minimum="0" Maximum="40" Value="{Binding InVel}"
↪   Unit="{Binding SpeedUnit}" />

<Label Grid.Row="3" Grid.Column="0">Material Friction</Label>
<helpercontrols:SliderWithValue x:Name="SliderChi" Grid.Row="3"
↪   Grid.Column="1" Minimum="0" Maximum="1" Value="{Binding Chi}"
↪   Unit="{Binding DimensionlessUnit}" />

<Label Grid.Row="4" Grid.Column="0">Simulation width</Label>
<helpercontrols:SliderWithValue x:Name="SliderWidth" Grid.Row="4"
↪   Grid.Column="1" Minimum="0" Maximum="5" Value="{Binding Width}"
↪   Unit="{Binding LengthUnit}" />

<Label Grid.Row="5" Grid.Column="0">Simulation height</Label>
<helpercontrols:SliderWithValue x:Name="SliderHeight" Grid.Row="5"
↪   Grid.Column="1" Minimum="0" Maximum="5" Value="{Binding Height}"
↪   Unit="{Binding LengthUnit}" />

<Button x:Name="BtnReset" Grid.Row="6" Grid.Column="0" Grid.ColumnSpan="2"
↪   Command="{Binding ResetCommand}">Reset parameters</Button>

<Button Grid.Row="7" Grid.Column="0" Grid.ColumnSpan="2" Command="{Binding
↪   CreatePopupCommand}" CommandParameter="{x:Type
↪   local:AdvancedParameters}">Advanced parameters</Button>
<helpercontrols:ResizableCentredTextBox Grid.Row="8" Grid.Column="0"
↪   Grid.ColumnSpan="2" MaxHeight="50" Text="Fluid parameters" />
<Label Grid.Row="9" Grid.Column="0">Reynolds number</Label>
<helpercontrols:SliderWithValue x:Name="SliderReynoldsNo" Grid.Row="9"
↪   Grid.Column="1" Value="{Binding ReynoldsNo}" Minimum="1000"
↪   Maximum="100000" Unit="{Binding DimensionlessUnit}" />
<Label Grid.Row="10" Grid.Column="0">Viscosity</Label>
<helpercontrols:SliderWithValue x:Name="SliderViscosity" Grid.Row="10"
↪   Grid.Column="1" Value="{Binding Viscosity}" Minimum="2E-5" Maximum="1"
↪   Unit="{Binding ViscosityUnit}" />
<Label Grid.Row="11" Grid.Column="0">Density</Label>
```

```
        <helpercontrols:SliderWithValue x:Name="SliderDensity" Grid.Row="11"
        ↪  Grid.Column="1" Value="{Binding Density}" Minimum="0" Maximum="10"
        ↪  Unit="{Binding DensityUnit}" />
        <Button Grid.Row="12" Grid.Column="0" Grid.ColumnSpan="2" Command="{Binding
        ↪  SetAirCommand}">Reset to air at room temperature (20°C)</Button>
        <ContentControl Grid.Row="2" Grid.Column="2" Grid.RowSpan="6"
        ↪  Grid.ColumnSpan="2" Content="{Binding UnitsPanel}" />
        <helpercontrols:ResizableCentredTextBox Grid.Row="8" Grid.Column="2"
        ↪  Grid.ColumnSpan="2" MaxHeight="50" Text="Obstacle file" />
        <Label Grid.Row="9" Grid.Column="2" VerticalAlignment="Center" Margin="5 0 0
        ↪  0">Use obstacles from file</Label>
        <CheckBox Grid.Row="9" Grid.Column="3" VerticalAlignment="Center"
        ↪  IsChecked="{Binding UsingObstacleFile}" d:IsChecked="True" />
        <Button Grid.Row="10" Grid.Column="2" Grid.ColumnSpan="2" Margin="5 0 0 0"
        ↪  Command="{Binding SelectObstacleFileCommand}">Select file</Button>
        <TextBlock Grid.Row="11" Grid.Column="2" Grid.ColumnSpan="2" Margin="5 5 5
        ↪  5" Text="{Binding DisplayFileName}" d:Text="No file selected"
        ↪  TextWrapping="Wrap" />
        <Button Grid.Row="14" Grid.Column="2" Grid.ColumnSpan="2" Command="{Binding
        ↪  TrySimulateCommand}" Margin="5 0 0 0">Simulate</Button>
    </Grid>
</UserControl>
```

ConfigScreen.xaml.cs

```
using System.Windows.Controls;
using UserInterface.HelperClasses;
using UserInterface.ViewModels;

namespace UserInterface.Views
{
    /// <summary>
    /// Interaction logic for ConfigScreen.xaml
    /// </summary>
    public partial class ConfigScreen : UserControl
    {
        public ConfigScreen(ParameterHolder parameterHolder, UnitHolder unitHolder,
        ↪  ObstacleHolder obstacleHolder)
        {
            InitializeComponent();
            DataContext = new ConfigScreenVM(parameterHolder, unitHolder,
            ↪  obstacleHolder);
        }
    }
}
```

MainWindow.xaml

```
<Window x:Class="UserInterface.Views.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:UserInterface.Views"
        mc:Ignorable="d"
        Title="Fluid Dynamics Sim"
        WindowState="Maximized">
</Window>
```

MainWindow.xaml.cs

```
using System.Windows;

namespace UserInterface.Views
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
    }
}
```

PopupWindow.xaml

```
<Window x:Class="UserInterface.Views.PopupWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:UserInterface.Views"
        mc:Ignorable="d"
        Title="Fluid Dynamics Sim">
</Window>
```

PopupWindow.xaml.cs

```
using System.Windows;

namespace UserInterface.Views
{
    /// <summary>
    /// Interaction logic for PopupWindow.xaml
    /// </summary>
    public partial class PopupWindow : Window
    {
        public PopupWindow()
        {
            InitializeComponent();
        }
    }
}
```

SimulationScreen.xaml

```
<UserControl x:Class="UserInterface.Views.SimulationScreen"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             xmlns:local="clr-namespace:UserInterface.Views"
             xmlns:helpercontrols="clr-namespace:UserInterface.HelperControls"
             xmlns:diag="clr-namespace:System.Diagnostics;assembly=WindowsBase"

             ↪  xmlns:viewmodels="clr-namespace:UserInterface.ViewModels;assembly=UserInterfa
             xmlns:converters="clr-namespace:UserInterface.Converters"
             d:DataContext="{d:DesignInstance Type=viewmodels:SimulationScreenVM}"
             mc:Ignorable="d"
             d:DesignHeight="720" d:DesignWidth="1280">
    <UserControl.Resources>
        <ResourceDictionary>
            <converters:SignificantFigures x:Key="SignificantFiguresConverter" />
            <converters:PolarListToRectList x:Key="PolarListToRectListConverter" />
        </ResourceDictionary>
    </UserControl.Resources>
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" MaxHeight="150"/>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" MaxWidth="50" />
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="8*" />
        </Grid.ColumnDefinitions>

        <helpercontrols:ResizableCentredTextBox Grid.Row="0" Grid.Column="0"
         ↪  Grid.ColumnSpan="3" Text="Simulating" MaxWidth="300" />

        <StackPanel Grid.Row="1" Grid.Column="0">
            <StackPanel.Resources>
                <Style TargetType="Button">
                    <Setter Property="Margin" Value="0 0 0 0" />
                </Style>
            </StackPanel.Resources>
            <Button x:Name="BtnParametersSelect" Command="{Binding
            ↪  SwitchPanelCommand}" CommandParameter="{Binding
            ↪  RelativeSource={RelativeSource Self}}">
                <Button.Style>
                    <Style TargetType="Button">
                        <Setter Property="Background" Value="LightGray" />
                        <Style.Triggers>
                            <DataTrigger Binding="{Binding CurrentButton}"
                             ↪  Value="BtnParametersSelect">
                                <Setter Property="Background" Value="#AAAAAA" />
                            </DataTrigger>
                        </Style.Triggers>
                    </Style>
                </Button.Style>
                <Image Source="/Images/ParametersIcon.png" />
            </Button>
            <Button x:Name="BtnUnitsSelect" Command="{Binding SwitchPanelCommand}"
             ↪   CommandParameter="{Binding RelativeSource={RelativeSource Self}}">
                <Button.Style>
                    <Style TargetType="Button">
                        <Setter Property="Background" Value="LightGray" />
                        <Style.Triggers>
                            <DataTrigger Binding="{Binding CurrentButton}"
                             ↪   Value="BtnUnitsSelect">
                                <Setter Property="Background" Value="#AAAAAA" />
                            </DataTrigger>
                        </Style.Triggers>
                    </Style>
                </Button.Style>
                <Image Source="/Images/UnitsIcon.png" />
            </Button>
            <Button x:Name="BtnVisualisationSettingsSelect" Command="{Binding
                ↪   SwitchPanelCommand}" CommandParameter="{Binding
                ↪   RelativeSource={RelativeSource Self}}">
                <Button.Style>
                    <Style TargetType="Button">
                        <Setter Property="Background" Value="LightGray" />
                        <Style.Triggers>
                            <DataTrigger Binding="{Binding CurrentButton}"
                             ↪   Value="BtnVisualisationSettingsSelect">
                                <Setter Property="Background" Value="#AAAAAA" />
                            </DataTrigger>
                        </Style.Triggers>
                    </Style>
                </Button.Style>
                <Image Source="/Images/VisualisationIcon.png" />
            </Button>
        </StackPanel>

        <StackPanel Grid.Row="2" Grid.Column="1">
            <StackPanel.Style>
                <Style TargetType="StackPanel">
                    <Setter Property="Visibility" Value="Collapsed" />
                    <Style.Triggers>
                        <DataTrigger Binding="{Binding CurrentButton}"
                         ↪   Value="BtnParametersSelect">
                            <Setter Property="Visibility" Value="Visible" />
                        </DataTrigger>
                    </Style.Triggers>
                </Style>
            </StackPanel.Style>
            <Label>Fluid Velocity</Label>
            <helpercontrols:SliderWithValue x:Name="SliderInVel" Minimum="0"
             ↪   Maximum="18" Value="{Binding InVel}" Unit="{Binding SpeedUnit}" />
```

```
            <Label>Material Friction</Label>
            <helpercontrols:SliderWithValue x:Name="SliderChi" Minimum="0"
            ↪  Maximum="1" Value="{Binding Chi}" Unit="{Binding DimensionlessUnit}"
            ↪  />
        </StackPanel>
        <ContentControl Grid.Row="2" Grid.Column="1" Content="{Binding UnitsPanel}">
            <ContentControl.Style>
                <Style TargetType="ContentControl">
                    <Setter Property="Visibility" Value="Collapsed" />
                    <Style.Triggers>
                        <DataTrigger Binding="{Binding CurrentButton}"
                        ↪  Value="BtnUnitsSelect">
                            <Setter Property="Visibility" Value="Visible" />
                        </DataTrigger>
                    </Style.Triggers>
                </Style>
            </ContentControl.Style>
        </ContentControl>
        <StackPanel Grid.Row="2" Grid.Column="1">
            <StackPanel.Style>
                <Style TargetType="StackPanel">
                    <Setter Property="Visibility" Value="Collapsed" />
                    <Style.Triggers>
                        <DataTrigger Binding="{Binding CurrentButton}"
                        ↪  Value="BtnVisualisationSettingsSelect">
                            <Setter Property="Visibility" Value="Visible" />
                        </DataTrigger>
                    </Style.Triggers>
                </Style>
            </StackPanel.Style>
            <Label>Field to visualise</Label>
            <RadioButton x:Name="RBPressure" GroupName="FieldSelector"
            ↪  IsChecked="{Binding PressureChecked}">Pressure</RadioButton>
            <RadioButton x:Name="RBVelocity" GroupName="FieldSelector"
            ↪  IsChecked="{Binding VelocityChecked}">Velocity</RadioButton>
            <Label>Minimum value</Label>
            <helpercontrols:SliderWithValue x:Name="SliderMin" Minimum="{Binding
            ↪  VisLowerBound, Mode=OneWay}" Maximum="{Binding VisUpperBound,
            ↪  Mode=OneWay}" Value="{Binding VisMin}"/>
            <Label>Maximum value</Label>
            <helpercontrols:SliderWithValue x:Name="SliderMax" Minimum="{Binding
            ↪  VisLowerBound, Mode=OneWay}" Maximum="{Binding VisUpperBound,
            ↪  Mode=OneWay}" Value="{Binding VisMax}" />
            <StackPanel Orientation="Horizontal">
                <Label Margin="0 0 10 0">Streamlines</Label>
                <CheckBox x:Name="CBContourLines" IsChecked="{Binding
                ↪  StreamlinesEnabled}" VerticalAlignment="Center" />
            </StackPanel>
            <Label>Number of streamlines</Label>
            <helpercontrols:SliderWithValue x:Name="SliderContourSpacing"
            ↪  Minimum="0" Maximum="100" ForceIntegers="True" Value="{Binding
            ↪  NumContours}" />
            <Label>Streamline contour tolerance</Label>
```

```
    <helpercontrols:SliderWithValue x:Name="SliderContourTolerance"
    ↪   Minimum="0" Maximum="0.05" Value="{Binding ContourTolerance}" />
</StackPanel>
<ContentControl Grid.Row="1" Grid.RowSpan="2" Grid.Column="2" Margin="10 0
↪   10 5" x:Name="VisualisationControlHolder" Content="{Binding
↪   VisualisationControl, Mode=OneWay}">
    <ContentControl.Style>
        <Style TargetType="ContentControl">
            <Setter Property="Opacity" Value="1" />
            <Style.Triggers>
                <DataTrigger Binding="{Binding EditingObstacles}"
                ↪   Value="True">
                    <Setter Property="Opacity" Value="0.5" />
                </DataTrigger>
            </Style.Triggers>
        </Style>
    </ContentControl.Style>
</ContentControl>
<Viewbox Grid.Row="1" Grid.RowSpan="2" Grid.Column="2" Margin="10 0 10 5"
↪   Stretch="Fill" StretchDirection="Both">
    <Canvas Width="100" Height="100" x:Name="SimulationCanvas"
    ↪   MouseLeftButtonDown="CanvasMouseLeftButtonDown"
    ↪   MouseRightButtonDown="CanvasMouseRightButtonDown"
    ↪   MouseLeftButtonUp="CanvasMouseLeftButtonUp"
    ↪   MouseMove="CanvasMouseMove">
        <Polygon x:Name="ObstaclePolygon" Fill="Black">
            <Polygon.Points>
                <MultiBinding Converter="{StaticResource
                ↪   PolarListToRectListConverter}">
                    <Binding Path="ObstaclePoints" />
                    <Binding Path="ObstacleCentre" />
                </MultiBinding>
            </Polygon.Points>
        </Polygon>
    </Canvas>
</Viewbox>
<Button Grid.Row="3" Grid.Column="0" Command="{Binding
↪   BackCommand}">Back</Button>
<StackPanel Grid.Row="3" Grid.Column="0" Grid.ColumnSpan="3"
↪   Orientation="Horizontal" HorizontalAlignment="Right">
    <StackPanel.Resources>
        <Style TargetType="Button">
            <Setter Property="FontSize" Value="16" />
            <Setter Property="Margin" Value="2.5 0 2.5 5" />
        </Style>
        <Style TargetType="Run">
            <Setter Property="FontSize" Value="16" />
        </Style>
    </StackPanel.Resources>
```

```
            <TextBlock Margin="0 0 5 0"><Run Text="Drag coefficient: " /><Run
            ↪   x:Name="RunDragCoefficient" Text="{Binding DragCoefficient,
            ↪   Converter={StaticResource SignificantFiguresConverter},
            ↪   ConverterParameter=2, Mode=OneWay}" d:Text="0" /><Run Text=",
            ↪   Visualisation: " /><Run x:Name="RunVisFPS" Text="{Binding VisFPS,
            ↪   Converter={StaticResource SignificantFiguresConverter},
            ↪   ConverterParameter=3, Mode=OneWay}" d:Text="0" /><Run Text=" FPS,
            ↪   Backend: " /><Run x:Name="RunBackFPS" Text="{Binding BackFPS,
            ↪   Converter={StaticResource SignificantFiguresConverter},
            ↪   ConverterParameter=3, Mode=OneWay}" d:Text="0" /><Run Text="
            ↪   FPS."/></TextBlock>
            <Button Command="{Binding EditObstaclesCommand}" Content="{Binding
            ↪   EditObstaclesButtonText}" d:Content="Edit simulation obstacles" />
            <Button Command="{Binding CreatePopupCommand}" CommandParameter="{x:Type
            ↪   local:AdvancedParameters}">Advanced parameters</Button>
            <Button Content="{Binding BackendButtonText}" Command="{Binding
            ↪   BackendCommand}" d:Content="Pause Simulation" />
        </StackPanel>
    </Grid>
</UserControl>
```

SimulationScreen.xaml.cs

```
using System;
using System.Collections.Specialized;
using System.ComponentModel;
using System.Diagnostics;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Shapes;
using UserInterface.Converters;
using UserInterface.HelperClasses;
using UserInterface.HelperControls;
using UserInterface.ViewModels;

namespace UserInterface.Views
{
    /// <summary>
    /// Interaction logic for SimulationScreen.xaml
    /// </summary>
    public partial class SimulationScreen : UserControl
    {
        private const float CELL_EXTRA_SIZE = 0.2f;

        private readonly SimulationScreenVM viewModel;
        private readonly RectangularToPolar RecToPolConverter;
        public SimulationScreenVM ViewModel => viewModel;

        private VisualPoint? draggedPoint;
        private bool isCentreMoved;
        private Point mousePosition;
        private bool pointsPlaced;
        private static readonly double POINT_TOLERANCE = 0.1f;

        public SimulationScreen(ParameterHolder parameterHolder, UnitHolder
         ↪  unitHolder, ObstacleHolder obstacleHolder)
        {
            InitializeComponent();

            viewModel = new SimulationScreenVM(parameterHolder, unitHolder,
             ↪  obstacleHolder);
            DataContext = viewModel;

            RecToPolConverter = new();
            pointsPlaced = false;
            isCentreMoved = false;
            ViewModel.PropertyChanged += OnViewModelPropertyChanged;
            if (obstacleHolder.UsingObstacleFile)
            {
                PlaceObstacleCells();
            }
        }

        #region Private helper methods
        private void PlaceObstacleCells()
        {
            foreach (ObstacleCell cell in ViewModel.ObstacleCells)
            {
                Rectangle rectangle = new Rectangle
                {
                    Width = cell.Width + CELL_EXTRA_SIZE,
                    Height = cell.Height + CELL_EXTRA_SIZE,
                    Fill = Brushes.Black,
                    Stroke = null
                };

                Canvas.SetLeft(rectangle, cell.X);
                Canvas.SetBottom(rectangle, cell.Y);
                SimulationCanvas.Children.Add(rectangle);
            }
        }

        private void PlaceInitialPoints()
        {
            foreach (PolarPoint polarPoint in ViewModel.ControlPoints)
            {
                SimulationCanvas.Children.Add(new
                 ↪  VisualPoint(ConvertToRect(polarPoint)));
            }
            SimulationCanvas.Children.Add(new
             ↪  VisualPoint(ViewModel.ObstacleCentre));
        }

        private void MoveControlPoints(Vector translation)
```

```
{
    for (int i = 0; i < SimulationCanvas.Children.Count; i++)
    {
        if (SimulationCanvas.Children[i] is VisualPoint point)
        {
            point.Point += translation;
        }
    }
}

private void AddPoint(VisualPoint point)
{
    SimulationCanvas.Children.Add(point);
    ViewModel.ControlPoints.Add(ConvertToPolar(point.Point));
}

private void RemovePoint(VisualPoint point)
{
    SimulationCanvas.Children.Remove(point);
    PolarPoint polarPoint = ConvertToPolar(point.Point);
    int polarPointIndex = FindIndexOfPolarPoint(polarPoint);
    ViewModel.ControlPoints.RemoveAt(polarPointIndex);
}

private int FindIndexOfPolarPoint(PolarPoint polarPoint)
{
    int polarPointIndex = ViewModel.ControlPoints.IndexOf(polarPoint);
    if (polarPointIndex == -1)
    {
        for (int i = 0; i < ViewModel.ControlPoints.Count; i++)
        {
            PolarPoint comparisonPoint = ViewModel.ControlPoints[i];
            if (Math.Abs(polarPoint.Radius - comparisonPoint.Radius) <
            ↪   POINT_TOLERANCE && Math.Abs(polarPoint.Angle -
            ↪   comparisonPoint.Angle) < POINT_TOLERANCE) // Allow some
            ↪   inexactness
            {
                polarPointIndex = i;
            }
        }
        if (polarPointIndex == -1) // If still not found
        {
            throw new InvalidOperationException("Could not find index of
            ↪   polar point.");
        }
    }

    return polarPointIndex;
}

private PolarPoint ConvertToPolar(Point rectangularPoint)
{
    return (PolarPoint)RecToPolConverter.Convert(rectangularPoint,
        ↪  typeof(PolarPoint), ViewModel.ObstacleCentre,
        ↪  System.Globalization.CultureInfo.CurrentCulture);
}

private Point ConvertToRect(PolarPoint polarPoint)
{
    return (Point)RecToPolConverter.ConvertBack(polarPoint,
        ↪  typeof(PolarPoint), ViewModel.ObstacleCentre,
        ↪  System.Globalization.CultureInfo.CurrentCulture);
}
#endregion

#region Event handlers
private void OnViewModelPropertyChanged(object? sender,
 ↪  PropertyChangedEventArgs e)
{
    switch (e.PropertyName)
    {
        case nameof(ViewModel.EditingObstacles):
            OnEditingObstalesChanged();
            break;
        default:
            break;
    }
}

private void OnEditingObstalesChanged()
{
    if (ViewModel.EditingObstacles && !pointsPlaced) // Place the points the
     ↪  first time app enters obstacle editing.
    {
        PlaceInitialPoints();
        pointsPlaced = true;
    }

    foreach (UIElement element in SimulationCanvas.Children) // Set the
        ↪  visibility of all of the control points on entering/leaving obstacle
        ↪  editing
    {
        if (element is VisualPoint)
        {
            element.Visibility = ViewModel.EditingObstacles ?
             ↪  Visibility.Visible : Visibility.Hidden;
        }
    }
}

private void CanvasMouseLeftButtonDown(object sender, MouseButtonEventArgs
 ↪  e)
{
    if (e.Source is VisualPoint point && SimulationCanvas.CaptureMouse())
    {
        mousePosition = e.GetPosition(SimulationCanvas);
        draggedPoint = point;
        draggedPoint.IsDragged = true;
        Trace.WriteLine($"difference in X coordinate:
        ↪   {Math.Abs(draggedPoint.Point.X - ViewModel.ObstacleCentre.X)}");
        Trace.WriteLine($"difference in Y coordinate:
        ↪   {Math.Abs(draggedPoint.Point.Y - ViewModel.ObstacleCentre.Y)}");
        isCentreMoved = Math.Abs(draggedPoint.Point.X -
        ↪   ViewModel.ObstacleCentre.X) < POINT_TOLERANCE &&
        ↪   Math.Abs(draggedPoint.Point.Y - ViewModel.ObstacleCentre.Y) <
        ↪   POINT_TOLERANCE;

        Panel.SetZIndex(draggedPoint, 1); // Make point go in front of
        ↪   everything else while is is dragged
    }
    else
    {
        Point clickPosition = e.GetPosition(SimulationCanvas); // Check
        ↪   whether mouse positions map correctly or not.
        AddPoint(new VisualPoint(clickPosition.X, 100 - clickPosition.Y));
    }
}

private void CanvasMouseLeftButtonUp(object sender, MouseButtonEventArgs e)
{
    if (draggedPoint is not null)
    {
        SimulationCanvas.ReleaseMouseCapture();
        Panel.SetZIndex(draggedPoint, 0);
        draggedPoint.IsDragged = false;
        draggedPoint = null;
        isCentreMoved = false;
    }
}

private void CanvasMouseRightButtonDown(object sender, MouseButtonEventArgs
↪   e)
{
    if (e.Source is VisualPoint point && point.Point !=
    ↪   ViewModel.ObstacleCentre)
    {
        RemovePoint(point);
    }
}

private void CanvasMouseMove(object sender, MouseEventArgs e)
{
    if (draggedPoint != null)
    {
        Point position = e.GetPosition(SimulationCanvas);
        Vector offset = position - mousePosition;
        mousePosition = position;
        Vector offsetYFlip = new Vector(offset.X, -offset.Y);
```

```
                if (!isCentreMoved) // Normal control points
                {
                    int draggedPointIndex =
                     ↪  FindIndexOfPolarPoint(ConvertToPolar(draggedPoint.Point));
                    draggedPoint.Point += offsetYFlip;
                    ViewModel.ControlPoints[draggedPointIndex] =
                     ↪  ConvertToPolar(draggedPoint.Point);
                }
                else
                {
                    MoveControlPoints(offsetYFlip);

                    ViewModel.ObstacleCentre = draggedPoint.Point;
                }
            }
        }
        #endregion
    }
}
```

VisualisationControl.xaml

```
<UserControl x:Class="UserInterface.VisualisationControl"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             xmlns:local="clr-namespace:UserInterface"
             xmlns:GLWPF="clr-namespace:OpenTK.Wpf;assembly=GLWpfControl"
             mc:Ignorable="d"
             d:DesignHeight="450" d:DesignWidth="800">
    <GLWPF:GLWpfControl x:Name="GLControl" Render="GLControl_OnRender" />
</UserControl>
```

VisualisationControl.xaml.cs

```
//#define HOLLOW_TRIANGLES

using OpenTK.Graphics.OpenGL4;
using OpenTK.Wpf;
using System;
using System.ComponentModel;
using System.Diagnostics;
using System.Windows.Controls;
using UserInterface.HelperClasses;
using Visualisation;

namespace UserInterface
{
    /// <summary>
    /// Interaction logic for VisualisationControl.xaml
    /// </summary>
    public partial class VisualisationControl : UserControl, INotifyPropertyChanged
    {
```

```
private readonly ShaderManager fieldShaderManager;
private readonly ShaderManager contourShaderManager;

private readonly float[] vertices;
private readonly uint[] fieldIndices;
private uint[] contourIndices;

private const uint primitiveRestartIndex = uint.MaxValue;

private int hVBO;
private int hFieldVAO;

private int hContourVAO;

private int hSubtrahend;
private int hScalar;

private ParameterHolder parameterHolder;
private float[] streamFunction;

private int dataWidth;
private int dataHeight;

private float frameTime;

public int DataWidth { get => dataWidth; set => dataWidth = value; }
public int DataHeight { get => dataHeight; set => dataHeight = value; }
public float[] StreamFunction { get => streamFunction; set => streamFunction
↪  = value; }
public float FrameTime { get => frameTime; }

public event PropertyChangedEventHandler? PropertyChanged;

public VisualisationControl(ParameterHolder parameterHolder, float[]
↪  streamFunction, int dataWidth, int dataHeight)
{
    this.parameterHolder = parameterHolder;
    this.streamFunction = streamFunction;
    this.dataWidth = dataWidth;
    this.dataHeight = dataHeight;

    InitializeComponent();
    DataContext = this;

    SetUpGL(out fieldShaderManager, out contourShaderManager, out vertices,
    ↪  out fieldIndices, out contourIndices); // Using out parameters so
    ↪  that these can be returned to the control of the constructor and not
    ↪  generate warnings
}

~VisualisationControl()
{
    fieldShaderManager.Dispose();
```

```
        contourShaderManager.Dispose();
    }


    private void SetUpGL(out ShaderManager fieldShaderManager, out ShaderManager
    ↪   contourShaderManager, out float[] vertices, out uint[] fieldIndices, out
    ↪   uint[] contourIndices)
    {
        GLWpfControlSettings settings = new() { MajorVersion = 3, MinorVersion =
        ↪   1 };
        GLControl.Start(settings);

        fieldShaderManager = new ShaderManager([("fieldShader.frag",
        ↪   ShaderType.FragmentShader), ("fieldShader.vert",
        ↪   ShaderType.VertexShader)]);
        contourShaderManager = new ShaderManager([("contourShader.frag",
        ↪   ShaderType.FragmentShader), ("contourShader.vert",
        ↪   ShaderType.VertexShader)]);

        GL.Enable(EnableCap.PrimitiveRestart);
        GL.PrimitiveRestartIndex(primitiveRestartIndex);

        HandleData(out vertices, out fieldIndices, out contourIndices);
    }

    private void HandleData(out float[] vertices, out uint[] fieldIndices, out
    ↪   uint[] contourIndices)
    {
        vertices = GLHelper.FillVertices(dataWidth, dataHeight);
        fieldIndices = GLHelper.FillIndices(dataWidth, dataHeight);
        contourIndices = GLHelper.FindContourIndices(streamFunction,
        ↪   parameterHolder.ContourTolerance.Value,
        ↪   parameterHolder.NumContours.Value, primitiveRestartIndex, dataWidth,
        ↪   dataHeight);

        FieldParameters fieldParameters = parameterHolder.FieldParameters.Value;

        // Setting up data for field visualisation
        hFieldVAO = GLHelper.CreateVAO();
        hVBO = GLHelper.CreateVBO(vertices.Length +
        ↪   fieldParameters.field.Length);

        GLHelper.BufferSubData(vertices, 0);
        GLHelper.BufferSubData(fieldParameters.field, vertices.Length);

        GLHelper.CreateAttribPointer(0, 2, 2, 0); // Vertex pointer
        GLHelper.CreateAttribPointer(1, 1, 1, vertices.Length); // Field value
        ↪   pointer

        _ = GLHelper.CreateEBO(fieldIndices); // EBO handle is never used
        ↪   because it is bound to the VAO

        // Setting up data for contour line plotting
```

```
            hContourVAO = GLHelper.CreateVAO();
            GL.BindBuffer(BufferTarget.ArrayBuffer, hVBO); // Bind the same VBO

            GLHelper.CreateAttribPointer(0, 2, 2, 0); // And the same for attribute
            ↪  pointers
            GLHelper.CreateAttribPointer(1, 1, 1, vertices.Length);

            _ = GLHelper.CreateEBO(contourIndices);

            // Return to field context
            GL.BindVertexArray(hFieldVAO);

            hSubtrahend = fieldShaderManager.GetUniformLocation("subtrahend");
            hScalar = fieldShaderManager.GetUniformLocation("scalar");
        }

        public void GLControl_OnRender(TimeSpan delta)
        {
            GL.Clear(ClearBufferMask.ColorBufferBit);
            FieldParameters fieldParameters = parameterHolder.FieldParameters.Value;
            ↪  // Get the most recent field parameters

            // For each draw command, need to bind the program, set uniforms, bind
            ↪  VAO, draw

            // Drawing field value spectrum
            fieldShaderManager.Use();

            fieldShaderManager.SetUniform(hSubtrahend, fieldParameters.min);
            fieldShaderManager.SetUniform(hScalar, 1 / (fieldParameters.max -
            ↪  fieldParameters.min));

            GL.BindVertexArray(hFieldVAO);
            GLHelper.BufferSubData(fieldParameters.field, vertices.Length); //
            ↪  Update the field values

#if HOLLOW_TRIANGLES
            GLHelper.Draw(fieldIndices, PrimitiveType.LineStrip); // For alignment
            ↪  testing - don't fill in triangles.
#else
            GLHelper.Draw(fieldIndices, PrimitiveType.Triangles);
#endif

            // Drawing contour lines over the top
            if (parameterHolder.DrawContours.Value)
            {
                contourIndices = GLHelper.FindContourIndices(streamFunction,
                ↪  parameterHolder.ContourTolerance.Value,
                ↪  parameterHolder.NumContours.Value, primitiveRestartIndex,
                ↪  dataWidth, dataHeight);
                contourShaderManager.Use();

                GL.BindVertexArray(hContourVAO);
                GLHelper.UpdateEBO(contourIndices, BufferUsageHint.DynamicDraw);

                GLHelper.Draw(contourIndices, PrimitiveType.LineStrip);
            }

            frameTime = (float)delta.TotalSeconds;
            PropertyChanged?.Invoke(this, new
             ↪  PropertyChangedEventArgs(nameof(FrameTime)));

            ErrorCode errorCode = GL.GetError();
            if (errorCode != ErrorCode.NoError)
            {
                Trace.WriteLine("\x1B[31m" + errorCode.ToString() + "\033[0m");
            }
        }
    }
}
```

contourShader.frag

```
#version 330 core

out vec4 FragColour;

void main() {
        FragColour = vec4(0.0f, 0.0f, 0.0f, 1.0f);
}
```

contourShader.vert

```
#version 330 core

layout (location = 0) in vec2 position;

void main()
{
        gl_Position = vec4(position, 0.0f, 1.0f);
}
```

fieldShader.frag

```
#version 330 core

in float relativeStrength;

uniform float subtrahend; // All strength values must have this subtracted from them
 ↪  for the range [0, max] ...
uniform float scalar; // ... and then must be multiplied by this to be in the range
 ↪  [0, 1] for processing

out vec4 FragColour;

void BluePurpleScale(out vec4 colourVector, in float strength) {
    colourVector = vec4(strength, 0.15625, 0.96875, 1.0);
}

void GreenRedScale(out vec4 colourVector, in float strength)
{
    // Red: 0 for strength < 0.5 then linear increase on [0.5, 0.75] then 1 for
    ↪  strength > 0.75
    // Green: 0 for strength < 0.25, then linear increase on [0.25, 0.5] to 1 then
    ↪  linear decrease on [0.5, 0.75], then 0 for strength > 0.75
    // Blue: 1 for strength < 0.25 then linear decrease on [0.25, 0.5] then 0 for
    ↪  strength > 0.5 (opposite of red)
    if (strength < 0.25)
    {
        colourVector = vec4(0.0, 0.0, 1.0, 1.0);
    }
    else if (strength < 0.5) // Interval [0.25, 0.5]
    {
        colourVector = vec4(0.0, (strength - 0.25) * 4.0, 1.0 - (strength - 0.25) *
        ↪  4, 1.0);
    }
    else if (strength < 0.75) // Interval [0.5, 0.75]
    {
        colourVector = vec4((strength - 0.5) * 4.0, 1.0 - (strength - 0.5) * 4, 0.0,
        ↪  1.0);
    }
    else
    {
        colourVector = vec4(1.0, 0.0, 0.0, 1.0);
    }
}


void main()
{
    float normalisedStrength = (relativeStrength - subtrahend) * scalar;
    BluePurpleScale(FragColour, normalisedStrength);
}
```

fieldShader.vert

```
#version 330 core

layout (location = 0) in vec2 position;
layout (location = 1) in float strength;

out float relativeStrength;

void main()
{
        relativeStrength = strength;
        gl_Position = vec4(position, 0.0f, 1.0f);
}
```

GLHelper.cs

```
using OpenTK.Graphics.OpenGL4;
using System.Diagnostics;

namespace Visualisation
{
    public static class GLHelper
    {
        /// <summary>
        /// Creates an array of vertices, with values for each coordinate in the
        ↪ field.
        /// </summary>
        /// <param name="fieldValues">A flattened array of values for the
        ↪ field.</param>
        /// <param name="width">The number of vertices in the x direction.</param>
        /// <param name="height">The number of vertices in the y direction</param>
        /// <returns>An array of floats to be passed to the vertex
        ↪ shader.s</returns>
        public static float[] FillVertices(int width, int height)
        {
            float[] vertices = new float[2 * width * height];

            float horizontalStep = 2f / (width - 1);
            float verticalStep = 2f / (height - 1);

            for (int i = 0; i < width; i++)
            {
                for (int j = 0; j < height; j++)
                {
                    // Need to start at bottom left (-1, -1) and go vertically then
                    ↪ horizontally to top right (1, 1)
                    vertices[i * height * 2 + j * 2 + 0] = i * horizontalStep - 1;
                    ↪ // Starting at -1, increase x coordinate each iteration of
                    ↪ outer loop
                    vertices[i * height * 2 + j * 2 + 1] = j * verticalStep - 1;  //
                    ↪ Starting at -1, increase y coordinate after each iteration
                    ↪ of inner loop
                }
            }
            return vertices;
        }

        /// <summary>
        /// Creates an index array for triangles to be drawn into a grid
        /// </summary>
        /// <param name="width">The width of the simulation space</param>
        /// <param name="height">The height of the simulation space</param>
        /// <returns>An array of unsigned integers representing the indices of each
        ↪ triangle, flattened</returns>
        public static uint[] FillIndices(int width, int height)
        {
```

```
        // Note that the given data has first data point bottom left, then
        ↪ moving upwards (in the positive y direction) then moving left
        ↪ (positive x direction)
        uint[] indices = new uint[(height - 1) * (width - 1) * 6];
        // For each 2x2 square of vertices, we need 2 triangles with the
        ↪ hypotenuses on the leading diagonal.
        for (int i = 0; i < width - 1; i++)
        {
            for (int j = 0; j < height - 1; j++)
            {
                indices[i * (height - 1) * 6 + j * 6 + 0] = (uint)(i * height +
                ↪ j);           // Top left
                indices[i * (height - 1) * 6 + j * 6 + 1] = (uint)(i * height +
                ↪ j + 1);       // Top right
                indices[i * (height - 1) * 6 + j * 6 + 2] = (uint)((i + 1) *
                ↪ height + j + 1); // Bottom right
                indices[i * (height - 1) * 6 + j * 6 + 3] = (uint)(i * height +
                ↪ j);           // Top left
                indices[i * (height - 1) * 6 + j * 6 + 4] = (uint)((i + 1) *
                ↪ height + j + 1); // Bottom right
                indices[i * (height - 1) * 6 + j * 6 + 5] = (uint)((i + 1) *
                ↪ height + j);     // Bottom left
            }
        }
        return indices;
    }

    /// <summary>
    /// Creates an array of <c>uint</c>s, representing the indices of where
    ↪ contour vertices should be, with each level set separated by <paramref
    ↪ name="primitiveRestartSentinel"/>.
    /// </summary>
    /// <param name="streamFunction">The values of the stream function for the
    ↪ simulation domain.</param>
    /// <param name="contourTolerance">The tolerance for accepting a vertex into
    ↪ the level set.</param>
    /// <param name="spacingMultiplier">A multiplier, such that vertices that
    ↪ have a stream function value that is an integer multiple of this
    ↪ multiplier will be included into the level set</param>
    /// <param name="primitiveRestartSentinel">The sentinel value, such as
    ↪ <c>uint.MaxValue</c></param>
    /// <param name="width">The width of the simulation space</param>
    /// <param name="height">The height of the simulation space</param>
    /// <returns>An array of <c>uint</c>s, to be passed to the EBO</returns>
    public static uint[] FindContourIndices(float[] streamFunction, float
    ↪ contourTolerance, float numContours, uint primitiveRestartSentinel, int
    ↪ width, int height)
    {
        float spacingMultiplier = streamFunction[height - 1] / numContours;
        List<List<uint>> levelSets = new();
        for (int j = 0; j < height; j++) // Find level sets
        {
            float streamFunctionValue = streamFunction[j];
            if (streamFunctionValue == 0)
            {
                continue;
            }
            float distanceFromMultiple = streamFunctionValue %
            ↪  spacingMultiplier;
            int levelSet;
            if (distanceFromMultiple < contourTolerance || spacingMultiplier -
            ↪  distanceFromMultiple < contourTolerance)
            {
                levelSet = (int)Math.Round(streamFunctionValue /
                ↪  spacingMultiplier); // Round the value to get the correct
                ↪  level set
            }
            else
            {
                continue;
            }

            while (levelSet >= levelSets.Count) // Add level set lists until
            ↪  there is one for the current level set
            {
                levelSets.Add(new List<uint>());
            }

            levelSets[levelSet].Add((uint)j); // Add the current index
        }

        List<uint> indices = new();

        for (int levelSetNum = 1; levelSetNum < levelSets.Count; levelSetNum++)
        ↪  // Go through each level set, finding coordinates that belong to the
        ↪  level set. Start at 1 because the 0 level set is not drawn.
        {
            if (levelSets[levelSetNum].Count == 0) continue; // The level set
            ↪  does not exist
            int currentHeight = (int)levelSets[levelSetNum][0]; // Get the
            ↪  starting height of the level set
            float targetValue = levelSetNum * spacingMultiplier;
            for (int i = 1; i < width-1; i++)
            {
                if (!(streamFunction[i * height + currentHeight] - targetValue >
                ↪  contourTolerance) && !(targetValue - streamFunction[i *
                ↪  height + currentHeight] > contourTolerance)) // Add in
                ↪  another condition to avoid floating point error (which
                ↪  should be always less than contour tolerance)
                {
                    levelSets[levelSetNum].Add((uint)(i * height +
                    ↪  currentHeight));
                    continue;
                }
            if (streamFunction[i * height + currentHeight] > targetValue) //
             ↪  Possibilities: current value is too big, need to move down;
             ↪  or current value is too small, need to move up. For both
             ↪  cases, either there exists a member of the level set or
             ↪  there does not.
            { // Stream function greater than target, need to move
             ↪  downwards
                while (currentHeight > 0 && streamFunction[i * height +
                 ↪  currentHeight] - targetValue > contourTolerance) //
                 ↪  While we are still too big, decrease height until 0
                {
                    currentHeight--;
                }
                // Now, current height is either larger than target but
                 ↪  within tolerance, below target but within tolerance, or
                 ↪  neither
                if (streamFunction[i * height + currentHeight] > targetValue
                 ↪  || targetValue - streamFunction[i * height +
                 ↪  currentHeight] < contourTolerance) // Within tolerance
                 ↪  either side of target
                {
                    levelSets[levelSetNum].Add((uint)(i * height +
                     ↪  currentHeight));
                }
                // If it is not within the tolerance, there does not exist a
                 ↪  stream function value at this x coordinate in the level
                 ↪  set.
            }
            else // Current height's contour value is too small
            {
                while (currentHeight < height - 1 && streamFunction[i *
                 ↪  height + currentHeight] < targetValue) // While we are
                 ↪  still too small, increase height until limit
                {
                    currentHeight++;
                }
                // Now, current height is either smaller than target but
                 ↪  within tolerance, above target but within tolerance, or
                 ↪  neither
                if (targetValue > streamFunction[i * height + currentHeight]
                 ↪  || streamFunction[i * height + currentHeight] -
                 ↪  targetValue < contourTolerance)
                {
                    levelSets[levelSetNum].Add((uint)(i * height +
                     ↪  currentHeight));
                }
            }
        }
        indices.AddRange(levelSets[levelSetNum]);
        indices.Add(primitiveRestartSentinel);
    }
    return indices.ToArray();
}
```

```
/// <summary>
/// Creates an element buffer object, and buffers the indices array.
/// </summary>
/// <param name="indices">An array representing the indices of the
↪  primitives that are to be drawn.</param>
/// <returns>A handle to the created EBO.</returns>
public static int CreateEBO(uint[] indices)
{
    int EBOHandle = GL.GenBuffer();
    GL.BindBuffer(BufferTarget.ElementArrayBuffer, EBOHandle);
    GL.BufferData(BufferTarget.ElementArrayBuffer, indices.Length *
    ↪  sizeof(uint), indices, BufferUsageHint.StaticDraw);
    return EBOHandle;
}

/// <summary>
/// Creates an element buffer object, and buffers the indices array.
/// </summary>
/// <param name="indices">An array representing the indices of the
↪  primitives that are to be drawn.</param>
/// <param name="bufferUsageHint">The enum value to tell the GPU which type
↪  of memory it should use.</param>
/// <returns>A handle to the created EBO.</returns>
public static int CreateEBO(uint[] indices, BufferUsageHint bufferUsageHint)
{
    int EBOHandle = GL.GenBuffer();
    GL.BindBuffer(BufferTarget.ElementArrayBuffer, EBOHandle);
    GL.BufferData(BufferTarget.ElementArrayBuffer, indices.Length *
    ↪  sizeof(uint), indices, bufferUsageHint);
    return EBOHandle;
}

/// <summary>
/// Buffers new data into the currently bound EBO.
/// </summary>
/// <param name="indices">An array representing the indices of the
↪  primitives that are to be drawn.</param>
/// <param name="bufferUsageHint">The enum value to tell the GPU which type
↪  of memory it should use.</param>
public static void UpdateEBO(uint[] indices, BufferUsageHint
↪  bufferUsageHint)
{
    GL.BufferData(BufferTarget.ElementArrayBuffer, indices.Length *
    ↪  sizeof(uint), indices, bufferUsageHint);
}

/// <summary>
/// Creates a vertex array object, which will hold the data to be passed to
↪  the vertex shader.
/// </summary>
/// <returns>A handle to the created VAO</returns>
public static int CreateVAO()
```

```
{
    int VAOHandle = GL.GenVertexArray();
    GL.BindVertexArray(VAOHandle);
    return VAOHandle;
}

/// <summary>
/// Creates an attribute pointer, providing metadata to OpenGL when passing
↪  data to the vertex shader.
/// </summary>
/// <param name="pointerNumber">The number of this pointer - this is the
↪  number passed to layout in the vertex shader.</param>
/// <param name="length">The dimension of the resulting vector</param>
/// <param name="stride">The width (in number of floats) of the subsections
↪  of the vertex array</param>
/// <param name="offset">The position (in number of floats) of the first
↪  element to include in the resulting vector</param>
public static void CreateAttribPointer(int pointerNumber, int length, int
↪  stride, int offset)
{
    GL.VertexAttribPointer(pointerNumber, length,
    ↪  VertexAttribPointerType.Float, false, stride * sizeof(float), offset
    ↪  * sizeof(float));
    GL.EnableVertexAttribArray(pointerNumber);
}

/// <summary>
/// Creates a buffer and binds it.
/// </summary>
/// <returns>A handle to the created VBO.</returns>
public static int CreateVBO()
{
    int VBOHandle = GL.GenBuffer();
    GL.BindBuffer(BufferTarget.ArrayBuffer, VBOHandle);
    return VBOHandle;
}

/// <summary>
/// Creates a buffer and binds it, filling it with blank data to ensure it
↪  is the correct size.
/// </summary>
/// <param name="size">The length, in number of floats, of the desired
↪  buffer.</param>
/// <returns>A handle to the created VBO.</returns>
public static int CreateVBO(int size)
{
    int VBOHandle = GL.GenBuffer();
    GL.BindBuffer(BufferTarget.ArrayBuffer, VBOHandle);
    GL.BufferData(BufferTarget.ArrayBuffer, size * sizeof(float), new
    ↪  float[size], BufferUsageHint.StreamDraw);
    return VBOHandle;
}
```

```
        /// <summary>
        /// Copies a <c>float[]</c> into part of a buffer, starting at <paramref
        ↪   name="offset"/>
        /// </summary>
        /// <param name="data">The <c>float[]</c> to be copied into the
        ↪   buffer</param>
        /// <param name="offset">The desired index of the first float to be
        ↪   copied</param>
        public static void BufferSubData(float[] data, int offset)
        {
            GL.BufferSubData(BufferTarget.ArrayBuffer, offset * sizeof(float),
            ↪   data.Length * sizeof(float), data);
        }

        /// <summary>
        /// Draws the grid, using triangles with indices specified in <paramref
        ↪   name="indices"/>.
        /// </summary>
        /// <param name="indices">An array of unsigned integers specifying the order
        ↪   in which to link vertices together.</param>
        /// <param name="primitiveType">Which type of primitive type to use for
        ↪   drawing.</param>
        public static void Draw(uint[] indices, PrimitiveType primitiveType)
        {
            GL.DrawElements(primitiveType, indices.Length,
            ↪   DrawElementsType.UnsignedInt, 0);
        }
    }
}
```

ShaderManager.cs

```
using OpenTK.Graphics.OpenGL4;

namespace Visualisation
{
    public class ShaderManager : IDisposable
    {
        private int programHandle;
        private bool isDisposed = false;

        public int Handle { get => programHandle; set => programHandle = value; }

        private static void ExtractShaderSource(string path, ShaderType type, out
        ↪   int shaderHandle)
        {
            string shaderSource = File.ReadAllText(path);

            shaderHandle = GL.CreateShader(type);
            GL.ShaderSource(shaderHandle, shaderSource);
        }

        private static void CompileShader(int shaderHandle)
        {
```

```
        GL.CompileShader(shaderHandle);
        GL.GetShader(shaderHandle, ShaderParameter.CompileStatus, out int
        ↪  success);
        if (success == 0) // Error in compilation
        {
            Console.WriteLine(GL.GetShaderInfoLog(shaderHandle));
        }
    }

    private void LinkShaders(int[] shaderHandles)
    {
        programHandle = GL.CreateProgram();

        foreach (int shaderHandle in shaderHandles)
        {
            GL.AttachShader(programHandle, shaderHandle);
        }

        GL.LinkProgram(programHandle);

        GL.GetProgram(programHandle, GetProgramParameterName.LinkStatus, out int
        ↪  success);
        if (success == 0) // Error case
        {
            Console.WriteLine(GL.GetProgramInfoLog(programHandle));
        }
    }


    private void BuildProgram((string, ShaderType)[] shadersWithPaths)
    {
        int[] handles = new int[shadersWithPaths.Length];
        for (int i = 0; i < shadersWithPaths.Length; i++)
        {
            ExtractShaderSource(shadersWithPaths[i].Item1,
            ↪  shadersWithPaths[i].Item2, out handles[i]);
            CompileShader(handles[i]);
        }

        LinkShaders(handles);

        foreach(int shaderHandle in handles)
        {
            GL.DetachShader(programHandle, shaderHandle);
            GL.DeleteShader(shaderHandle);
        }
    }

    public ShaderManager((string, ShaderType)[] shadersWithPaths)
    {
        BuildProgram(shadersWithPaths);
    }
```

```
        ~ShaderManager()
        {
            if (!isDisposed)
            {
                Console.WriteLine("Object not disposed of correctly");
                throw new InvalidOperationException("Object was not disposed of
                ↪  correctly.");
            }
        }

        public void Use()
        {
            GL.UseProgram(programHandle);
        }

        public int GetUniformLocation(string uniformName)
        {
            return GL.GetUniformLocation(programHandle, uniformName);
        }

        public void SetUniform(int uniformLocation, float value)
        {
            GL.Uniform1(uniformLocation, value);
        }

        public void Dispose()
        {
            if (!isDisposed)
            {
                GL.DeleteProgram(programHandle);
                isDisposed = true;
            }
            GC.SuppressFinalize(this);
        }
    }
}
```