NEAFluidDynamics

- BackendLib
  - BackendCoordinator.cpp
  - BackendCoordinator.h
  - BackendLib.cpp
  - Definitions.h
  - Flags.cpp
  - Flags.h
  - Init.cpp
  - Init.h
  - pch.cpp
  - pch.h
  - PipeConstants.h
  - PipeManager.cpp
  - PipeManager.h
  - Solver.cpp
  - Solver.h
- CPUBackend
  - Boundary.cpp
  - Boundary.h
  - Computation.cpp
  - Computation.h
  - CPUBackend.cpp
  - CPUSolver.cpp
  - CPUSolver.h
  - DiscreteDerivatives.cpp
  - DiscreteDerivatives.h
- GPUBackend
  - Boundary.cu
  - Boundary.cuh
  - Computation.cu
  - Computation.cuh
  - Definitions.cuh
  - DiscreteDerivatives.cu
  - DiscreteDerivatives.cuh
  - GPUSolver.cu
  - GPUSolver.cuh
  - kernel.cu
  - PressureComputation.cu
  - PressureComputation.cuh
  - ReductionKernels.cu
  - ReductionKernels.cuh
- UserInterface
  - App.xaml
  - App.xaml.cs
  - AssemblyInfo.cs
  - Converters
    - AbsoluteRectToRelativePol.cs
    - AbsoluteToRelativeRect.cs
    - BoolToTickPlacement.cs
    - CoordinateDifference.cs
    - PolarListToRectList.cs
    - RectangularToPolar.cs
    - RelativeDimension.cs
    - RelativePolToAbsoluteRect.cs
    - SignificantFigures.cs
    - VisualisationCoordinate.cs
    - VisualisationXCoordinate.cs
    - VisualisationYCoordinate.cs
    - VisualisationYCoordinateInverted.cs
    - XCoordinateDifference.cs
    - YCoordinateDifference.cs
  - HelperClasses
    - BackendManager.cs
    - CircularQueue.cs
    - Commands.cs
    - DefaultParameters.cs
    - MovingAverage.cs
    - ParameterChangedEventArgs.cs
    - ParameterHolder.cs
    - PipeConstants.cs
    - PipeManager.cs
    - PolarPoint.cs
    - PolarSplineCalculator.cs
    - Queue.cs
    - ResizableLinearQueue.cs
  - HelperControls
    - ResizableCentredTextBox.xaml
    - ResizableCentredTextBox.xaml.cs
    - SliderWithValue.xaml
    - SliderWithValue.xaml.cs
- V

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::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;
        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 = 2000;
    parameters.dynamicViscosity = (REAL)0.00001983;
    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.


struct DoubleField
{
        REAL** x;
        REAL** y;
};

struct DoubleReal
{
        REAL x;
        REAL y;
};

struct SimulationParameters
{
        REAL width;
        REAL height;
        REAL timeStepSafetyFactor;
        REAL relaxationParameter;
```

```
        REAL pressureResidualTolerance;
        int pressureMinIterations;
        int pressureMaxIterations;
        REAL reynoldsNo;
        REAL dynamicViscosity;
        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;
```

```cpp
    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

```cpp
#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

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

//bool** ReadObstaclesFromFile(std::string filename) {
//        std::ifstream binFile(filename);
//        std::vector<BYTE>
↪  buffer(std::istreambuf_iterator<char>(binFile), {}); // Copy
↪  the contents of the file to a buffer
//        int xLength = *reinterpret_cast<int*>(&buffer[0]); //
↪  Use the fact that the bits are stored concurrently in memory
↪  to get the xLength and yLength from the first and second 4
↪  bits.
//        int yLength = *reinterpret_cast<int*>(&buffer[4]);
//        bool** obstacles = new bool* [xLength];
//        for (int i = 0; i < xLength; i++) {
//                obstacles[i] = new bool[yLength]();
```

```
//                  for (int j = 0; j < yLength; j++) {
//                          obstacles[i][j] =
↪   *reinterpret_cast<bool*>(&buffer[yLength * i + j + 8]);
//                  }
//          }
//          return obstacles;
//}

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) {
```

```cpp
	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

```cpp
#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

```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

// add headers that you want to pre-compile here
#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;
}
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

        BYTE buffer[13];
        Read(buffer, 12);

        if (buffer[1] != (PipeConstants::Marker::PRMSTART |
        ↪  PipeConstants::Marker::IMAX)) { return std::pair<int,
        ↪  int>(0, 0); } // Should start with PRMSTART
        int iMax = *reinterpret_cast<int*>(buffer + 2);
```

```
        if (buffer[6] != (PipeConstants::Marker::PRMEND |
        ↪  PipeConstants::Marker::IMAX)) { return std::pair<int,
        ↪  int>(0, 0); } // Should end with PRMEND

        if (buffer[7] != (PipeConstants::Marker::PRMSTART |
        ↪  PipeConstants::Marker::JMAX)) { return std::pair<int,
        ↪  int>(0, 0); }
        int jMax = *reinterpret_cast<int*>(buffer + 8);
        if (buffer[12] != (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;
}

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>
/// 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;

    /// <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>
```

```cpp
#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];
```

```
    //velocities.x[iMax][j] = velocities.x[iMax - 1][j] * 0.5
    ↪  + 0.5 * inflowVelocity; // Get some of the velocity
    ↪  value from the previous cell, some from inflow
    ↪  velocity (avoids pushback of fluid at the right
    ↪  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.
        }
    }
}
```

```cpp
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

```cpp
#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

```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>
```

```cpp
//#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

    int iMax = 100;
    int jMax = 100;
    SimulationParameters parameters = SimulationParameters();
    if (argc == 1) { // Not linked to a frontend.
        parameters.width = 1;
        parameters.height = 1;
        parameters.timeStepSafetyFactor = (REAL)0.5;
        parameters.relaxationParameter = (REAL)1.7;
        parameters.pressureResidualTolerance = 1;
        parameters.pressureMinIterations = 10;
        parameters.pressureMaxIterations = 1000;
        parameters.reynoldsNo = 1000;
        parameters.inflowVelocity = 1;
        parameters.surfaceFrictionalPermissibility = 0;
        DoubleReal bodyForces = DoubleReal();
        bodyForces.x = 0;
        bodyForces.y = 0;
        parameters.bodyForces = bodyForces;

        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)(0.55 * 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;
            }
```

```
        }

        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);
            std::cout << "Iteration " << i << ", time taken: " <<
             ↪  cumulativeTimestep << ".\n";
        }
        return 0;
    }
    else if (argc == 2) { // Linked to a frontend.
        char* pipeName = argv[1];
        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 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"

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();
}

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;
}

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;

    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
```

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. This could probably be optimised.
                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))
```

```
template <typename T>
struct PointerWithPitch
{
    T* ptr;
    size_t pitch;
};

#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
```

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);

    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)ceilf(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;
}

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);

    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.

printf("Number of iterations: %i, residual norm: %f.\n",
↪  pressureIterations, pressureResidualNorm);

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"

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.

    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;

    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) {
    int iMax = 100;
    int jMax = 100;
    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";
        parameters.width = 1;
        parameters.height = 1;
        parameters.timeStepSafetyFactor = (REAL)0.5;
        parameters.relaxationParameter = (REAL)1.7;
        parameters.pressureResidualTolerance = 35;
        parameters.pressureMinIterations = 10;
        parameters.pressureMaxIterations = 1000;
        parameters.reynoldsNo = 1000;
        parameters.inflowVelocity = 1;
        parameters.surfaceFrictionalPermissibility = 0;
        DoubleReal bodyForces = DoubleReal();
        bodyForces.x = 0;
        bodyForces.y = 0;
        parameters.bodyForces = bodyForces;

        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)(0.25 * iMax);
int boundaryRight = (int)(0.35 * 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;
    }
}

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

REAL cumulativeTimestep = 0;

int numIterations = 10;
std::cerr << "2 seconds to attach profiler / debugger\n";
Sleep(2000);
/*std::cout << "Enter number of iterations: ";
std::cin >> numIterations;*/

float timeTakenSum = 0;

for (int i = 0; i < numIterations; i++) {
    auto startTime =
    ↪  std::chrono::high_resolution_clock::now();
    solver.Timestep(cumulativeTimestep);
    auto endTime =
    ↪  std::chrono::high_resolution_clock::now();
    float millisecondsDuration = (endTime -
    ↪  startTime).count() / 1000000.0f;
    timeTakenSum += millisecondsDuration;
    std::cout << "Iteration " << i << ", cumulative
    ↪  timestep: " << cumulativeTimestep << ", time to
    ↪  execute:" << millisecondsDuration << " ms.\n";
}

std::cout << std::endl << "Average over " <<
↪  numIterations << " iterations: " << timeTakenSum /
↪  numIterations << " ms.\n";
```

```
        return 0;
    }
    else if (argc == 2) { // Linked to a frontend.
        char* pipeName = argv[1];
        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 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];
    }
}

/// <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)
{
    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 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 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 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

```xml
<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>
    </Application.Resources>
</Application>
```

App.xaml.cs

```csharp
using System;
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 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;
```

```
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]); // 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]);
    popupWindowContainer.Content = currentPopup;

    popupWindowContainer.Show();
}

private void DeletePopup(object? sender, EventArgs e)
{
    currentPopup = null;
    popupWindowContainer.Content = currentPopup;

    popupWindowContainer.Hide();
}

// Static method 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 = 400,
        Width = 700
    };

    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.SURFACE_FRICTION, new
    ↪  FieldParameters(),
    ↪  DefaultParameters.DRAW_CONTOURS,
    ↪  DefaultParameters.CONTOUR_TOLERANCE,
    ↪  DefaultParameters.CONTOUR_SPACING); // Use the
    ↪  defaults from DefaultParameters constant holder

    currentUserControl = new
    ↪  ConfigScreen(parameterHolder);
    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).
            {
                simulationScreen.ViewModel.CloseBackend();
            }
        }

        //public void StartVisualisationDebugging(object sender,
        ↪  StartupEventArgs e)
        //{
        //    fullScreenWindowContainer = new MainWindow();
        //    fullScreenWindowContainer.Content = new
        ↪  VisualisationControl();
        //    fullScreenWindowContainer.Show();
        //}
    }

    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)
)]
```

AbsoluteRectToRelativePol.cs

```
using System;
using System.Globalization;
using System.Linq;
using System.Windows;
using System.Windows.Data;

namespace UserInterface.Converters
{
    public class AbsoluteRectToRelativePol : IMultiValueConverter
    {
        /// <summary>
        /// Converts an absolute rectangular coordinate to a
        ↪  relative polar one, relative to a specified canvas
        ↪  dimensions and optionally an origin.
        /// </summary>
        /// <param name="values">3 values: the <see
        ↪  cref="Point"/> to convert, the canvas width and the
        ↪  canvas height.</param>
        /// <param name="parameter">An optional origin, specified
        ↪  as a relative <see cref="Point"/> or <see
        ↪  cref="string"/>.</param>
        /// <returns>A relative polar coordinate.</returns>
        public object Convert(object[] values, Type targetType,
        ↪  object parameter, CultureInfo culture)
        {
```

```
RelativeDimension CoordinateConverter = new
↪  RelativeDimension();
RectangularToPolar RecToPolConverter = new
↪  RectangularToPolar();

if (values[0] is not Point || values[1] is not double
↪  || values[2] is not double)
{
    return DependencyProperty.UnsetValue;
}
Point point = (Point)values[0];
double canvasWidth = (double)values[1];
double canvasHeight = (double)values[2];

Point origin;
if (parameter is null)
{
    origin = new Point(0, 0);
}
else 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);
}

double xCoordinate = point.X * 2;
double yCoordinate = point.Y * 2;


Point relativePoint = new Point(
```

```
                ↪  (double)CoordinateConverter.ConvertBack(canvasWidth,
                ↪  targetType, xCoordinate.ToString(), culture),
                1 -
                ↪  (double)CoordinateConverter.ConvertBack(canvasHeight,
                ↪  targetType, yCoordinate.ToString(),
                ↪  culture)); // Flip the y coordinate about the
                ↪  centre of the canvas to make it 0 at the
                ↪  bottom rather than at the top.
            return RecToPolConverter.Convert(relativePoint,
            ↪  targetType, origin, culture);
        }

        public object[] ConvertBack(object value, Type[]
        ↪  targetTypes, object parameter, CultureInfo culture)
        {
            throw new InvalidOperationException("Use
            ↪  RelativePolToAbsoluteRect instead.");
        }
    }
}
```

AbsoluteToRelativeRect.cs

```
using System;
using System.Globalization;
using System.Windows;
using System.Windows.Data;

namespace UserInterface.Converters
{
    public class AbsoluteToRelativeRect : IMultiValueConverter
    {
        /// <summary>
        /// Converts an absolute rectangular coordinate to a
        ↪  relative rectangular coordinate, optionally flipping
        ↪  the y coordinate to make the origin bottom-left
        ↪  rather than bottom-right.
        /// </summary>
        /// <param name="values">An array of objects in the form:
        ↪  point, parent width, parent height.</param>
        /// <param name="parameter">A boolean specifying whether
        ↪  to flip the y coordinate.</param>
        /// <returns>A relative rectangular coordinate.</returns>
        public object Convert(object[] values, Type targetType,
        ↪  object parameter, CultureInfo culture)
```

```
{
    RelativeDimension CoordinateConverter = new
    ↪  RelativeDimension();

    if (values[0] is not Point || values[1] is not double
    ↪  || values[2] is not double)
    {
        return DependencyProperty.UnsetValue;
    }
    Point point = (Point)values[0];
    double canvasWidth = (double)values[1];
    double canvasHeight = (double)values[2];

    bool flipY;
    if (parameter is null)
    {
        flipY = false;
    }
    else if (parameter is bool)
    {
        flipY = (bool)parameter;
    }
    else if (parameter is string)
    {
        if (!bool.TryParse(((string)parameter).ToLower(),
        ↪  out flipY))
        {
            return DependencyProperty.UnsetValue;
        }
    }
    else
    {
        return DependencyProperty.UnsetValue;
    }

    double xCoordinate = point.X * 2;
    double yCoordinate = point.Y * 2;

    double relativeX =
    ↪  (double)CoordinateConverter.ConvertBack(canvasWidth,
    ↪  targetType, xCoordinate.ToString(), culture);
    double relativeY =
    ↪  (double)CoordinateConverter.ConvertBack(canvasHeight,
    ↪  targetType, yCoordinate.ToString(), culture);
    if (flipY)
    {
```

```
                relativeY = 1 - relativeY;
            }

            return new Point(relativeX, relativeY);
        }

        public object[] ConvertBack(object value, Type[]
         ↪  targetTypes, object parameter, CultureInfo culture)
        {
            throw new InvalidOperationException("Cannot convert
             ↪  back.");
        }
    }
}
```

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");
        }
    }
}
```

CoordinateDifference.cs

```
using System;
using System.Globalization;
using System.Linq;
using System.Windows;
using System.Windows.Data;

namespace UserInterface.Converters
{
    [ValueConversion(typeof(double), typeof(double))]
    public abstract class CoordinateDifference : IValueConverter
    {
        protected readonly VisualisationCoordinate
        ↪  VisualisationCoordinateConverter;

        protected abstract double FindLength(double start, double
        ↪  end);

        public object Convert(object value, Type targetType,
        ↪  object parameter, CultureInfo culture)
        {
            if (value is not double || parameter is not string ||
            ↪  !((string)parameter).Contains(','))
            {
                return DependencyProperty.UnsetValue;
            }

            string parameterNoSpaces =
            ↪  new(((string)parameter).ToCharArray().Where(c =>
            ↪  !char.IsWhiteSpace(c)).ToArray());
            string[] parameters = parameterNoSpaces.Split(',');

            if (!double.TryParse(parameters[0], out double
            ↪  elementStartCoord) ||
            ↪  !double.TryParse(parameters[1], out double
            ↪  elementEndCoord))
            {
                return DependencyProperty.UnsetValue;
            }
```

```
            double elementLength = FindLength(elementStartCoord,
            ↪  elementEndCoord); // Use formula to get relative
            ↪  length of element
            return
            ↪  VisualisationCoordinateConverter.Convert(value,
            ↪  targetType, elementLength.ToString(), culture);
            ↪  // Convert this to actual coordinates
        }

        public object ConvertBack(object value, Type targetType,
        ↪  object parameter, CultureInfo culture)
        {
            throw new InvalidOperationException();
        }

        public CoordinateDifference(VisualisationCoordinate
        ↪  VisualisationCoordinateConverter) // Derived classes
        ↪  will need to instantiate
        ↪  VisualisationCoordinateConverter in their
        ↪  constructors.
        {
            this.VisualisationCoordinateConverter =
            ↪  VisualisationCoordinateConverter;
        }
    }
}
```

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.
        }
    }
}
```

RelativeDimension.cs

```
using System;
using System.Globalization;
using System.Windows;
using System.Windows.Data;

namespace UserInterface.Converters
{
    /// <summary>
    /// Converts between relative (0-1) and absolute coordinates.
    /// </summary>
    public class RelativeDimension : IValueConverter
    {
        /// <summary>
        /// Converts a number between 0 and 1 into a fraction of
        ↪  the dimension of the parent.
        /// </summary>
        /// <param name="value">The dimension of the
        ↪  parent.</param>
        /// <param name="parameter">The relative
        ↪  coordinate.</param>
        /// <returns>The value that represents the fraction of
        ↪  the parent dimension.</returns>
        public object Convert(object value, Type targetType,
        ↪  object parameter, CultureInfo culture)
        {
            if (value is not double || parameter is not string ||
            ↪  !double.TryParse((string)parameter, out double
            ↪  fractionOfParent))
            {
                return DependencyProperty.UnsetValue;
            }
            double parentDimension = (double)value;
```

```
            return fractionOfParent * parentDimension;
        }

        /// <summary>
        /// Converts an absolute dimension to a number between 0
        ↪  and 1 relative to the dimension of the parent.
        /// </summary>
        /// <param name="value">The dimension of the
        ↪  parent.</param>
        /// <param name="parameter">The absolute
        ↪  coordinate.</param>
        /// <returns>A relative coordinate between 0 and
        ↪  1</returns>
        public object ConvertBack(object value, Type targetType,
        ↪  object parameter, CultureInfo culture)
        {
            if (value is not double || parameter is not string ||
            ↪  !double.TryParse((string)parameter, out double
            ↪  absoluteCoordinate))
            {
                return DependencyProperty.UnsetValue;
            }
            double parentDimension = (double)value;
            return absoluteCoordinate / parentDimension;
        }
    }
}
```

RelativePolToAbsoluteRect.cs

```
using UserInterface.HelperClasses;
using System;
using System.Globalization;
using System.Linq;
using System.Windows;
using System.Windows.Data;

namespace UserInterface.Converters
{
    internal class RelativePolToAbsoluteRect :
    ↪  IMultiValueConverter
    {
        /// <summary>
        /// Converts a relative polar coordinate to an asbolute
        ↪  rectangular one, relative to a specified canvas
        ↪  dimensions and optionally an origin.
```

```
/// </summary>
/// <param name="values">3 values: the <see
↪  cref="PolarPoint"/> to convert, the canvas width and
↪  the canvas height.</param>
/// <param name="parameter">An optional origin, specified
↪  as a relative <see cref="Point"/> or <see
↪  cref="string"/>.</param>
/// <returns>A relative polar coordinate.</returns>
public object Convert(object[] values, Type targetType,
↪  object parameter, CultureInfo culture)
{
    RelativeDimension CoordinateConverter = new
    ↪  RelativeDimension();
    RectangularToPolar RecToPolConverter = new
    ↪  RectangularToPolar();

    if (values[0] is not PolarPoint || values[1] is not
    ↪  double || values[2] is not double)
    {
        return DependencyProperty.UnsetValue;
    }
    PolarPoint point = (PolarPoint)values[0];
    double canvasWidth = (double)values[1];
    double canvasHeight = (double)values[2];

    Point origin;
    if (parameter is null)
    {
        origin = new Point(0, 0);
    }
    else 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);
        }

        Point relativePoint =
        ↪  (Point)RecToPolConverter.ConvertBack(point,
        ↪  targetType, origin, culture);
        Point absolutePoint =  new Point(
            (double)CoordinateConverter.Convert(canvasWidth,
            ↪  targetType, relativePoint.X.ToString(),
            ↪  culture),
            (double)CoordinateConverter.Convert(canvasHeight,
            ↪  targetType, (1 - relativePoint.Y).ToString(),
            ↪  culture));

        return new Point(absolutePoint.X / 2, absolutePoint.Y
        ↪  / 2);
    }

    public object[] ConvertBack(object value, Type[]
    ↪  targetTypes, object parameter, CultureInfo culture)
    {
        throw new InvalidOperationException("Use
        ↪  AbsoluteRectToRelativePol instead.");
    }
  }
}
```

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)
        {
            if (value is not float ||
            ↪  !int.TryParse((string)parameter, out int
            ↪  iParameter)) // Input validation
            {
                return "";
            }

            return ((float)value).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.");
        }
    }
}
```

VisualisationCoordinate.cs

```
using System;
using System.Globalization;
using System.Windows;

namespace UserInterface.Converters
{
    public abstract class VisualisationCoordinate
    {
        private readonly RelativeDimension
        ↪  RelativeDimensionConverter;

        public abstract double
        ↪  TranslateVisualisationCoordinate(double p);
```

```
public abstract double TranslateCanvasCoordinate(double
↪  p);

/// <summary>
/// Converts a canvas dimension and fraction to a
↪  coordinate as displayed by the visualisation.
/// </summary>
/// <param name="value">The canvas dimension.</param>
/// <param name="parameter">The fraction of the canvas to
↪  use.</param>
/// <returns>An absolute coordinate that aligns with
↪  coordinates of the visualisation.</returns>
public object Convert(object value, Type targetType,
↪  object parameter, CultureInfo culture)
{
    if (parameter is not string ||
    ↪  !double.TryParse((string)parameter, out double
    ↪  fractionOfCanvas))
    {
        return DependencyProperty.UnsetValue;
    }
    fractionOfCanvas =
    ↪  TranslateVisualisationCoordinate(fractionOfCanvas);

    return RelativeDimensionConverter.Convert(value,
    ↪  targetType, fractionOfCanvas.ToString(),
    ↪  culture);
}


/// <summary>
/// Converts an absolute coordinate as displayed by the
↪  visualisation to a fraction of the canvas.
/// </summary>
/// <param name="value">The canvas dimension.</param>
/// <param name="parameter">The absolute coordinate that
↪  aligns with the visualisation.</param>
/// <returns>A coordinate relative to the canvas [0,
↪  1].</returns>
public object ConvertBack(object value, Type targetType,
↪  object parameter, CultureInfo culture)
{
    // Input validation is done by
    ↪  RelativeDimensionConverter
```

```
            double relativeCoordinate =
            ↪  (double)RelativeDimensionConverter.ConvertBack(value,
            ↪  targetType, parameter, culture);

            return TranslateCanvasCoordinate(relativeCoordinate);
        }

        public VisualisationCoordinate()
        {
            RelativeDimensionConverter = new RelativeDimension();
        }
    }
}
```

VisualisationXCoordinate.cs

```
using System.Windows.Data;

namespace UserInterface.Converters
{
    public class VisualisationXCoordinate :
    ↪  VisualisationCoordinate, IValueConverter
    {
        /// <summary>
        /// Translates an x coordinate from [0, 1] to the precise
        ↪  point as rendered by <see cref="VisualisationControl"
        ↪  />.
        /// </summary>
        /// <param name="p">The x coordinate in [0, 1] to
        ↪  translate.</param>
        /// <returns>A point that maps to the <see
        ↪  cref="VisualisationControl"/> space.</returns>
        public override double
        ↪  TranslateVisualisationCoordinate(double p)
        {
            return -0.000451565 + 1.01036 * p;
        }

        /// <summary>
        /// Translates an x coordinate from the precise point as
        ↪  rendered by <see cref="VisualisationControl" /> to
        ↪  [0, 1].
        /// </summary>
        /// <param name="p">The point that maps to the <see
        ↪  cref="VisualisationControl"/> space to
        ↪  translate.</param>
```

```
        /// <returns>An x coordinate in [0, 1].</returns>
        public override double TranslateCanvasCoordinate(double
         ↪  p)
        {
            return (p + 0.000451565) / 1.01036;
        }
    }
}
```

VisualisationYCoordinate.cs

```
using System.Windows.Data;

namespace UserInterface.Converters
{
    public class VisualisationYCoordinate :
     ↪  VisualisationCoordinate, IValueConverter
    {
        /// <summary>
        /// Translates a y coordinate from [0, 1] to the precise
         ↪  point as rendered by <see cref="VisualisationControl"
         ↪  />.
        /// </summary>
        /// <param name="p">The y coordinate in [0, 1] to
         ↪  translate.</param>
        /// <returns>A point that maps to the <see
         ↪  cref="VisualisationControl"/> space.</returns>
        public override double
         ↪  TranslateVisualisationCoordinate(double p)
        {
            return 1.009 - 1.0099 * p;
        }

        /// <summary>
        /// Translates a y coordinate from the precise point as
         ↪  rendered by <see cref="VisualisationControl" /> to
         ↪  [0, 1].
        /// </summary>
        /// <param name="p">The point that maps to the <see
         ↪  cref="VisualisationControl"/> space to
         ↪  translate.</param>
        /// <returns>A y coordinate in [0, 1].</returns>
        public override double TranslateCanvasCoordinate(double
         ↪  p)
        {
            return (1.009 - p) / 1.0099;
```

```
        }
    }
}
```

VisualisationYCoordinateInverted.cs

```
using System.Windows.Data;

namespace UserInterface.Converters
{
    public class VisualisationYCoordinateInverted :
     ↪  VisualisationCoordinate, IValueConverter
    {
        /// <summary>
        /// Translates a y coordinate from [0, 1] to the precise
         ↪  point as rendered by <see cref="VisualisationControl"
         ↪  />.
        /// </summary>
        /// <param name="p">The y coordinate in [0, 1] to
         ↪  translate.</param>
        /// <returns>A point that maps to the <see
         ↪  cref="VisualisationControl"/> space.</returns>
        public override double
         ↪  TranslateVisualisationCoordinate(double p)
        {
            return -0.0009 + 1.0099 * p;
        }

        public override double TranslateCanvasCoordinate(double
         ↪  p)
        {
            return (p + 0.0009) / 1.0099;
        }
    }
}
```

XCoordinateDifference.cs

```
namespace UserInterface.Converters
{
    public class XCoordinateDifference : CoordinateDifference
    {
        /// <summary>
        /// Translates an x coordinate from [0, 1] to the precise
         ↪  point as rendered by <see cref="VisualisationControl"
         ↪  />.
```

```
        /// </summary>
        /// <param name="p">The x coordinate in [0, 1] to
        ↪  translate.</param>
        /// <returns>A point that maps to the <see
        ↪  cref="VisualisationControl"/> space.</returns>
        private double TranslateVisualisationCoordinate(double p)
        {
            return -0.000451565 + 1.01036 * p;
        }

        protected override double FindLength(double start, double
        ↪  end)
        {
            return TranslateVisualisationCoordinate(end) -
            ↪  TranslateVisualisationCoordinate(start) + 0.0006;
        }

        public XCoordinateDifference() : base(new
        ↪  VisualisationXCoordinate()) { }
    }
}
```

YCoordinateDifference.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace UserInterface.Converters
{
    public class YCoordinateDifference : CoordinateDifference
    {
        /// <summary>
        /// Translates a y coordinate from [0, 1] to the precise
        ↪  point as rendered by <see cref="VisualisationControl"
        ↪  />.
        /// </summary>
        /// <param name="p">The y coordinate in [0, 1] to
        ↪  translate.</param>
        /// <returns>A point that maps to the <see
        ↪  cref="VisualisationControl"/> space.</returns>
        public double TranslateVisualisationCoordinate(double p)
        {
            return 1.009 - 1.0099 * p;
```

```
        }

        protected override double FindLength(double start, double
        ↪  end)
        {
            return TranslateVisualisationCoordinate(end) -
            ↪  TranslateVisualisationCoordinate(start) + 0.0017;
        }

        public YCoordinateDifference() : base(new
        ↪  VisualisationYCoordinateInverted()) { }
    }
}
```

BackendManager.cs

```
//#define NO_GPU_BACKEND

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Diagnostics;
using System.IO;
using System.Threading;
using System.Threading.Tasks;
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 Stopwatch frameTimer;

    private ResizableLinearQueue<ParameterChangedEventArgs>
    ↪  parameterSendQueue;
    private ParameterHolder parameterHolder;

    private readonly string pipeName =
    ↪  "NEAFluidDynamicsPipe";

    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 BackendStatus BackendStatus
{
    get => backendStatus;
    private set
    {
        backendStatus = value;
        PropertyChanged?.Invoke(value, new
        ↪  PropertyChangedEventArgs(nameof(BackendStatus)));
    }
}

public event PropertyChangedEventHandler?
↪  PropertyChanged;

private bool CreateBackend()
{
    try
    {
        backendProcess = new Process();
        backendProcess.StartInfo.FileName = filePath;

        ↪  backendProcess.StartInfo.ArgumentList.Add(pipeName);
        //backendProcess.StartInfo.CreateNoWindow = true;
        backendProcess.Start();
        BackendStatus = BackendStatus.NotStarted;
        return true;
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
        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,
        _ => 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");
                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");
                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 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.FluidViscosity.Value,
    ↪ PipeConstants.Marker.MU);
    if (await pipeManager.ReadAsync() !=
    ↪ PipeConstants.Status.OK) throw new
    ↪ IOException("Backend did not read parameters
    ↪ correctly");
}

/// <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 fieldStart = await
        ↪  pipeManager.ReadAsync();
        if (fieldStart !=
        ↪  (PipeConstants.Marker.FLDSTART |
        ↪  fieldBits)) { throw new
        ↪  IOException($"Backend did not send data
        ↪  correctly. Bits were {fieldStart}"); } //
        ↪  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
    }

    if (await pipeManager.ReadAsync() !=
    ↪  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;
            }
            if (!backendProcess.HasExited)
            {
                return false;
            }
            backendProcess.Close();
            return true;
        }

        public void ForceCloseBackend()
        {
            backendProcess.Kill();
        }
    }
}
```

CircularQueue.cs

```
using System;

namespace UserInterface.HelperClasses
{
    public class CircularQueue<T> : Queue<T>
    {
        //protected bool isFull;
        //protected bool isEmpty;
        protected int count;

        public CircularQueue(int length) : base(length)
        {
            //isEmpty = true;
```

```
    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;
    }
    //isEmpty = false; //Set isEmpty to false, and
    ↪  IsEmpty to true if the front is equal to the back
    //isFull = (front == back);
    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;
    }
    //isFull = false; //Set isFull to false, and isEmpty
    ↪  to true if the front is equal to the back
    //isEmpty = (front == back);
    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 System;
using System.ComponentModel;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Input;
using UserInterface.ViewModels;

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;
    }

    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 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);
            }
        }
    }
}
```

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 = 1f;
        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 CONTOUR_SPACING = 0.05f;
        public static readonly int FPS_WINDOW_SIZE = 500;

        public static readonly float VELOCITY_MIN = 0f;
        public static readonly float VELOCITY_MAX = 5f;
        public static readonly float PRESSURE_MIN = 1000f;
        public static readonly float PRESSURE_MAX = 5000f;
    }
}
```

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
        }
    }
}
```

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> surfaceFriction;

// Visualisation parameters
private ParameterStruct<FieldParameters> fieldParameters;
private ParameterStruct<bool> drawContours;
private ParameterStruct<float> contourTolerance;
private ParameterStruct<float> contourSpacing;

#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> 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> ContourSpacing
{
    get => contourSpacing;

    set
    {
        contourSpacing = 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
↪  surfaceFriction, FieldParameters fieldParameters,
↪  bool drawContours, float contourTolerance, float
↪  contourSpacing)
{

    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.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.contourSpacing = new
            ↪  ParameterStruct<float>(contourSpacing,
            ↪  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;
        }

        /// <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 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;
    }

    pipeStream.WaitForPipeDrain();

    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.IMAX); // Send PRMEND

    pipeStream.Write(new ReadOnlySpan<byte>(buffer));

    pipeStream.WaitForPipeDrain();

    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();
}

/// <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[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>
/// 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;
using System.Collections.Generic;
using System.Windows.Navigation;

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;
    }
}
```

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:converters="clr-namespace:UserInterface.Converters"
             mc:Ignorable="d"
             d:DesignHeight="450" d:DesignWidth="800">
    <UserControl.Resources>
        <ResourceDictionary>
            <converters:BoolToTickPlacement
            ↪   x:Key="BoolToTickPlacementConverter" />
        </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
        ↪   Minimum}" />
        <Label Grid.Row="0" Grid.Column="2" Content="{Binding
        ↪   Maximum}" />
```

```
        <Slider Grid.Row="1" Grid.Column="0" Grid.ColumnSpan="3"
        ↪  Minimum="{Binding Minimum}" Maximum="{Binding
        ↪  Maximum}" Value="{Binding Value}"  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
        ↪  Value}" TextAlignment="Right" Width="40" />
    </Grid>
</UserControl>
```

SliderWithValue.xaml.cs

```
using System.Windows;
using System.Windows.Controls;
using System.Windows.Controls.Primitives;
using UserInterface.HelperClasses;

namespace UserInterface.HelperControls
{
    /// <summary>
    /// Interaction logic for SliderWithValue.xaml
    /// </summary>
    public partial class SliderWithValue : UserControl
    {

        // 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));
            ↪  // Value is two-way: changes to the slider must
            ↪  be passed to the source that uses this
            ↪  UserControl.

        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));

        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));

        public SliderWithValue()
        {
            InitializeComponent();
            LayoutRoot.DataContext = this;
        }

        public bool ForceIntegers { get; set; } = false;

        public TickPlacement TickPlacement { get; } =
         ↪  TickPlacement.None;
    }
}
```

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 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 Commands.AdvancedParametersReset ResetCommand {
↪  get; set; }

public Commands.SaveParameters SaveCommand { get; set; }
#endregion

public AdvancedParametersVM(ParameterHolder
↪  parameterHolder) : base(parameterHolder)
{
    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;

namespace UserInterface.ViewModels
{
    public class ConfigScreenVM : ViewModel
    {
        private float inVel;
        private float chi;
        private float width;
        private float height;
        private float reynoldsNo;
        private float viscosity;

        public float InVel
        {
            get => inVel;
            set { inVel = value; OnPropertyChanged(this,
            ↪  nameof(InVel)); parameterHolder.InflowVelocity =
            ↪  ModifyParameterValue(parameterHolder.InflowVelocity,
            ↪  inVel); }
        }

        public float Chi
        {
            get => chi;
            set { chi = value; OnPropertyChanged(this,
            ↪  nameof(Chi)); parameterHolder.SurfaceFriction =
            ↪  ModifyParameterValue(parameterHolder.SurfaceFriction,
            ↪  chi); }
        }

        public float Width
        {
            get => width;
            set { width = value; OnPropertyChanged(this,
            ↪  nameof(Width)); parameterHolder.Width =
            ↪  ModifyParameterValue(parameterHolder.Width,
            ↪  width); }
```

```
}

public float Height
{
    get => height;
    set { height = value; OnPropertyChanged(this,
    ↪  nameof(Height)); parameterHolder.Height =
    ↪  ModifyParameterValue(parameterHolder.Height,
    ↪  height); }
}

public float ReynoldsNo
{
    get => reynoldsNo;
    set { reynoldsNo = value; OnPropertyChanged(this,
    ↪  nameof(ReynoldsNo));
    ↪  parameterHolder.ReynoldsNumber =
    ↪  ModifyParameterValue(parameterHolder.ReynoldsNumber,
    ↪  reynoldsNo); }
}

public float Viscosity
{
    get => viscosity;
    set { viscosity = value; OnPropertyChanged(this,
    ↪  nameof(Viscosity));
    ↪  parameterHolder.FluidViscosity =
    ↪  ModifyParameterValue(parameterHolder.FluidViscosity,
    ↪  viscosity); }
}

public Commands.ConfigScreenReset ResetCommand { get;
↪  set; }
public Commands.SetAirParameters SetAirCommand { get;
↪  set; }
public Commands.ChangeWindow ChangeWindowCommand { get;
↪  set; }
public Commands.CreatePopup CreatePopupCommand { get;
↪  set; }

public ConfigScreenVM(ParameterHolder parameterHolder) :
↪  base(parameterHolder)
{
    InVel = parameterHolder.InflowVelocity.Value;
    Chi = parameterHolder.SurfaceFriction.Value;
    Width = parameterHolder.Width.Value;
```

```
            Height = parameterHolder.Height.Value;
            ResetCommand = new Commands.ConfigScreenReset(this,
             ↪  parameterHolder);
            SetAirCommand = new Commands.SetAirParameters(this);
            ChangeWindowCommand = new Commands.ChangeWindow();
            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;

namespace UserInterface.ViewModels
{
    public class SimulationScreenVM : ViewModel
    {
        #region Fields, Properties and Enums
        private SidePanelButton? currentButton;

        private float visLowerBound;
        private float visUpperBound;

        private float[] velocity;
        private float[] pressure;
        private float[] streamFunction;
        private FieldParameters pressureParameters;
        private FieldParameters velocityParameters;
        private SelectedField selectedField;

        private BackendManager backendManager;
        private CancellationTokenSource backendCTS;
        private VisualisationControl visualisationControl;
        private MovingAverage<float> visFrameTimeAverage;
```

```
private MovingAverage<float> backFrameTimeAverage;
private RectangularToPolar RecToPolConverter;

private int dataWidth;
private int dataHeight;

private ObservableCollection<PolarPoint> obstaclePoints;
private ObservableCollection<PolarPoint> controlPoints;
private PolarSplineCalculator obstaclePointCalculator;
private bool editingObstacles;
private Point obstacleCentre;

private const int numObstaclePoints = 80;
private const float boundaryTop = 0.55f;
private const float boundaryLeft = 0.15f;
private const float boundaryHeight = 0.1f;
private const float boundaryWidth = 0.1f;


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 ContourSpacing
{
    get => parameterHolder.ContourSpacing.Value;
    set
    {
        parameterHolder.ContourSpacing =
        ↪  ModifyParameterValue(parameterHolder.ContourSpacing,
        ↪  value);
        OnPropertyChanged(this, nameof(ContourSpacing));
    }
}
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 VisualisationControl VisualisationControl { get =>
↪  visualisationControl; }

public float VisFPS { get => 1 /
↪  visFrameTimeAverage.Average; }
public float BackFPS { get => 1 /
↪  backFrameTimeAverage.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; }

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) : base(parameterHolder)
{
    #region Parameters related to View
    currentButton = null; // Initially no panel selected
    obstaclePoints = new
     ↪  ObservableCollection<PolarPoint>();
```

```
controlPoints = new
↪  ObservableCollection<PolarPoint>();
obstacleCentre = new Point(50, 50);
obstaclePointCalculator = new
↪  PolarSplineCalculator();
CreateDefaultObstacle();
controlPoints.CollectionChanged +=
↪  OnControlPointsChanged;
editingObstacles = false;
InVel = parameterHolder.InflowVelocity.Value;
Chi = parameterHolder.SurfaceFriction.Value;


SwitchPanelCommand = new Commands.SwitchPanel(this);
BackendCommand = new
↪  Commands.PauseResumeBackend(this);
EditObstaclesCommand = new
↪  Commands.EditObstacles(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();

backendManager = new BackendManager(parameterHolder);
bool connectionSuccess =
↪  backendManager.ConnectBackend();
if (!connectionSuccess)
{
    MessageBox.Show("Fatal 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;
```

```
    //SendObstacles();
    EmbedObstacles();

    Task.Run(StartComputation);
    backFrameTimeAverage = new
    ↪  MovingAverage<float>(DefaultParameters.FPS_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 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;
}

private bool SendObstacles() // Temporary method to
↪  create a square of obstacle cells
{
    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
        }
    }

    int leftCell = (int)(boundaryLeft * dataWidth);
    int rightCell = (int)((boundaryLeft + boundaryWidth)
    ↪  * dataWidth);
    int bottomCell = (int)((boundaryTop - boundaryHeight)
    ↪  * dataHeight);
    int topCell = (int)(boundaryTop * dataHeight);

    for (int i = leftCell; i < rightCell; i++)
    { // Create a square of boundary cells
        for (int j = bottomCell; j < topCell; j++)
        {
            obstacles[(i + 1) * (dataHeight + 2) + j + 1]
            ↪  = false;
        }
    }


    return backendManager.SendObstacles(obstacles);
}

public void StartComputation()
{
    try
    {
```

```
        backendCTS = new CancellationTokenSource();
        backendManager.GetFieldStreamsAsync(velocity,
         ↪  null, pressure, streamFunction,
         ↪  backendCTS.Token);
    }
    catch (IOException e)
    {
        MessageBox.Show(e.Message);
    }
    catch (Exception e)
    {
        MessageBox.Show($"Generic error: {e.Message}");
    }
}

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));
}

/// <summary>
/// Gets the Smallest Enclosing Rectangle (SER) for the
↪   drawn obstacle.
/// </summary>
/// <returns>The rectangle coordinates in the form: left
↪   x, bottom y, right x, right y.</returns>
private (int, int, int, int) GetObstacleSER()
{
    return (0, 0, 100, 100);
}

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
        }
    }
    (int left, int bottom, int right, int top) =
    ↪   GetObstacleSER();
    for (int i = left; i <= right; i++)
    {
        for (int j = bottom; j <= top; j++)
        {
            PolarPoint polarPoint =
            ↪   (PolarPoint)RecToPolConverter.Convert(new
            ↪   Point(i, j), 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 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();
    }
}

private void BackFPSUpdate()
```

```
        {
            
             ↪ backFrameTimeAverage.UpdateAverage(backendManager.FrameTime);
            OnPropertyChanged(this, nameof(BackFPS));
        }
    }
}
```

ViewModel.cs

```
using System.ComponentModel;
using UserInterface.HelperClasses;

namespace UserInterface.ViewModels
{
    public abstract class ViewModel : INotifyPropertyChanged //
     ↪ The equivalent of SwappableScreen for Views, provides
     ↪ handling of ParameterHolder, as well as some VM-specific
     ↪ features
    {
        public event PropertyChangedEventHandler?
         ↪ PropertyChanged;

        protected void OnPropertyChanged(object? sender, string
         ↪ propertyName)
        {
            PropertyChanged?.Invoke(sender, new
             ↪ PropertyChangedEventArgs(propertyName));
        }

        protected ParameterHolder parameterHolder;

        protected ParameterStruct<T>
         ↪ ModifyParameterValue<T>(ParameterStruct<T>
         ↪ parameterStruct, T newValue)
        {
            parameterStruct.Value = newValue;
            return parameterStruct;
        }

        public ParameterHolder ParameterHolder { get =>
         ↪ parameterHolder; set => parameterHolder = value; }

        public ViewModel(ParameterHolder parameterHolder)
        {
            this.parameterHolder = parameterHolder;
```

```
        }
    }
}
```

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="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}"/>
        <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}" />
        <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="10"
        ↪  Value="{Binding RMax}" />
        <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}" />
        <Label Grid.Row="5" Grid.Column="0">Grid cells per
        ↪  compute unit</Label>
        <helpercontrols:SliderWithValue Grid.Row="5"
        ↪  Grid.Column="1" Maximum="10" ForceIntegers="True"/>
        <Button x:Name="ResetButton" Grid.Row="6" Grid.Column="0"
        ↪  FontSize="16" Command="{Binding ResetCommand}">Reset
        ↪  values</Button>
        <Button x:Name="SaveButton" Grid.Row="6" 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) // Sets the parameter holder
        {
            InitializeComponent();
            DataContext = new
            ↪  AdvancedParametersVM(parameterHolder);
        }
    }
}
```

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="8*" />
            <RowDefinition Height="*" MaxHeight="50" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="3*" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>
        <helpercontrols:ResizableCentredTextBox Grid.Row="0"
        ↪  Grid.Column="0" Text="Simulation configuration"
        ↪  Grid.ColumnSpan="2" />
```

```
<helpercontrols:ResizableCentredTextBox Grid.Row="1"
↪  Grid.Column="0" Text="Parameters" />
<helpercontrols:ResizableCentredTextBox Grid.Row="1"
↪  Grid.Column="1" Text="Simulated objects" />
<Grid Grid.Row="2" Grid.Column="0" Margin="0 0 10 0">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <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">Flow
    ↪  velocity</Label>
    <helpercontrols:SliderWithValue x:Name="SliderInVel"
    ↪  Grid.Row="0" Grid.Column="1" Minimum="0"
    ↪  Maximum="40" Value="{Binding InVel}" />

    <Label Grid.Row="1" Grid.Column="0">Material
    ↪  Friction</Label>
    <helpercontrols:SliderWithValue x:Name="SliderChi"
    ↪  Grid.Row="1" Grid.Column="1" Minimum="0"
    ↪  Maximum="1" Value="{Binding Chi}" />

    <Label Grid.Row="2" Grid.Column="0">Simulation
    ↪  width</Label>
    <helpercontrols:SliderWithValue x:Name="SliderWidth"
    ↪  Grid.Row="2" Grid.Column="1" Minimum="0"
    ↪  Maximum="5" Value="{Binding Width}"  />

    <Label Grid.Row="3" Grid.Column="0">Simulation
    ↪  height</Label>
    <helpercontrols:SliderWithValue x:Name="SliderHeight"
    ↪  Grid.Row="3" Grid.Column="1" Minimum="0"
    ↪  Maximum="5" Value="{Binding Height}" />
```

```
            <Button x:Name="BtnReset" Grid.Row="4"
            ↪  Grid.Column="0" Grid.ColumnSpan="2"
            ↪  Command="{Binding ResetCommand}">Reset
            ↪  parameters</Button>

            <Button Grid.Row="5" Grid.Column="0"
            ↪  Grid.ColumnSpan="2" Command="{Binding
            ↪  CreatePopupCommand}" CommandParameter="{x:Type
            ↪  local:AdvancedParameters}">Advanced
            ↪  parameters</Button>
            <helpercontrols:ResizableCentredTextBox Grid.Row="6"
            ↪  Grid.Column="0" Grid.ColumnSpan="2"
            ↪  MaxHeight="50" Text="Fluid parameters" />
            <Label Grid.Row="7" Grid.Column="0">Reynolds
            ↪  number</Label>
            <helpercontrols:SliderWithValue
            ↪  x:Name="SliderReynoldsNo" Grid.Row="7"
            ↪  Grid.Column="1" Value="{Binding ReynoldsNo}"
            ↪  Minimum="1000" Maximum="100000" />
            <Label Grid.Row="8" Grid.Column="0">Viscosity</Label>
            <helpercontrols:SliderWithValue
            ↪  x:Name="SliderViscosity" Grid.Row="8"
            ↪  Grid.Column="1" Value="{Binding Viscosity}"
            ↪  Minimum="2E-5" Maximum="1" />
            <Button Grid.Row="9" Grid.Column="0"
            ↪  Grid.ColumnSpan="2" Command="{Binding
            ↪  SetAirCommand}">Reset to air at room temperature
            ↪  (20°C)</Button>
        </Grid>
        <StackPanel Grid.Row="2" Grid.Column="1">
            <Button>Select file</Button>
            <TextBlock>No file selected..</TextBlock>
        </StackPanel>
        <Button Grid.Row="3" Grid.Column="1" Command="{Binding
        ↪  ChangeWindowCommand}" CommandParameter="{x:Type
        ↪  local:SimulationScreen}">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)
        {
            InitializeComponent();
            DataContext = new ConfigScreenVM(parameterHolder);
        }
    }
}
```

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=UserInte

         ↪  xmlns:converters="clr-namespace:UserInterface.Converters"
         d:DataContext="{d:DesignInstance
         ↪  Type=viewmodels:SimulationScreenVM}"
         mc:Ignorable="d"
         d:DesignHeight="450" d:DesignWidth="800">
<UserControl.Resources>
    <ResourceDictionary>
        <converters:SignificantFigures
        ↪  x:Key="SignificantFiguresConverter" />
        <converters:VisualisationXCoordinate
        ↪  x:Key="XCoordinateConverter" />
        <converters:VisualisationYCoordinate
        ↪  x:Key="YCoordinateConverter" />
        <converters:XCoordinateDifference
        ↪  x:Key="XCoordinateDifferenceConverter" />
        <converters:YCoordinateDifference
        ↪  x:Key="YCoordinateDifferenceConverter" />
        <converters:PolarListToRectList
        ↪  x:Key="PolarListToRectListConverter" />
        <converters:AbsoluteToRelativeRect
        ↪  x:Key="CoordinateConverter" />
    </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>
    <Button x:Name="BtnRecordingSelect" 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="BtnRecordingSelect">
                        <Setter Property="Background"
                         ↪   Value="#AAAAAA" />
                    </DataTrigger>
                </Style.Triggers>
            </Style>
        </Button.Style>
        <Image Source="/Images/RecordingIcon.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="40" Value="{Binding InVel}"
    ↪  />
    <Label>Material Friction</Label>
    <helpercontrols:SliderWithValue x:Name="SliderChi"
    ↪  Minimum="0" Maximum="1" Value="{Binding Chi}" />
</StackPanel>
<Label Grid.Row="2" Grid.Column="1">
    <Label.Style>
        <Style TargetType="Label">
            <Setter Property="Visibility"
            ↪  Value="Collapsed" />
            <Style.Triggers>
                <DataTrigger Binding="{Binding
                ↪  CurrentButton}"
                ↪  Value="BtnUnitsSelect">
                    <Setter Property="Visibility"
                    ↪  Value="Visible" />
                </DataTrigger>
            </Style.Triggers>
        </Style>
    </Label.Style>
    This is the units selection panel
</Label>
<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>Streamlines</Label>
        <CheckBox x:Name="CBContourLines"
        ↪ IsChecked="{Binding StreamlinesEnabled}" />
    </StackPanel>
    <Label>Streamline contour tolerance</Label>
    <helpercontrols:SliderWithValue
    ↪ x:Name="SliderContourTolerance" Minimum="0"
    ↪ Maximum="0.05" Value="{Binding ContourTolerance}"
    ↪ />
    <Label>Streamline contour spacing</Label>
    <helpercontrols:SliderWithValue
    ↪ x:Name="SliderContourSpacing" Minimum="0"
    ↪ Maximum="0.5" Value="{Binding ContourSpacing}" />
</StackPanel>
<Label Grid.Row="2" Grid.Column="1">
    <Label.Style>
        <Style TargetType="Label">
            <Setter Property="Visibility"
            ↪ Value="Collapsed" />
```

```
                <Style.Triggers>
                    <DataTrigger Binding="{Binding
                    ↪  CurrentButton}"
                    ↪  Value="BtnRecordingSelect">
                        <Setter Property="Visibility"
                        ↪  Value="Visible" />
                    </DataTrigger>
                </Style.Triggers>
            </Style>
        </Label.Style>
        This is the recording settings panel
    </Label>
    <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>
        <StackPanel Grid.Row="3" Grid.Column="0"
         ↪  Grid.ColumnSpan="3" Orientation="Horizontal"
         ↪  HorizontalAlignment="Right">
            <StackPanel.Resources>
                <Style TargetType="Button">
                    <Setter Property="Margin" Value="2.5 0 2.5 5"
                     ↪  />
                </Style>
            </StackPanel.Resources>
            <TextBlock Margin="0 0 5 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>Save as image</Button>
            <Button>Save as video</Button>
            <Button Content="{Binding BackendButtonText}"
             ↪  Command="{Binding BackendCommand}"
             ↪  d:Content="Pause Simulation" />
        </StackPanel>
    </Grid>
</UserControl>
```

SimulationScreen.xaml.cs

```
using System;
using System.ComponentModel;
using System.Diagnostics;
using System.Windows;
using System.Windows.Controls;
```

```csharp
using System.Windows.Input;
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 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)
        {
            InitializeComponent();

            viewModel = new SimulationScreenVM(parameterHolder);
            DataContext = viewModel;

            RecToPolConverter = new();
            pointsPlaced = false;
            isCentreMoved = false;
            ViewModel.PropertyChanged +=
            ↪  OnViewModelPropertyChanged;
        }

        #region Private helper methods
        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 ComputeShaderManager computeShaderManager;

        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 hFieldEBO;

        private int hContourVAO;
        //private int hContourEBO;

        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)]);
    //computeShaderManager = new
     ↪   ComputeShaderManager("shader.comp");

    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)
{
    //GL.ClearColor(0.1f, 0.7f, 0.5f, 1.0f);

    vertices = GLHelper.FillVertices(dataWidth,
     ↪   dataHeight);
    fieldIndices = GLHelper.FillIndices(dataWidth,
     ↪   dataHeight);
    contourIndices =
     ↪   GLHelper.FindContourIndices(streamFunction,
     ↪   parameterHolder.ContourTolerance.Value,
     ↪   parameterHolder.ContourSpacing.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);
    //Trace.WriteLine(GL.GetError().ToString());
    GLHelper.BufferSubData(fieldParameters.field,
    ↪  vertices.Length);
    //Trace.WriteLine(GL.GetError().ToString());

    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.ContourSpacing.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");
            }
        }
    }
}
```

ComputeShaderManager.cs

```
using OpenTK.Graphics.OpenGL4;

namespace Visualisation
{
    public class ComputeShaderManager : ShaderManager
    {
        public ComputeShaderManager(string path) : base(new
        ↪  (string, ShaderType)[] { (path,
        ↪  ShaderType.ComputeShader) }) { }
    }
}
```

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
↪  spacingMultiplier, uint primitiveRestartSentinel, int
↪  width, int height)
{
    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 * width +
         ↪  currentHeight] - targetValue >
         ↪  contourTolerance) && !(targetValue -
         ↪  streamFunction[i * width + currentHeight]
         ↪  > contourTolerance)) // Add in another
         ↪  condition to avoid floating point error
         ↪  (which should be always less than contour
         ↪  tolerance)
        {
            levelSets[levelSetNum].Add((uint)(i *
             ↪  width + currentHeight));
            continue;
        }
        if (streamFunction[i * width + 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 * width +
    ↪  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 * width +
    ↪  currentHeight] > targetValue ||
    ↪  targetValue - streamFunction[i *
    ↪  width + currentHeight] <
    ↪  contourTolerance) // Within tolerance
    ↪  either side of target
    {
        levelSets[levelSetNum].Add((uint)(i *
         ↪  width + 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 * width +
    ↪  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 *
                ↪  width + currentHeight] ||
                ↪  streamFunction[i * width +
                ↪  currentHeight] - targetValue <
                ↪  contourTolerance)
                {
                    levelSets[levelSetNum].Add((uint)(i *
                    ↪  width + 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);
        }
    }
}
```