THE UNIVERSITY OF NEW SOUTH WALES

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

A Parallel Fluid Flow Simulator in Haskell

Ben Lambert-Smith

Thesis submitted as a requirement for the degree
Bachelor of Science, Honours (Computer Science)

Supervisor: Manuel Chakravarty

Assessor: Ben Lippmeier

Submitted: May 31, 2011

Abstract

Fluid simulators have been an area of intense research for many years. Many solutions have been created that aim to render highly realistic simulations with enough performance to allow real-time interaction with users. In recent years an abundance of multi-core processors have become available. Using these processors with parallelism provides a means to boost performance.

In this thesis we explore the implementation of a popular simulator algorithm by Jos Stam using a Haskell parallel array library, Repa. With the use of Repa it was aimed to gain performance from parallelism without losing declarativeness relative to sequential implementations.

Acknowledgements

I would like to thank the PLS group at UNSW for all the support and feedback throughout this thesis. In particular I'd like to thank Manuel Chakravarty, for inspiring me to do this, and Ben Lippmeier, for pushing me in the right direction when I got lost. Thanks also go to Jos Stam for his simple, yet effective, stable fluids algorithm. Finally, thank you to the friends and family who provided feedback on my writing and use of language.

Contents

List of Figures

CHAPTER 1

Introduction

There have been many fluid simulators designed and created over time, with a variety of methods used to balance performance and accuracy. However, the increase in performance commonly comes at a cost to code readability and to declarativeness. This thesis is an attempt to develop a fluid simulator that retains a strong and obvious link to the original equations which are its mathematical basis, while still performing competitively with a sequential version, using an equivalent algorithm and allowing real-time user interaction.

To achieve this extra performance we use a well known sequential algorithm by Jos Stam [Sta03] and implement it with parallelism. Using parallelism can cause a loss of declarativeness, compared to a sequential implementation, due to the overheads of managing multiple threads of execution. Our method of implementation, however, is in Haskell, a high-level functional language, using a recently developed parallel array library, Repa. With the use of Haskell and the Repa library, we aimed to keep our code declarative while using a parallel implementation.

Our benchmarking shows that we can achieve competitive performance to the C implementation written by Jos Stam when using two threads, and can achieved better than C performance with three or more threads. However, the scaling of performance with the increase of threads is not ideal, but future work could be done to increase the parallelism of the simulator.

1.1 Overview

In Chapter 2 we explore fluid simulators in general, discussing general concepts relating to the field, and in Chapter 3 we define the specific mathematics and algorithm of the fluid simulator that will form the basis of our implementation.

As the Haskell library, Repa, plays a large role in the implementation of our fluid simulator we look into the workings and use of the library in Chapter 4.

We will then discuss the specific aims we set out to achieve with this thesis in Chapter 5 before looking at some samples of other research in the area of fluid simulation in Chapter 6.

Chapters 7 and 8 will describe how our implementation was designed and written, and will introduce some of the shortcomings found during the development process.

Our results and analysis of performance are then described in Chapter 9 with a closer look at the shortcomings found in our implementation. We will also present some ideas for how the implementation could be improved.

Finally, some closing remarks and ideas for future work will be shown in Chapter 10 with a reflection on the overall outcome of the thesis.

CHAPTER 2

Fluid Simulators: An Overview

Fluid simulators are, in general, developed for two reasons:

- Engineering
- Aesthetics

Fluid simulators for engineering purposes, such as modelling the aerodynamics of a wing, require an accurate correlation between the simulators behaviour and real life. For aesthetic purposes, such as in games, the simulator does not need to be a physically accurate model provided the simulation appears plausibly realistic.

In this chapter we discuss fluid simulators in general. As the study of computational fluid dynamics is a vast research area we will focus on simulators that use the Navier-Stokes equations, as these equations describe a larger variety of fluids than other fluid equations, such as Euler's equations. The Navier-Stokes equations also form the mathematical basis for the fluid simulator we will be implementing. The descriptions in this chapter will be in only enough detail to give an intuition, with a more in depth mathematical look at fluids to be described in Chapter 3.

2.1 Conservation and compressibility

There are many ways that fluids can be modelled and simulated, some of which we will be looking at in this chapter. What they all share in common is that they aim to simulate the laws of physics. That is, a fluid conserves its mass and its momentum. Mass conservation dictates that a fluid's mass remains constant over time when in a closed system. Momentum conservation means that a fluid's momentum is only changed equal to the force that is applied to it. As we are dealing with a simulated reality these rules can be bent and broken, but for the most part should hold true. The Navier-Stokes equations specify this conservation of momentum and mass.

Another characteristic of fluid simulators is the compressibility of the fluid. Compressibility occurs when a force changes the volume of the fluid. In reality all fluids are compressible, otherwise sound waves would not propagate. However, causing visible changes in most fluids from compressibility requires an unusually large force, like a sonic boom, and modelling this effect increases the complexity of the simulator considerably. In general, most simulators created for aesthetic purposes make a fluid *incompressible*. This does not change the behaviour of most fluids drastically, as most fluids have low compressibility, and allows more computation time to be allocated to other areas of the simulation. Bridson et. al. in *Fluid Simulation* [BFMF06, Section 1.4] goes into further details regarding incompressibility, however these additional mathematical details are omitted here.

2.2 Navier-Stokes equations

The Navier-Stokes equations are a general mathematical description of fluid flow over time. The equations can describe compressible and incompressible fluids. Simulating a compressible fluid is used when we need a fluid to react to pressure stimuli, an example of this would be modelling sound waves through a fluid medium. More commonly used are the incompressible Navier-Stokes equations from [Sta99] shown in Figure 2.1.

The Navier-Stokes equations may seem daunting at first but we will break them down into their constituent parts and explain each component's purpose. In Chapter 3 we will look at how these equations are used in the fluid simulator being implemented. As a general explanation, the Navier-Stokes equations describe a fluid's motion and impose mass and momentum conservation on the fluid.

The first equation is the incompressibility constraint, and imposes that the mass does not change over time. The second equation enforces the conservation of momentum and consists of four components:

- Advection

$$\nabla \cdot \mathbf{u} = 0$$
$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \frac{1}{\rho}\nabla p + \nu \nabla^2 \mathbf{u} + \mathbf{f}$$

Figure 2.1: Incompressible Navier-Stokes equations

- Pressure
- Viscosity
- External forces

2.2.1 Advection

Advection is a description of how the velocity of each particle in the fluid affects the fluid's movement as a whole. It is represented in the Navier-Stokes equations as:

$$-(\mathbf{u} \cdot \nabla)\mathbf{u} \tag{2.1}$$

Where $\mathbf{u}$ is the velocity field for the fluid and $(\mathbf{u} \cdot \nabla)$ is the divergence operator, measuring if $\mathbf{u}$ is converging or diverging. See Appendix B for further details about $(\mathbf{u} \cdot \nabla)$.

2.2.2 Pressure

Pressure is the force the fluid exerts on itself from its mass. It describes the fluid's change in density depending on the position of the point in the fluid.

$$-\frac{1}{\rho}\nabla p \tag{2.2}$$

Where ρ is the fluid's density and ∇p is the direction the pressure is affecting.

2.2.3 Viscosity

The viscosity of a fluid describes how quickly a fluid loses its shape. Bridson, Fedkiw and Muller-Fischer in *Fluid Simulation* [BFMF06] state that this component is commonly removed from Navier-Stokes fluid simulators to reduce complexity as the viscosity does not change the visuals of the fluid. The viscosity rate is described by the coefficient ν, which affects all dimensions of the fluid evenly. This coefficient's value is set depending on how viscous the fluid is to be.

$$\nu \nabla^2 \mathbf{u} \tag{2.3}$$

∇^2 is the Laplace operator, and $\nabla^2 \mathbf{u}$ describes the divergence of the gradient of $\mathbf{u}$.

2.2.4 External forces

The external forces make up everything else that may affect the movement of the fluid simulator. A common external force to use is gravity, though other forces like the force exerted by a fan or propeller would also be included in this component.

$$\mathbf{f} \tag{2.4}$$

2.3 Discretisation viewpoints

Because it would be computationally infeasible to apply the Navier-Stokes equations to every physical particle in a fluid there are two viewpoints for how the particles of a fluid are discretised. They are the Lagrangian and Eulerian viewpoints.

The Lagrangian viewpoint represents a fluid by a collection of particles, with each particle having a location and velocity. This fluid model is similar to how fluids are physically composed, though each particle in the Lagrangian viewpoint is an abstraction of what would be the particles in an area around each Lagrangian particle. Grouping the Lagrangian particles into one body can be done in a variety of ways and is dependent on the particular implementation. One benefit of using the Lagrangian viewpoint is that conservation of mass, the prevention of mass being spontaneously created or destroyed in a simulator, is built into the computational model as the number of Lagrangian particles remains constant. Though if the mass that each Lagrangian particle represents is not static the mass conservation becomes a concern again.

In contrast to the Lagrangian viewpoint, the Eulerian viewpoint uses a mesh independent of the fluid, which samples the density and velocity of the fluid at each point on the mesh. This data is then used for computing the movement of the fluid around these points, and rendering the fluid based on the densities at each point.

Both viewpoints have benefits and drawbacks, while a Lagrangian implementation would be easier to understand as it is the way we generally imagine a fluid's composition, the Eulerian viewpoint is much easier to implement using array computations, due to the mesh being static. The algorithms designed for Eulerian discretisation define computations to pass values between these statically placed cells. A problem with an Eulerian implementation occurs when the simulation is only sparsely occupied by a fluid. In this instance a Lagrangian method is more appropriate as the fluid computations only occur at the location of the fluid, rather than over the entire simulation area.

CHAPTER 3

Stam's Stable Fluids

The particular solver we will be implementing for our thesis is based around the method first described by Jos Stam in *Stable Fluids* [Sta99] and further developed in *Real-Time Fluid Dynamics for Games* [Sta03].

Stam's solver is predominantly an Eulerian fluid simulator, it uses a grid with a static structure to represent the space the fluid is being simulated in. There is one component, advection, that does not follow this structure explicitly, which will be explained in Section 3.3.3. Each grid cell in the simulator contains two values, the density and the velocity. We will look at the difference between these two in detail in Section 3.4.

As has been mentioned previously, this method uses the Navier-Stokes equations defined in Chapter 2. In this chapter we will look at how the Navier-Stokes equations are transformed into the form used in the simulator, and how each component of the equations are computed.

3.1 Stability

Stability in numerical simulations refers to the simulator being robust under different time steps. Conversely, instability in numerical simulations refers to the limitations on the size of the time step that exists before the simulator has a tendency to "blow-up". "Blowing-up" refers to the oscillation and divergence of the simulated fluid values when a time step that is too large is chosen.

Stam's method is one of the most widespread methods for building stability into a simulator.

Other methods require a limit to be put on the time step. However Stam's method can take any time step, though larger time steps will decrease the accuracy of the simulation.

3.2 Derivation

Starting with the Navier-Stokes equations shown in Chapter 2:

$$\nabla \cdot \mathbf{u} = 0 \tag{3.1}$$

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \frac{1}{\rho}\nabla p + \nu \nabla^2 \mathbf{u} + \mathbf{f} \tag{3.2}$$

We will derive the equations used for the implemented simulator, following the work of Jos Stam.

To implement the simulator we need to combine the two equations. To do so we will use a result from mathematics known as the Helmholtz-Hodge Decomposition, described in [CM92, p. 36]. This result states that any velocity field can be uniquely decomposed into the sum of a mass conserving field and a gradient field.

If we use the decomposition to convert a velocity field into a mass conserving field we can use this to combine the mass conservation constraint from (3.1) with (3.2). So from the aforementioned Helmholtz-Hodge Decomposition we split a velocity field, $\mathbf{w}$, into the sum of an incompressible velocity field, $\mathbf{u}$, and a gradient field ∇q shown in (3.3):

$$\mathbf{w} = \mathbf{u} + \nabla q \tag{3.3}$$

As $\mathbf{u}$ in (3.3) is mass conserving, it meets the incompressibility and mass conservation constraint from (3.1). So if we can convert $\mathbf{w}$ into $\mathbf{u}$ then we will be able to combine the two Navier-Stokes equations. We do so by rearranging the equations, as seen in (3.4), so that an incompressible field is the difference between a velocity field and its unique gradient field:

$$\mathbf{u} = \mathbf{w} - \nabla q \tag{3.4}$$

We turn (3.4) into a function, $\mathbf{P}$, and apply it to (3.2) as shown in Figure 3.1. For the equation, Stam used that fact that $\mathbf{P}\mathbf{u} = \mathbf{u}$ and $\mathbf{P} \cdot \nabla p = 0$, eliminating the pressure component from the equations.

$$\frac{\partial \mathbf{u}}{\partial t} = \mathbf{P}(-(\mathbf{u} \cdot \nabla)\mathbf{u} + \nu \nabla^2 \mathbf{u} + \mathbf{f})$$

Figure 3.1: Derived Navier-Stokes equation

$$\mathbf{w}_0(\mathbf{x}) \overbrace{\rightarrow}^{\text{add forces}} \mathbf{w}_1(\mathbf{x}) \overbrace{\rightarrow}^{\text{diffuse}} \mathbf{w}_2(\mathbf{x}) \overbrace{\rightarrow}^{\text{advect}} \mathbf{w}_3(\mathbf{x}) \overbrace{\rightarrow}^{\text{project}} \mathbf{w}_4(\mathbf{x})$$

Figure 3.2: Pipeline used for manipulating the field each time step

3.3 Solving components

We now look at the method by which each component in Figure 3.1 is solved. The procedure to follow is a pipeline, manipulating the field by each component one at a time. Figure 3.2 gives a diagram of this pipeline. Where $\mathbf{w}_i$ ($i \in \{1, 2, 3, 4\}$) is the field at each particular stage of the pipeline, $\mathbf{w}_0$ is the field after the previous time step, and $\mathbf{w}_i(\mathbf{x})$ is the field's value at position $\mathbf{x}$. We denote the current time by t and with each full completion of the pipeline the simulation moves forward by the time step, denoted by Δt.

Note that between [Sta99] and [Sta03] the ordering of the pipeline changed slightly. In [Sta99] advection is done before diffusion, however for our implementation we will follow what is being described here and in [Sta03].

3.3.1 Addition of forces

Adding forces is the simplest component to solve. We assume that the force remains constant over the time step, Δt, which gives us the following transition function:

$$\mathbf{w}_1(\mathbf{x}) = \mathbf{w}_0(\mathbf{x}) + \Delta t \cdot \mathrm{f}(\mathbf{x}, t) \tag{3.5}$$

The function $\mathrm{f}(\mathbf{x}, t)$ adds the force at time t to position $\mathbf{x}$. As more (or less) force is added depending on how long the time step is, the time step is used as a scaling factor on the force added.

3.3.2 Diffusion

Diffusion, and similarly viscosity, are simulated in this stage. In Stam's implementation diffusion is simulated as the grid cells exchanging densities in a ratio specified by the diffusion coefficient and, similarly, viscosity is simulated as the grid cells exchanging velocities in a ratio specified by the viscosity coefficient. Both coefficients are represented in (3.6) by ν. A means of visualising this is shown in Figure 3.3 where a grid cell gives a portion of its value to neighbouring cells and gets a portion in return. The actual amount transferred is defined by a ratio calculated by multiplying the time step, diffusion/viscosity rate and the Laplace operator. The Laplace

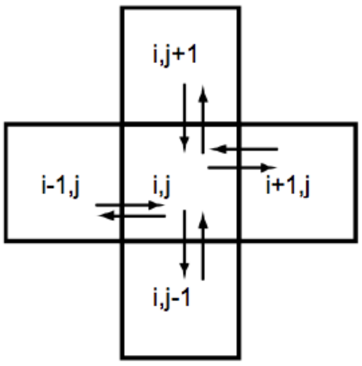


Figure 3.3: Exchange of values between neighbours in diffusion stage. [Sta03]

operator describes the divergence of the gradient, and is further explained in Appendix B. Mathematically the step is represented as:

$$(\mathbf{I} - \nu \Delta t \nabla^2)\mathbf{w}_3(\mathbf{x}) = \mathbf{w}_2(\mathbf{x}) \tag{3.6}$$

In the equation, **I** is the identity operator. This is a sparse linear system for the unknown field $\mathbf{w}_3$. To approximate this Stam suggests finding the value that when diffused backwards in time would result in the value we started with. Which is defined as:

$$\mathrm{x}^0_{i,j} = \mathrm{x}_{i,j} - a(\mathrm{x}_{i+1,j} + \mathrm{x}_{i-1,j} + \mathrm{x}_{i,j-1} + \mathrm{x}_{i,j+1} - 4\mathrm{x}_{i,j}) \tag{3.7}$$

Where $\mathrm{x}^0_{i,j}$ is the value at position (i, j) at the start of the stage, a is the previously mentioned ratio, $\mathrm{x}_{k,l}$ corresponds to the cell value at position (k, l) in the grid, and $\mathbf{x}$ from (3.6) corresponds to $\mathrm{x}_{i,j}$.

Rearranging (3.7) to make $\mathrm{x}_{i,j}$ the subject we get:

$$\mathrm{x}_{i,j} = \frac{\mathrm{x}^0_{i,j} + a(\mathrm{x}_{i+1,j} + \mathrm{x}_{i-1,j} + \mathrm{x}_{i,j-1} + \mathrm{x}_{i,j+1})}{1 + 4a} \tag{3.8}$$

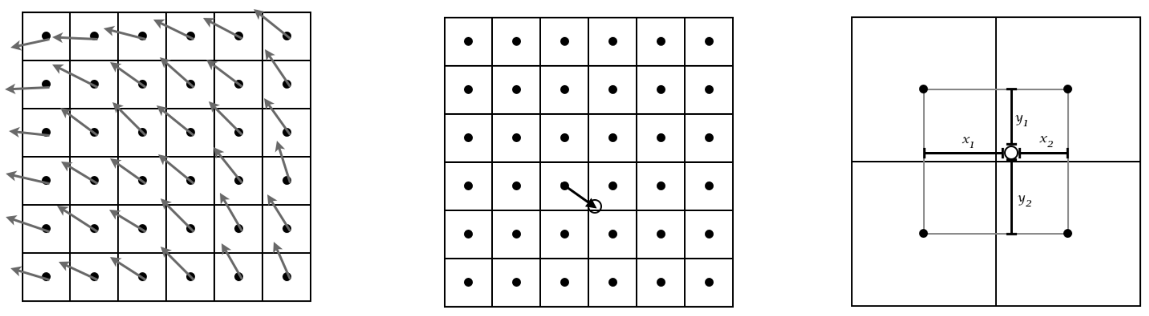
The aim of this equation is to approximate the Laplacian, time step and diffusion rate coefficients ($\nu \Delta t \nabla^2$). Putting this equation into an iterative solver allows us to approximate this coefficient, gaining accuracy through each iteration.

3.3.3 Advection

The advection stage moves the fluid values based on their velocities. Mathematically this component is represented as follows:

$$\mathbf{w}_2(\mathbf{x}) = \mathbf{w}_1(\mathbf{p}(\mathbf{x}, -\Delta t)) \tag{3.9}$$

Where $\mathbf{p}(\mathbf{x}, -\Delta t)$ is the path of the density at $\mathbf{x}$ one time step ago.



(a) Vector field

(b) Backtracking a vector

(c) Take weighted average of surrounding cell centers

Figure 3.4: Advection stage backtracking

To simulate this step Stam suggests backtracking where the value was in the previous time step based on the negative of the current velocity. The value is then linearly interpolated from the surrounding four grid cells. Linear interpolation produces a weighted average of these four grid cells, where the weighting corresponds to how close the backtracked position is to the center of each grid cell. As shown in Figure 3.4, with 3.4(c) being a magnification of cells surrounding the location pointed to in 3.4(b).

This way we calculate the values that would end up at the centre of each particular grid cell, a simpler method than if we calculated values moving forward in time. Solving advection in this manner uses a semi-Lagrangian method, as we do not rely solely on a static grid to simulate the particles.

The advection stage gives the solution its stability, and in subsequent research it is the method commonly taken from Stam's research.

3.3.4 Projection

As mentioned in Section 3.2, the projection stage turns the field from a regular velocity field into a mass conserving one. The transition function is described as:

$$\mathbf{w}_4 = \mathbf{w}_3 - \nabla q \tag{3.10}$$

The gradient field, ∇q, is found by again using Gauss-Seidel relaxation. Though we do not use the same coefficient in this application of relaxation as was used for the diffusion component. The gradient field is calculated by finding the direction of steepest descent in the original field. The equation used in the iterative linear solver is similar to that used by diffusion:

$$\mathrm{x}_{i,j} = \frac{\mathrm{x}^0_{i,j} + (\mathrm{x}_{i+1,j} + \mathrm{x}_{i-1,j} + \mathrm{x}_{i,j-1} + \mathrm{x}_{i,j+1})}{4} \tag{3.11}$$

As can be seen in (3.10) this gradient field is then subtracted from the original field, $\mathbf{w}_3$, to get the mass conserving and incompressible field $\mathbf{w}_4$.

3.4 Velocities and densities

As mentioned at the start of the chapter, the structure representing a fluid is a field of densities and a field of velocities. The fluid itself is described by the velocity field, while the density field describes particles moving within the fluid, for example smoke particles in the air. Each field is passed through the pipeline shown in Figure 3.2. There is a difference between how the two fields are treated though, as the density field does not pass through the projection stage. Only the velocity field needs to be passed through to keep the simulation mass conserving. Stam describes two equations in [Sta03], which are included below, to describe the differing type of values being manipulated. The first describes the equation for evolving the velocities:

$$\frac{\partial \mathbf{u}}{\partial t} = \mathbf{P}(-(\mathbf{u} \cdot \nabla)\mathbf{u} + \nu\nabla^2\mathbf{u} + \mathbf{f}) \tag{3.12}$$

And the second describes the equation for evolving the densities:

$$\frac{\partial \rho}{\partial t} = -(\mathbf{u} \cdot \nabla)\rho + \kappa\nabla^2\rho + S \tag{3.13}$$

Where $\mathbf{u}$ is the velocity, ρ is the density, ν and κ represent the diffusion rate for velocities and densities respectively, and $\mathbf{f}$ and S represent the forces or sources to be added to the respective fields. Notice that the projection operator is not present in (3.13).

CHAPTER 4

Repa Library

Haskell, a lazy and functional programming language, is currently being extended by the Data Parallel Haskell project. This project is made up of a number of sub-projects which aim to extend and improve Haskell's handling of data parallelism. Data parallel programming involves the same function being performed over different data items at the same time as each other. One of these sub-projects, Repa, is a data parallel array library. As this library will be used prominently in the implementation of this thesis, we will be describing the details of it in this chapter. The library was described in two papers: *Regular, Shape-polymorphic, Parallel Arrays in Haskell* [KCL+10], which described the original library, and *Efficient Parallel Stencil Convolution in Haskell* [LKPJ11], which expanded upon the original library by defining stencil operations that could be applied to Repa's arrays.

4.1 Original library

Repa, first described in [KCL+10], is an array computation library written completely in Haskell with the aim of providing a fast and pure interface for array manipulation. A pure function is one without side-effects and is deterministic. In reality all programs are impure, however Haskell allows the impurity to be managed by requiring impure areas to be declared as such. By making Repa's interface pure into the underlying impure operations, developers can define their programs to be pure, limiting the areas where bugs can be introduced from impurity.

The library provides support for regular, shape-polymorphic arrays. We will now look at the structure and features that characterise the library.

4.1.1 Shape-polymorphism

A function which accepts a shape-polymorphic array is one which can accept an array of arbitrary dimensions, and arbitrary extent in any of those dimensions.

Repa defines an array data type as:

```
Array sh a
```

Where `sh` is the shape of the array and `a` is the type being held in the array. A shape is defined in terms of the length of each dimension, beginning with the description of a scalar:

```
Z
```

And chaining lengths for subsequent dimensions to the right of it with the new `(:.)` operator:

```
Z :. Int :. Int ...
```

An example of this would be the description of a cube-shaped array with sides of length 10:

```
Z :. 10 :. 10 :. 10
```

As was mentioned earlier, these arrays are regular. Therefore the length of arrays can't vary in the dimension.

4.1.2 Manifest and delayed arrays

Haskell is a lazy language, which means that it delays executing code until the result is required. Before the code is executed it is stored in a *thunk*. A thunk is a piece of unexecuted code, where all the operations that have built up due to the laziness of Haskell are stored until the results of these operations are needed. A standard Haskell array is an array of thunks, with each thunk being the unexecuted code for the array element. As elements are accessed in the array, each thunk is executed and leaves the result in the thunk's place in the array. However Repa uses a different representation of arrays.

Repa's representation of arrays makes the entire array a thunk. Accessing any element in the array causes all the array elements to be calculated, which is the source of Repa's parallelism. Calculating all the elements in an array when one element is accessed is most useful when manipulating a dense array, where the majority of elements in the array will be accessed and processed. Figure 4.1 shows the difference in representation between the standard Haskell arrays and Repa arrays.

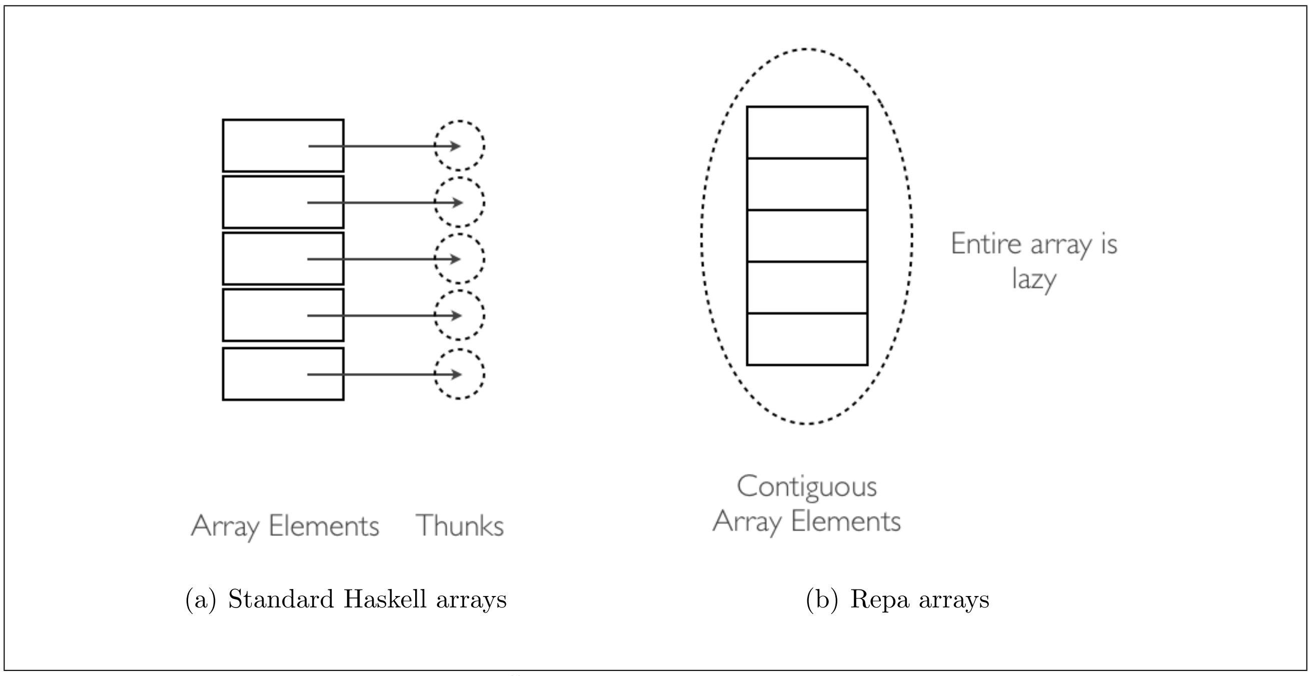


Figure 4.1: Difference between array representations

When the Repa array is a thunk it is said to be in delayed form. When the array elements are calculated it converts the array into a manifest array which is a contiguous series of results, similar to how an array is represented in C.

4.1.3 Data parallelism

As mentioned at the beginning of the chapter, Repa's operations are data parallel. This parallel computation occurs when transforming the Repa array from delayed to manifest form. When the transformation occurs, the array is partitioned evenly and given to a *gang of threads*. The exact operation of this gang is described in the paper *Data Parallel Haskell: a status report* [CLPJ+07]. We give an overview of how the gang parallelism works here. Each thread in the gang operates in a loop as follows:

1. Waits for a computation to be given to it
2. Executes the computation
3. Signals its completion
4. Blocks until the next computation arrives

4.1.4 Controlling parallelism

As Haskell is a lazy language it waits to execute a block of code until the result is required. To get the best results it is better that the programmer can specify when execution should occur. To this end there is the `force` command which indicates that a conversion from a delayed array to a manifest array should occur. However, lazy execution of `force` can result in the array elements remaining unexecuted. To ensure that the array results are executed, a combination of `force` and `deepSeqArray` must be used. The type definition of `deepSeqArray` is:

```
deepSeqArray :: Shape sh => Array sh a -> b -> b
```

Calling `deepSeqArray` creates a dependance on `Array sh a` being executed before `b`, ensuring that the array is executed at that point. Using this combination is important for our fluid simulator as each stage requires the execution of the previous stage, and delaying execution is detrimental to performance.

4.1.5 Categories of operations

There are four categories of array operations provided by the authors of Repa. They are:

- Structure-preserving: operations which alter the data in each element, but do not change the shape of the array
- Reductions: reduce the size of the array upon completion of the operation and may also change the type of values in the array
- Index space transformations: change the shape of the array, but do not change the type of elements
- General traversals: can change the shape and type of the array

Some of these operations can specify a subsection of the array, or generalise the rank of the array. To do so, the authors defined “slices” as the means to describe these.

4.1.6 Slices

For circumstances where the user of the Repa library does not wish to operate on the entire library, or wishes to extract a portion of the library, Repa provides *slices* to allow selection of a subsection of the array. Slices are used with the index space transformation operators `extend` and `slice` to describe how to increase or decrease the size of the array respectively.

Specifying a slice is similar to specifying the extent of an array. It uses `Z` as its basis, but adds `All` which can be used in place of a numerical index. Using `All` indicates to generically grab or leave that dimension, depending on the operation. For example we use `extend` to expand a 2-dimensional array into a 3-dimensional array when processing our simulation for display, see Section 7.3.2 for details of this. To do so we specify a slice as:

```
(Z :. All :. All :. 3)
```

To select all the elements in the first two dimensions and extend the array into the third dimension with extent 3.

Developing generic shape polymorphic functions sometimes requires limitations to be placed on the dimensionality. Repa accommodates this by allowing a minimal dimensionality to be set in the type definition of a function. For example, the Repa function `sum`, which sums the values in one dimension, uses this in its type definition:

```
sum :: (Shape sh, Elt a, Num a)
    => Array (sh :. Int) a
    -> Array sh a
```

Which limits the `Array` taken as argument to being at least one dimension.

4.2 Stencil extension

An extension to the Repa library was designed to meet a shortcoming when using stencils with Repa arrays. A stencil is a relatively small array made of coefficients. An example from [LKPJ11] is the 3×3 Laplacian stencil:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

The stencil passes over the array stopping at each cell. The coefficients are applied to the overlapping array cells and the result is placed in the current cell. To account for the stencil overlapping the edge of the array in border cases, checks must be made before accessing elements in the array to avoid accessing a position which is not actually in the array. When processing large arrays or, in the case of our simulator, processing an array as quickly as possible, these checks become quite expensive. This cost is especially high when considering that only a small fraction of the array needs these checks.

To rectify this performance problem, Repa was extended in *Efficient Parallel Stencil Convolution in Haskell* [LKPJ11] to accomodate the application of stencils on Repa arrays. Applying this stencilling in our own simulator proved very effective in boosting performance and we explore the two main boosts to performance gained by using Repa's stencilling. The complete definition of the Array type is included in Figure 4.2 for reference.

```
data Array sh a
   = Array       { arrayExtent  :: sh
                 , arrayRegions :: [Region sh a] }
data Region sh a
   = Region      { regionRange  :: Range sh
                 , regionGen    :: Generator sh a }
data Range sh
   = RangeAll
   | RangeRects  { rangeMatch   :: sh -> Bool
                 , rangeRects   :: [Rect sh] }
data Rect sh
   = Rect sh sh
data Generator sh a
   = GenManifest { genVector    :: Vector a }
   | forall cursor.
     GenCursored { genMake      :: sh -> cursor
                 , genShift     :: sh -> cursor -> cursor
                 , genLoad      :: cursor -> a }
```

Figure 4.2: Full definition of Repa arrays from [LKPJ11]

4.2.1 Partitioning

The strategy Lippmeier et. al. employed for minimising the number of bounds checks being performed was to change the underlying structure of the Repa array. The type of Repa arrays is still:

```
Array sh a
```

However, the constructor for the array is now a record containing the extent of the array, `sh`, and an array of `Region` types. A `Region` defines subsections of the array as a list of rectangular areas, or specifies that it covers the entire array.

Using the new partitioning structure, arrays can be partitioned into an edge region and an interior region, with the edge regions performing bounds checks when applying a stencil, and the interior region being safe to process without them.

4.2.2 Sharing

Lippmeier et. al. introduced another boost to performance for stencilling operations besides the use of partitioned arrays. Referring to Figure 4.3, it can be seen that as a 3×3 stencil is

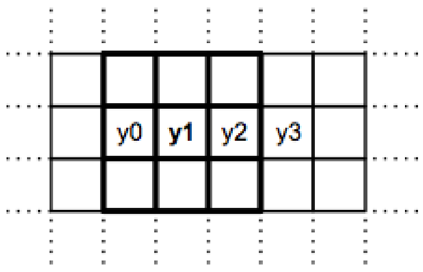


Figure 4.3: Overlapping cells diagram from [LKPJ11]

being applied to positions y_0 to y_3 there are cells being used multiple times in the calculation of these positions. For the example in Figure 4.3 there are 36 indexing calls when performing the stencil naively. Identifying these redundant indexes [LKPJ11] describes a method of removing the redundancy and in the example of Figure 4.3 can cut down the indexing calls to 18.

To do this required an additional change to the underlying structure of Repa by replacing the delayed arrays with *cursored* arrays. As described in [LKPJ11] “a cursor is an abstract representation of an index into the array”. The cursor is used with the `GenCursored` constructor for a `Generator`, which describes how to generate the elements in the array.

When applying the stencil the elements are evaluated in blocks, with the stencil code explicitly operating on four consecutive elements. To recover sharing in these blocks the stencilled code must be compiled with the Global Value Numbering optimisation. This optimisation is not available with a typical compilation of GHC, but must use the LLVM back-end which does use this optimisation.

CHAPTER 5

Aims and Design

5.1 Problem definition

Fluid simulators have been an active research area for many years due to their practical and aesthetic uses. With this high level of interest there have been many solutions generated, some of which we will sample in Chapter 6.

This thesis will focus on the implementation of a fluid simulator using approximations to the Navier-Stokes equations, and which we discussed in Chapter 3, that Jos Stam derived in his two papers: *Stable Fluids* [Sta99] and *Real-Time Fluid Dynamics for Games* [Sta03]. These equations are repeated in Figure 5.1.

To complement the algorithm outlined in his two papers Stam provided an implementation of it in C. Our simulator is developed in the Haskell programming language and our aim is that the simulator runs with performance comparable to Stam's C implementation. It is also an aim that our simulator's code be more declarative than the C version, making the link between the mathematical formulas in Figure 5.1 and the implementation clearer.

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} + \nu \nabla^2 \mathbf{u} + \mathbf{f}$$
$$\frac{\partial \rho}{\partial t} = -(\mathbf{u} \cdot \nabla)\rho + \kappa \nabla^2 \rho + S$$

Figure 5.1: Stam's Navier-Stokes Equations

The system's design can be separated into three sections: back-end, front-end and extensions.

5.2 Overall design

5.2.1 Front-end

For the front-end of the simulator we use a graphical Haskell library developed by Ben Lippmeier called Gloss. The library provides a pure and abstract interface for using the OpenGL Haskell library, which in turn is a Haskell library providing hooks into the C OpenGL library.

The Gloss library was extended to provide the functions required to display the simulator. We use a bitmap data type to display the simulator's data, which is a 2-dimensional array of RGBA values.

The simulator also provides the functionality required for a user to interact with the simulation in real-time. Gloss provides hooks into detecting user interaction, and allows us to define how the interactions affect the simulation.

5.2.2 Back-end

The back-end of the simulator is what performs the numerical analysis and manipulation necessary to describe how the fluid moves and reacts. As mentioned in Chapter 2, there are many methods and implementation strategies which can be used to describe the motion of fluids. For our thesis we use the method by Stam that we explored in Chapter 3.

The Haskell language has more overheads in its processing than C. This is due to additional services such as garbage collection and the increased generality of the machine code generated, as Haskell is a high-level language. These additional services do give some benefits to writing code in Haskell, most prominent of which are language safety and declarativeness, but developing a simulator in Haskell that runs in comparable time to the C implementation requires a means of getting additional processing power.

To get this additional performance we use the Repa library described in Chapter 4. This library allows us to add parallelism to our Haskell implementation, while still maintaining declarativeness, and running the simulation on numerous cores gives us the boost in processing power to run in comparable time to the sequential C implementation.

5.2.3 Extensions

There are a number of extensions that could be implemented on the basic simulator. However, due to additional time required for increasing performance these extensions were not implemented in the scope of this thesis. We define some possible extensions below but these are left for future work.

Extending to 3-dimensions

Extending the aimed 2-dimensional simulator to 3-dimensions would require taking advantage of Repa's shape polymorphism, described in Section 4.1.1. Extending the dimensionality of the simulator would have some impacts on the implementation, though should not affect the underlying design decisions.

The extension would require changing the front-end dramatically as the Gloss library we are using would need to be greatly altered or another method of graphical display used.

Add internal boundaries

The addition of internal boundaries refers to being able to add obstacles for the simulated fluid to interact with. This would require an additional stage for the velocity field and also an additional step prior to the start of the simulator to allow the user to input the obstacles. The related works that we will be looking at in Chapter 6 offer a few design ideas for dealing with these internal boundaries. Most promising of which is discussed by Wu et. al. in Section 6.4, as their base implementation is similar to our own.

Alleviate 'numerical dissipation'

One of the advertised drawbacks to Stam's fluid simulation algorithm is that the energy of the fluid dampens quickly. Stam's suggestion for remedying this is to use a technique known as *vorticity confinement* which involves pushing lost energy back into the simulation.

Wu et. al. once again have also implemented this extension to their own simulation. Calculating the vorticity confinement as part of the 'addition of forces' step in the simulation.

CHAPTER 6

Related Work

What follows in this section is a sampling of some of the many solutions that exist to the problem of rendering fluids. Each of these samples will be looked at briefly to give an overview of existing solutions.

6.1 *Particle-Based Fluid Simulation for Interactive Applications*

Müller et. al. in *Particle-Based Fluid Simulation for Interactive Applications* [MCG03] give an interesting solution using the particle-based Lagrangian viewpoint. Their solution is a means to "simulate fluids with free surfaces", which means they are attempting to simulate the dynamics of a fluid, such as water, moving and reacting like a liquid.

By using a Lagrangian model, Müller et. al. posit that because the particles move with the fluid the advection component can be removed. This simplifies the implementation as the simulator need only compute the pressure, diffusion and external forces components. Additionally, the Lagrangian particles have a constant mass, so mass conservation from the Navier-Stokes equations is built into the data structure and can be omitted from the calculations.

For combining the particles together to form the liquid, Müller et. al. recommend the use of Smoothed Particle Hydrodynamics (SPH), which is an interpolation method for particle systems. SPH uses a smoothing kernel to combine the Lagrangian particles into a body, or

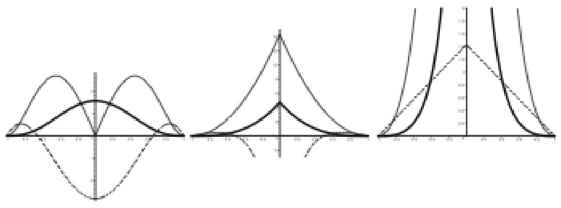


Figure 6.1: Smoothing kernel examples from [MCG03]

multiple bodies as the case may be. As the smoothing kernel affects the stability, accuracy and speed of the simulation, the selection of a smoothing kernel requires careful analysis to find which will give the best results.

The different smoothing kernels in Figure 6.1 describe different means to smooth the particles based on the distance between them, with the darker lines showing the kernels.

The fluid simulator also simulates the surface of a fluid as it interacts with another fluid. To determine where the surface is the authors use a colour field, which is a multi-dimensional array labelling locations including the fluid as 1 and other locations as 0. The surface is therefore locations of value 1 which have a neighbour with value 0. The modelling of a surface also introduces a surface tension coefficient which is calculated based on the two fluids that are interacting at the surface. To visualise and render the surface Müller et. al. suggest two methods: point splatting and marching cubes.

At the time of printing, the point splatting method still needed more work. However, using marching cubes allowed for good visualisation of the fluid's surface. Marching cubes works by calculating the iso surface, a surface which wraps over the points in the fluid to generate connectivity, by creating a static grid and identifying which cells contain surface particles. The grid cells are then recursively traversed to find neighbouring cells to generate the iso surface. To prevent cells from being visited more than once a hash table is maintained to keep track of visited cells.

An interesting solution to simulating and rendering a fluid is given in the paper, and boasts some good benchmarking figures, gaining 20fps running on a 1.8GHz Pentium IV PC with a nVidia GForce 4 graphics card. The simulator gets good results with detecting and tracking the edges of a fluid, and is a good example of the workings of a simulator using the Lagrangian discretisation method. The main drawback to this method lies in the use of SPH which requires analysis on which smoothing kernel to use, due to the large impact the smoothing kernel has on the efficiency and stability of the simulator.

This method shows an alternative implementation to our own, however it would be unsuitable for use with the Repa array due to the mesh being irregular and non-static in shape.

6.2 *Animation and Rendering of Complex Water Surfaces*

In *Animation and Rendering of Complex Water Surfaces* [EMF02] Enright et. al. describe a means to generate photorealistic water, or similar fluid, surfaces that can be incorporated into existing Navier-Stokes water simulations. This method is not useful if we want to include it with user interaction as it requires a considerable length of time to render each frame. Published results indicate approximately 11 minutes per frame. Although the method would provide a useful means to generate more realistic water effects for animation.

The described method concentrates on calculations at the surface, relying on a pre-existing simulator to do the general fluid calculations. To track the surface of the fluid the paper proposes an algorithm called the *particle level set method.* This method involves defining an area surrounding the actual surface, and within this area particles are placed. These particles are used as error correction for the surface as it evolves over time.

Error correction occurs when the surface of the fluid is inaccurate due to the coarseness of the computational grid not detecting sudden changes in the surface. The first step for error correction is defining if and where errors have occured. Each particle placed in the surface area has a radius which is defined dynamically as the simulation proceeds and the particle's location changes relative to the surface. If the particle's radius no longer overlaps the surface then it is said to have escaped. Any particles that have escaped either pull or push the surface, depending on which side of the surface they are supposed to be on.

To also assist with evolving the surface, the fluid's velocity field is extrapolated over the fluid's surface and into the air near the surface. This extrapolation helps with boundary conditions when processing the fluid's movement.

Incorporating the particle level set method with an existing solver is straightforward. Before the usual calculations for the velocity field are performed, the velocity field is first extrapolated over the surface into the air and the error correction for the fluid surface is performed. The original Navier-Stokes procedure is then performed on the velocity field.

6.3 *Visual Simulation of Smoke*

Fedkiw et. al. explore fluid simulation with a tangential approach to that of Stam in [Sta99]. *Visual Simulation of Smoke* [FSJ01] instead of using the Navier-Stokes equations like our other related works, uses the incompressible Euler equations. The reason for using the Euler equations is due to the specialised application Fedkiw et. al. are applying it to. The simulator described in [FSJ01] is specifically targetted at rendering smoke. Due to this specialisation, the simpler Euler equations can be used and still achieve the wanted results.

The Euler equations are similar to the Navier-Stokes equations except that the Euler equations describe ideal fluids, fluids that ignore the viscosity component that features in the Navier-Stokes equations. The Euler equations of motion are:

$$\nabla \cdot \mathbf{u} = 0 \tag{6.1}$$

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla) - \nabla p + \mathbf{f} \tag{6.2}$$

These equations describe the motion of the fluid. For the smoke simulation the density and temperature of the smoke needs to be calculated. The equation for density in this algorithm is much simpler than in [Sta99] while the equation for temperature is an equation not found in Stam's:

$$\rho = -(\mathbf{u} \cdot \nabla)\rho \tag{6.3}$$

$$T = -(\mathbf{u} \cdot \nabla)T \tag{6.4}$$

As can be seen from the equations, these values are just advected by the velocity field defined in (6.2).

The authors use Stam's semi-Lagrangian method for solving the advection component, $-(\mathbf{u} \cdot \nabla)$, to keep the solution stable. They do note, however, that the stability of the simulator relies on the forcing term, $\mathbf{f}$, being kept below a certain threshold, although the exact threshold is not specified.

The forcing term explicitly contains two forces in this model, as opposed to Stam's which describes generic external forces [Sta99]. The first of these forces is buoyancy defined as:

$$\mathbf{f}_{buoy} = -\alpha\rho\mathbf{z} + \beta(T - T_{amb})\mathbf{z} \tag{6.5}$$

Where T_{amb} is the smoke's ambient temperature; $\mathbf{z}$ is the direction of the force, making it go up, and α and β are constants defined by the developer to make the equation behave as is wanted. The equation basically defines that the smoke is pulled down by its density and pulled up depending on how much warmer the smoke is than the ambient temperature.

The second forcing term, vorticity confinement, is a method to inject energy back into the simulation that is lost using Stam's stable fluids algorithm. Stam's algorithm suffers from numerical dissipation, meaning that it loses energy quicker than is natural. As mentioned in Chapter 5 our own simulator suffers from this numerical dissipation and one of our aimed extensions was to incorporate vorticity confinement into the simulator as recommended by Stam in [Sta03].

Implementing vorticity confinement involves analysing the field to find locations where additional energy should be injected and then injecting it. To determine the amount to inject Fedkiw et. al. use the equation for vorticity in incompressible fluids:

$$\omega = \nabla \times \mathbf{u} \tag{6.6}$$

The $\nabla \times \mathbf{u}$ operation is also referred to as 'curl' and is discussed further in Appendix B.

Determining the location to insert the vorticity is done by moving over the field and using:

$$\mathbf{N} = \frac{\eta}{|\eta|} \tag{6.7}$$

to find the direction from lower vorticity concentrations to higher vorticity, which is the direction we want to add vorticity. Where η is the change in magnitude of vorticity. $\mathbf{N}$ is then used with the vorticity calculated in (6.6) to form the force to be inserted:

$$\mathbf{f}_{conf} = \epsilon\ h\ (\mathbf{N} \times \omega) \tag{6.8}$$

Where h is the length of the simulator grid, used to scale the force when the simulator gets finer grained, and ϵ is a programmer defined constant to control the amount of detail added into the flow.

The method for vorticity confinement used in [FSJ01] has direct consequences on our own implementation. We can use this method for implementing the vorticity confinement extension to the simulator by adding one extra traversal of the velocity field, calculating ω and $\mathbf{N}$.

The generated force, f_{conf}, is used when performing the 'addition of forces' stage of the velocity field.

6.4 *An improved study of real-time fluid simulation on GPU*

In *An improved study of real-time fluid simulation on GPU* [WLL04] Wu et. al. takes a similar viewpoint to us, that using parallelised infrastructure is an excellent way to gain additional performance. How Wu et. al. differ is in their use of graphical processing units (GPUs) to do the processing rather than central processing units (CPUs).

Wu et. al. use the Navier-Stokes equations but also add buoyancy and fluid temperature to increase the realism of the simulator. Buoyancy is added as one of the forces acting on the fluid, and the fluid temperature adds an additional equation to be processed similarly to the density equation:

$$\frac{\partial \rho}{\partial t} = -(\mathbf{u} \cdot \nabla)\rho + k_\rho \nabla^2 \rho + S_\rho \tag{6.9}$$

$$\frac{\partial T}{\partial t} = -(\mathbf{u} \cdot \nabla)T + k_T \nabla^2 T + S_T \tag{6.10}$$

The density field equation, (6.9), is the same as we are using and is defined by Stam in [Sta99]. The temperature equation, (6.10), is structured very similarly to the density equation and Wu et. al. process it similarly. Similar to [MCG03], Wu et. al. also allow for arbitrary internal boundaries.

To solve the Navier-Stokes equations, Stam's stable fluids are used, but adapted by adding in buoyancy, fluid temperature and pressure. To begin, the forces are added to the field from the previous time step:

$$\mathbf{u}^* = \mathbf{u_t} + (\mathbf{f}_{buoy} + \mathbf{f}_{conf} + \mathbf{f}_{user}) \cdot \Delta t \tag{6.11}$$

With $\mathbf{f}_{buoy}$ being the buoyancy force, $\mathbf{f}_{conf}$ is the vorticity confinement force used to inject energy into the simulation which is lost using Stam's method, $\mathbf{u_t}$ being the velocity after the last time step, $\mathbf{u}^*$ is the intermediate velocity after this stage and $\mathbf{f}_{user}$ being forces inserted by the user interacting with the fluid.

The fluid is then advected:

$$\frac{\partial \mathbf{u}^*}{\partial t} = -(\mathbf{u}^* \cdot \nabla)\mathbf{u}^* \tag{6.12}$$

And diffused:

$$\frac{\partial \mathbf{u}^*}{\partial t} = \nu \nabla^2 \mathbf{u}^* \tag{6.13}$$

Which are the same stages that we use in our own simulator as defined in [Sta99]. Wu et. al. extend this with the following two equations which are added in to correct the above calculations with the pressure field:

$$\nabla^2 p = \frac{1}{\Delta t} \nabla \cdot \mathbf{u}^* \tag{6.14}$$

$$\mathbf{u_{t+1}} = \mathbf{u}^* - \Delta t \cdot \nabla p \tag{6.15}$$

Their means of implementing the simulator uses code blocks known as fragment programs, which allows the developer to write code to customise the normal fixed graphics pipeline on the

GPU. Using these fragment programs give additional possibilities for parallelism, as modern GPUs have additional pipelines for processing them.

To increase parallelism and decrease the need for synchronisation between parallel threads, Wu et. al. combine the density, velocity and temperature fields into a RGBA-4 channel. Four channels are used as each velocity cell is composed of two values. Allowing the values for these fields to be processed at the same time, rather than sequencing the parallel operations on each field, increasing the available parallelisation. Extra care does need to be taken when combining the fields for processing. As may be recalled from Chapter 3, the velocity field and density field are treated slightly differently. The velocity field passes through an additional projection stage to keep the simulation mass conserving.

The treatment of internal boundary conditions is similar to [MCG03]. A colour map is used to determine where the internal boundaries lie on the internal grid. The normal at the edges of these internal objects is calculated and discretised to one of eight directions, to coincide with one of the eight neighbours of the cell in the grid and make processing easier. An additional processing pass through the velocity field is added to specially process these internal boundary conditions.

To render all this, including accepting user interactivity, the following stages are followed:

1. Users add obstacles into the simulation and an 'obstacle texture', i.e. a colour map, is generated.

2. Normals at the surface of the obstacles are calculated and cells in the grid are categorised as being a surface, solid obstacle or fluid.

3. Fields are processed similarly to Stam's method:

   (a) Add sources

   (b) Advect fields

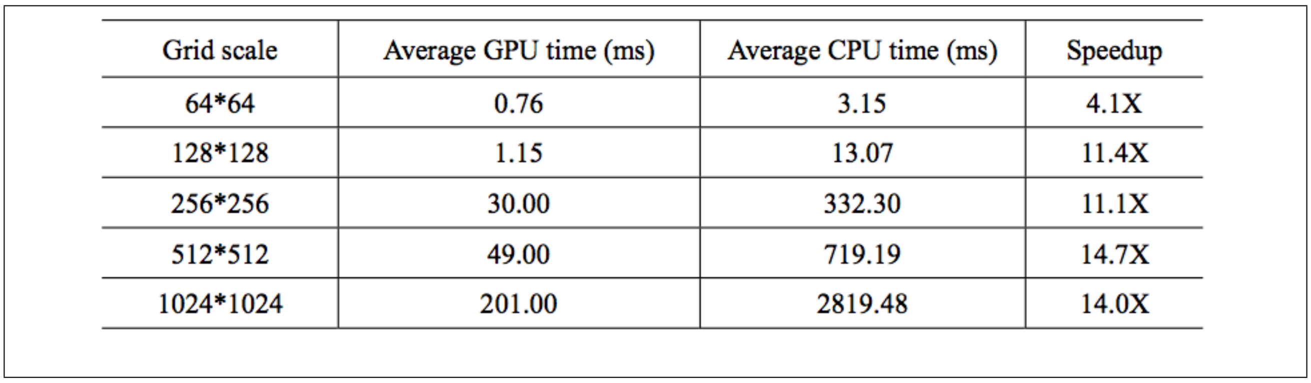
   (c) Diffuse fields

   (d) Apply mass conserving projection to velocity field

   (e) Revise velocity field with pressure equations mentioned earlier

4. Render the simulation using the density field

The published results of this method do show promise. However, the comparison of performance between the use of the GPU and CPU has some strange, unexplained anomalies.



Grid scale	Average GPU time (ms)	Average CPU time (ms)	Speedup
64*64	0.76	3.15	4.1X
128*128	1.15	13.07	11.4X
256*256	30.00	332.30	11.1X
512*512	49.00	719.19	14.7X
1024*1024	201.00	2819.48	14.0X

Figure 6.2: Results from [WLL04]

The provided table of time comparisons between the two implementations is shown in Figure 6.2. There is an inconsistent jump in GPU times between the 128*128, 256*256 and the 512*512 grids. This would imply that some memory bandwidth, or some other overhead, is being introduced with the increase in grid size. Unfortunately, the paper does not offer any insight into why these anomalies occur. The raw time these processings take is quite impressive, however, and imply that the simulation is running close to optimally.

The authors use a method quite similar to our own, although implemented on the GPU rather than the CPU. Their method for dealing with internal boundaries would be useful for our extension to the basic simulator we are implementing.

6.5 *Stable Fluids*

As we saw in Chapter 3, Jos Stam in [Sta99] and [Sta03] described an unconditionally stable fluid simulator algorithm. This algorithm is a fairly simple semi-Lagrangian method of simulating fluids and in the following chapters we will see how we use it for our own implementation.

Stam's implementation of his algorithm is a sequential implementation in C, and performs well enough for users to interact with the fluid in real-time. As was mentioned in Chapter 5 we hope to achieve competitive performance to Stam's implementation.

Having already looked at this work in depth in Chapter 3 we will not repeat it here, but direct the reader to the relevant chapter.

CHAPTER 7

Front-End Implementation

This chapter describes the implementation of the simulator front end, which is what the user sees and interacts with. For ease of development, a pre-existing Haskell library called Gloss was used. Gloss was developed by Ben Lippmeier. To use this in the simulator we needed to make some changes to it. The interface function `gameInWindow` will also be looked at, as it is the simulator's interface to the graphical rendering and takes the functions necessary to have the simulator operate as we want. In describing `gameInWindow` we will introduce the simulator's data structure, how it is displayed, and how the user interacts with it.

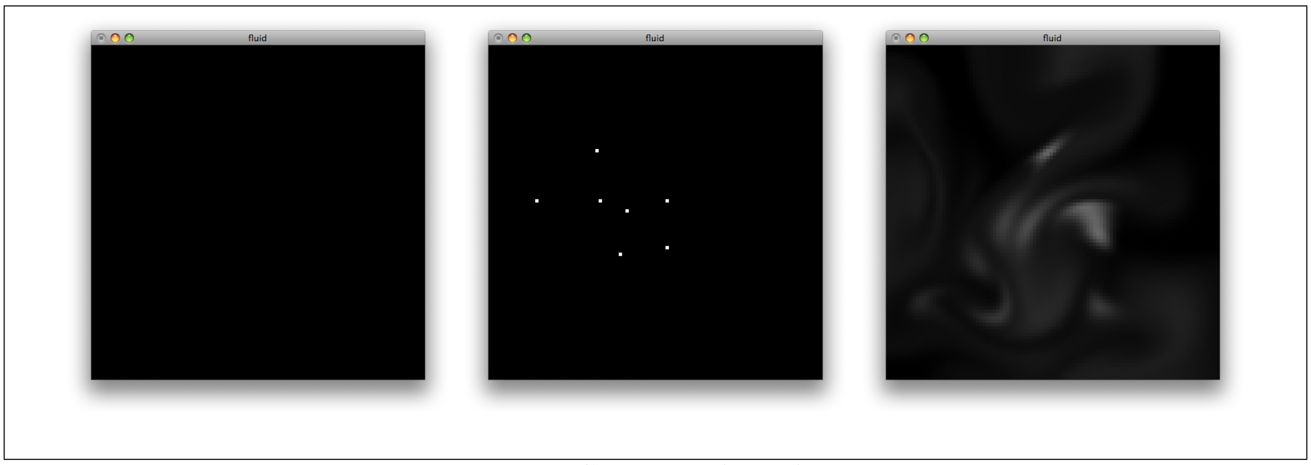


Figure 7.1: Samples of the front-end

```
data Picture
   | Circle    Float
   | Text      String
   | Color     Color     Picture
   | Translate Float     Float   Picture
   | Rotate    Float     Picture
   | Scale     Float     Float   Picture
   | Pictures  [Picture]
   ...
```

Figure 7.2: Examples of data constructors for `Picture` data type

7.1 Gloss

The Gloss library is a graphics library for 2-dimensional vector graphics and provides a pure abstraction over the OpenGL library, a commonly used library for graphical applications.

Gloss's abstraction defines a `Picture` data type to represent different vector shapes and objects, which can be manipulated by the basic geometrical operators scale, translate and rotate. A 'scene', the image that is displayed on the screen, is constructed by combining the different data constructors. Some examples of the constructors are show in Figure 7.2.

To initialise and run the window to display the `Picture` type constructors, Gloss provides a number of controlling functions which simplify and abstract the intricacies required to initialise OpenGL programs. Depending on how the developer wants the application to behave, the interfaces provide an increasing amount of customisation of the underlying system. These control interfaces are:

- `displayInWindow`
- `animateInWindow`
- `simulateInWindow`
- `gameInWindow`

The simplest interface, `displayInWindow`, takes a name, initial size, initial position, background colour and `Picture` to display. The `animateInWindow` function changes from simply taking a `Picture` to taking a function which when given a time since the program started outputs a `Picture` to be used.

The simulateInWindow and gameInWindow functions add a pervading world model which allows data to be retained between simulation steps. To accommodate this the interface takes an initial world model, a function that when given a model outputs a Picture and a function which moves the model forward one 'step'. The interface is also paramaterised by the number of steps to perform per second. Finally, gameInWindow adds an argument which is a function to handle user interaction events.

For our implementation we used gameInWindow as it allowed us to retain data between simulation steps using the world model and also allows the user to interact with the simulation, meeting one of our aims outlined in Chapter 5. We will discuss our use of gameInWindow in further detail in Section 7.3. But before this, we will look at an addition we needed to make in Picture to allow us to display the simulation.

7.2 Bitmap addition to Gloss

In Stam's algorithm there are two fields used for the simulation. One is the velocity field which expresses the movement of the fluid, and the other is the density field which simulates the particles in the fluid. For displaying the simulator we show the density field, which is represented by a 2-dimensional array of floating point values. We decided to render the density field by using a bitmap, a 2-dimensional array of pixel values, as it has the same structure as the density field.

To use the bitmap, we added a Bitmap type constructor to the Picture type as shown below:

```
data Picture
    ...
    | Bitmap Int Int (Ptr Word8)
    ...
```

Which takes in the width and height of the bitmap and a pointer to the raw image data, using Haskell's pointer to foreign data: Ptr.

The implementation for rendering Bitmap was based on the library that Gloss forked from, ANUPlot. Our initial implementation was based on the rasterised image Picture type constructor, which involved writing pixel data directly to the display buffer. This method was scrapped as it did not allow the Bitmap to be manipulated by Translate, Scale and Rotate and therefore would be contradictory to how the rest of the library was designed.

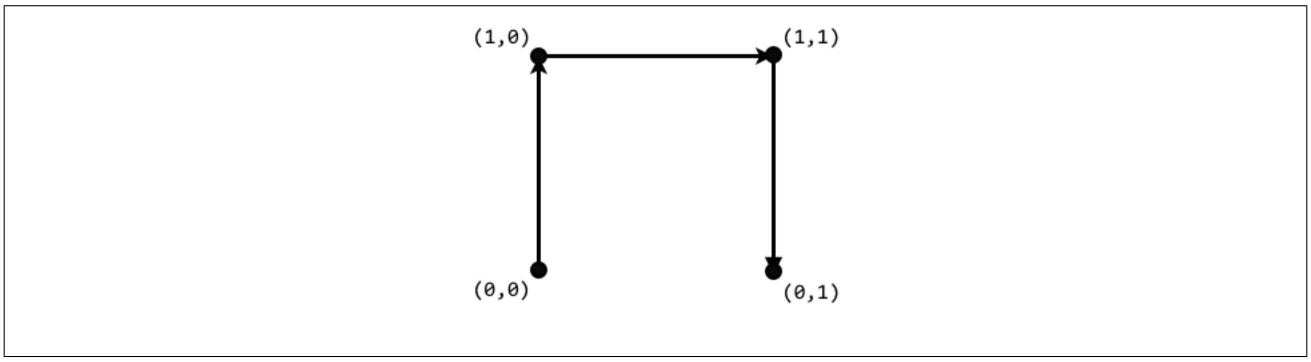


Figure 7.3: Order to specify corners in textured polygon

The final rendering method for `Bitmap` used OpenGL's textured polygon, so that it can be manipulated similarly to the other shapes in Gloss. However, there were two points of interest in implementing `Bitmap` with the textured polygon method that we will mention now.

The first is in how OpenGL reads the pixel data we pass to it. To represent a pixel we are using a 32-bit word which contains the four values red, green, blue and the alpha value. The three colours are mixed to form the colour that is wanted and the alpha value determines the opacity of the mixed colour. Each value is an 8-bit word. This format for storing the pixel data is referred to as RGBA, which also indicates the order these values are typically stored in. However, in OpenGL it reads the values in the opposite order, ABGR, which causes strange colouring to occur in displayed bitmaps if given in the RGBA order. To help interfacing with related libraries, like bmp a bitmap file loading library, we should not force responsibility of using ABGR ordered image data to the user of the Gloss library. Instead we add a function that reverses the order of the 8-bit words in each 32-bit word in the `Bitmap` constructor.

The second point of interest occurs when applying the texture from the pixel data to the polygon. When drawing the polygon it is important to draw the corners in the correct order otherwise the texture will appear distorted, for example it may be rotated the wrong direction or reflected about an axis. This does not require additional computation to solve, but requires some care in the implementation. Figure 7.3 shows the necessary order for the bitmap to be displayed correctly.

7.3 Interface with the user

We now look at our use of `gameInWindow`, focusing on how we display the simulation after each step and how users interact with the simulation. To begin, we introduce our world model, which will be looked at closer in Chapter 8. We have included the type signature for `gameInWindow` in Figure 7.4 as a reference.

```
gameInWindow
   :: forall world
   .  String                        -- Name of display window
   -> (Int, Int)                    -- Initial size of window
   -> (Int, Int)                    -- Initial position of window
   -> Color                         -- Background colour
   -> Int                           -- Simulation steps to take per second
   -> world                         -- Initial world
   -> (world -> Picture)            -- Function to convert world to Picture
   -> (Event -> world -> world)     -- Function to handle user events
   -> (Float -> world -> world)     -- Function to move world forward one step
   -> IO ()
```

Figure 7.4: Type signature for `gameInWindow`

Not including the `forall world` quantifier, which specifies the range available to the `world` type variable, the first five arguments are used simply to initialise the display window's settings. Following these, the `world` type variable allows the user to represent the world model however they choose. Our world model representation will be introduced briefly in Section 7.3.1. Next `gameInWindow` takes a function that when given a world model will give the `Picture` to display it, which we look at in Section 7.3.2. The penultimate argument is the function to handle user events, which we will discuss in Section 7.3.3. Finally, `gameInWindow` takes a function to move the simulation forward one step, which will be the focus of Chapter 8.

7.3.1 World model representation

For a full description of the world model, and the related design decisions we refer you to Chapter 8. As an overview, the world model is represented as a record data type and is defined as shown in Figure 7.5.

The `Model` record consists of: two arrays, the density field and the velocity field; storage for any new sources for both fields; storage to retain where the user last clicked, which we will discuss in Section 7.3.3, and a counter for how many steps have passed so far for benchmarking. The use of this last portion of `Model` will be explained in Chapter 9. The two fields for the density and velocity represent the fluid itself, as described in Chapter 3, and are an array of floats and float tuples respectively.

```
data Model
   = Model
   {
     densityField   :: DensityField
   , densitySource  :: Maybe (Source Float)
   , velocityField  :: VelocityField
   , velocitySource :: Maybe (Source (Float, Float))
   , clickLoc       :: Maybe (Int, Int)
   , stepsPassed    :: Int
   }
```

Figure 7.5: World model representation

7.3.2 Displaying the world model

The display function, now that we have seen the world model representation, can be concretely defined as `display :: Model -> Picture`. For displaying the simulation we only use the density field, and must convert its float values into 32-bit words to be passed into the `Bitmap` constructor for `Picture` described in Section 7.2.

To convert the density field into the form necessary for display we follow a number of stages:

1. Convert the array values from floating point numbers to 8-bit words, which are integer values between 0 and 255. These values are stored back into a Repa array using Repa's `map` function.

2. The new array is then extended using slices into the third dimension to represent the red, green and blue pixel values. Giving it extent `(Z:.height:.width:.3)`.

3. The alpha values for each pixel are set to always be 255 and are appended in the third dimension to the extended array.

4. The extended and appended array is then converted into a `ByteString` and passed into `Bitmap`.

As each of these stages is being performed by Repa arrays they are all parallelised, increasing the overall performance that the display function can achieve.

7.3.3 User interaction

Gloss provides the infrastructure for detecting and categorising different user events, such as input from the mouse or keyboard. We provide Gloss a function to handle these different user events when we call `gameInWindow`, which then updates the world model depending on the event. The function takes the detected event and the current world model as argument:

```
userEvent :: Event -> Model -> Model
```

The different user events are encapsulated by the `Event` type. To detect a specific user event that we wish to respond to, we pattern match on the given `Event` for the particular key being pressed and respond accordingly. The main events we are looking for are:

1. Left-button mouse click
2. Right-button mouse click
3. ‘q’ key

Responding to the ‘q’ key press just quits the simulator. The more interesting interactions occur when a mouse button event is detected, which adds a new density or velocity to the simulation.

When a mouse button event is detected we must determine where in the window the click took place. The number plane for click locations has the origin centered in the middle of the window, however the simulation has the origin at the bottom left corner of the window. Therefore, to determine the location in the array that a click took place, we map the click location to the array location. Once done we can perform the appropriate action depending whether it was a left- or right-button mouse click.

On a left-button mouse click the user is adding a density, and so `Model` stays the same except that we add the location in the array the user clicked on to `densitySource` in the `Model` definition in Figure 7.5, so that it can be added into the density field next time the simulation reaches the ‘addition of forces’ stage of Stam’s simulation pipeline (for more details of this stage please see Chapter 3).

The addition of velocities is done by right-clicking with the mouse. As the mouse cursor is moved around the simulation, the direction of movement is detected and a velocity is added to the starting location appropriately and the old starting location is updated to where the cursor is now. This continues until the right-mouse button is released. To maintain where the click was over multiple steps in the simulator, the previous location is stored in the world model in `clickLoc`.

CHAPTER 8

Back-End Implementation

Our back-end implementation is based heavily on the algorithm outlined by Jos Stam in [Sta99] and [Sta03]. However, our implementation uses the functional programming language Haskell, and have parallelised the algorithm with the use of the parallel array library Repa.

In this chapter we will discuss how the back-end, the simulator proper, was implemented. As part of this we will explain design decisions made during development and how performance of the simulator was increased. To begin we will discuss the world model introduced in Chapter 7 in more detail, and look at the ‘pipeline’ for processing the velocity and density fields. We will then look in detail at the individual stages which make up the pipeline before concluding with how performance was increased and some of the losses to declarativeness that were necessary for gaining enough performance to compete with Stam’s C implementation.

8.1 World model representation

As was mentioned in Chapter 7, the world model for the simulation consists of: two fields, a density and velocity field; two storages for new source locations, one for each field; the last click location; and the number of steps which have passed so far. For reference the definition for the `Model` type is shown in Figure 8.1. We will discuss these components in detail, except for `stepsPassed` which we will discuss in Chapter 9, and `clickLoc` which was discussed in Chapter 7. The design decisions for each will also be explored. First, however, we must discuss the type class that was developed for the two different field elements.

```
data Model
   = Model
   {
     densityField   :: DensityField
   , densitySource  :: Maybe (Source Float)
   , velocityField  :: VelocityField
   , velocitySource :: Maybe (Source (Float, Float))
   , clickLoc       :: Maybe (Int, Int)
   , stepsPassed    :: Int
   }
```

Figure 8.1: World model representation

```
class (Elt a) => FieldElt a where
   (~+~)      :: a    -> a     -> a
   (~-~)      :: a    -> a     -> a
   (~*~)      :: a    -> Float -> a
   (~/~)      :: a    -> Float -> a
   useIf      :: Bool -> a     -> a
   negate     :: a    -> a
   addSource  :: a    -> a     -> a
   zero       :: a
```

Figure 8.2: Type class `FieldElt` for categorising field element types

8.1.1 Field elements

In order for the stages performed on both the density field and velocity field to be written once, a type class had to be written that specified all the basic operations that needed to be performed on the field elements for each stage. This type class, `FieldElt`, allowed us to cut down on the amount of error-prone code in the simulator with only a few primitive class functions.

Figure 8.2 shows the definition of this class, which has two instances:

- `FieldElt Float`: for density field elements
- `FieldElt (Float, Float)`: for velocity field elements

With just the basic arithmetic operations: `(~+~)`, `(~-~)`, `(~*~)` and `(~/~)`; and a few helper functions shown in Figure 8.2 we were able to generalise the stages of the simulator for both fields. One note about the arithmetic operators `(~*~)` and `(~/~)`, it may be noticed that the second argument for these functions is a `Float` rather than `a`. This is so that we can generally define multiples of the elements for both fields.

8.1.2 The fields

To represent the two fields of our fluid we use two Repa arrays. In keeping with the generality of our `FieldElt` type class we define a polymorphic type:

```
type Field a = Array DIM2 a
```

With the `DensityField` type from our `Model` definition in Figure 8.1 defined as:

```
type DensityField = Field Float
```

And the `VelocityField` type defined as:

```
type VelocityField = Field (Float, Float)
```

`Field a` is used in the type definition of our fluid functions to allow for the polymorphism, with `FieldElt a` forming a type class restriction on which values of `a` can be accepted. This generality can cause problems for performance which we will discuss in Section 8.5.3.

8.1.3 Sources

When a new density or velocity is added by the user it is first placed in one of the source storages in `Model`. `Source` is defined as:

```
data Source a = Source DIM2 a
```

That is, `Source` is a polymorphic data type which is constructed by a position, `DIM2`, and a value to be inserted at that position.

Once the user has inserted a new density or velocity, it sits in the storage until the next 'Addition of forces' stage for the appropriate field is called.

8.2 Processing pipeline for fluid

In Chapter 3 we described the 'pipeline' for manipulating the fluid each time step. We repeat it here in Figure 8.3 for reference. This pipeline is slightly different in practice for both the density field and the velocity field. Our density pipeline is simplified in implementation:

$$\mathbf{w}_0(\mathbf{x}) \overbrace{\rightarrow}^{\text{add forces}} \mathbf{w}_1(\mathbf{x}) \overbrace{\rightarrow}^{\text{diffuse}} \mathbf{w}_2(\mathbf{x}) \overbrace{\rightarrow}^{\text{advect}} \mathbf{w}_3(\mathbf{x})$$

Whereas the velocity stage is increased in complexity:

$$\mathbf{w}_0(\mathbf{x}) \overbrace{\rightarrow}^{\text{add forces}} \mathbf{w}_1(\mathbf{x}) \overbrace{\rightarrow}^{\text{diffuse}} \mathbf{w}_2(\mathbf{x}) \overbrace{\rightarrow}^{\text{set boundary}} \mathbf{w}_3(\mathbf{x}) \overbrace{\rightarrow}^{\text{project}} \mathbf{w}_4(\mathbf{x})$$

$$\mathbf{w}_0(\mathbf{x}) \; \overbrace{\rightarrow}^{\text{add forces}} \; \mathbf{w}_1(\mathbf{x}) \overbrace{\rightarrow}^{\text{diffuse}} \mathbf{w}_2(\mathbf{x}) \overbrace{\rightarrow}^{\text{advect}} \mathbf{w}_3(\mathbf{x}) \overbrace{\rightarrow}^{\text{project}} \mathbf{w}_4(\mathbf{x})$$

Figure 8.3: Original pipeline from [Sta99]

$$\frac{\partial \mathbf{u}}{\partial t} = \mathbf{P}(-(\mathbf{u} \cdot \nabla)\mathbf{u} + \nu \nabla^2 \mathbf{u} + \mathbf{f}) \tag{8.1}$$

$$\frac{\partial \rho}{\partial t} = -(\mathbf{u} \cdot \nabla)\rho + \kappa \nabla^2 \rho + S \tag{8.2}$$

Figure 8.4: Navier-Stokes equations for velocity (8.1) and density (8.2)

$$\mathbf{w}_4(\mathbf{x}) \; \overbrace{\rightarrow}^{\text{set boundary}} \; \mathbf{w}_5(\mathbf{x}) \overbrace{\rightarrow}^{\text{advect}} \mathbf{w}_6(\mathbf{x}) \; \overbrace{\rightarrow}^{\text{set boundary}} \; \mathbf{w}_7(\mathbf{x}) \overbrace{\rightarrow}^{\text{project}} \mathbf{w}_8(\mathbf{x}) \; \overbrace{\rightarrow}^{\text{set boundary}} \; \mathbf{w}_9(\mathbf{x})$$

As can be seen, the velocity field is where the majority of the work is done by our simulator. The set boundary stage that is repeated throughout the pipeline is to ensure that the velocities on the edge of the simulation point inwards. As may be noticed, the project stage has been repeated also. This repetition is at the recommendation of Stam in [Sta03] as the advection stage is more accurate if it is being applied to a mass conserving field. In keeping with Stam's implementation, for each time step we calculate the velocity field and then the density field. This design limits parallelism, but allows more direct comparison of performance between implementations.

Each of these stages will be looked at and any design decisions that were made during the development of them will be mentioned.

8.3 Stages

8.3.1 Addition of forces

As may be recalled from Chapter 3, the addition of sources corresponds to the $\mathbf{f}$ or S component of the derived Navier-Stokes equations, repeated in Figure 8.4. After a user has inputted a new source for the density field or the velocity field, it sits in the corresponding type in `Model` until the `addSources` function is called. This function takes the position and value from the appropriate source, depending on whether we are in the velocity's add sources stage or density's, and adds it to the appropriate cell in the field.

Because Repa is designed to operate on entire arrays rather than on individual elements we cannot change just the one location without traversing over the entire array as there are no operations to affect such a small subset of the array.

8.3.2 Diffusion

Implementing the diffusion stage uses an iterative linear solver to approximate the diffusion and viscosity for the density and velocity field respectively. This linear solver is also used for the projection stage and so we will look at it in isolation of these stages. Implementing the linear solver separately from the diffusion and projection stages lowered the amount of locations for code faults and later allowed for performance analysis to be concentrated in a smaller area.

Recalling from Section 3.3.2 the approximation equation:

$$\mathrm{x}_{i,j} = \frac{\mathrm{x}^0_{i,j} + \mathrm{a}(\mathrm{x}_{i+1,j} + \mathrm{x}_{i-1,j} + \mathrm{x}_{i,j-1} + \mathrm{x}_{i,j+1})}{1 + 4\mathrm{a}} \tag{8.3}$$

The ‘a’ coefficient is calculated from the multiplication of the diffusion/viscosity rate and the square of the length of the field, scaled by the size of the time step. Remember, this equation is an approximation of $\nu\nabla^2\mathbf{u}$ and $\kappa\nabla^2\rho$. That is, (8.3) is approximating the multiple of the rate and the Laplacian coefficient (∇^2) being applied to the field value.

Approximating this equation through the iterative solver increases the accuracy, approaching the true value as the number of iterations approach infinity. However, we do not have infinite time to gain this accuracy and a compromise between accuracy and time must be made. Using Stam’s research 20 iterations of the solver tended to give smooth results. Through our own experimentation using less iterations tended to give jagged and unnatural looking edges during diffusion. Using more iterations distribute small fractions of the density to cells further away from the central density, making for a smoother gradient of densities. For this reason we kept the number of iterations at 20, the same as Stam’s. For general purposes this number of iterations can be kept static, however if being applied to systems with fewer computational resources or when the calculations need to be done faster, the number of iterations can be lowered at the sacrifice of some accuracy.

8.3.3 Advection

The advection stage, as was described in Chapter 3, involves backtracking the flow of the density or velocity that would end up at the center of the grid cell and linearly interpolating the four surrounding grid cells to get the approximation of what would have been there. Implementing this process is fairly straight-forward, but does require some care.

The first step to this stage is to determine the continuous point that the density/velocity would have been one time step ago based on the velocity at the location, illustrated in Figure 8.5. The discretised surrounding grid points are then calculated, and based on the original

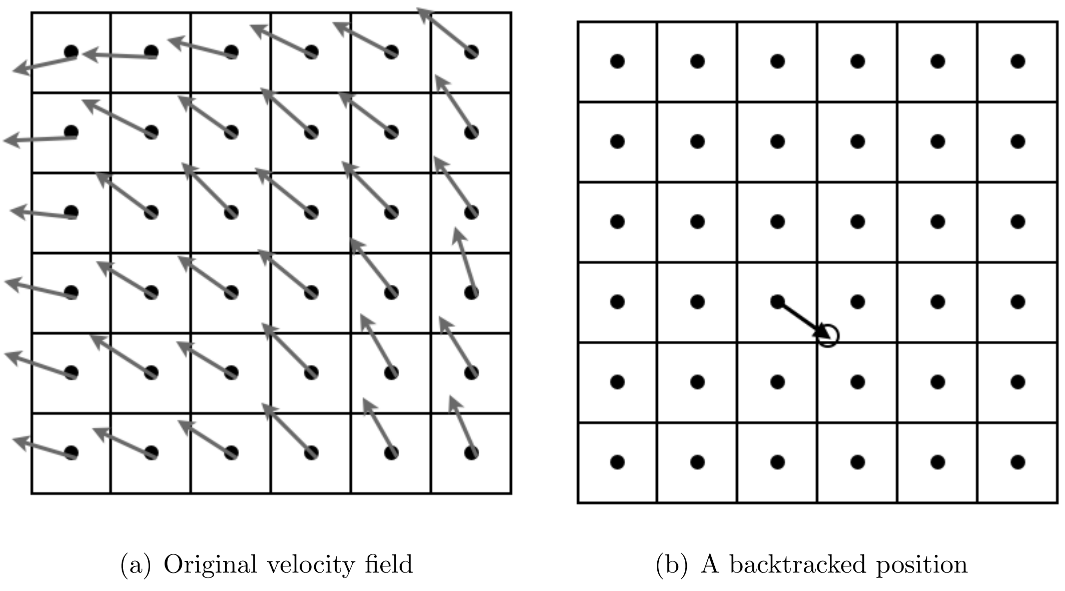

(a) Original velocity field (b) A backtracked position

Figure 8.5: Original velocity field and a backtracked position

continuous point's proximity to these values a ratio is calculated for use when performing the linear interpolation. Figure 8.6 shows the code for calculating this. In this code, `u` and `v` are the velocity at the current position of the array that we are operating on, and `dt0` is a scaling factor formed by multiplying the time step and the length of the side of the simulation.

Take note of the lines `i0 = truncate x` and `j0 = truncate y`, these lines convert the continuous values `x` and `y` to integer values by taking the floor. An early version of our implementation used `round` which rounds the float, this resulted in the fluid exponentially increasing in mass as the simulator took it's values from the wrong four grid cells.

We use `traverse` as the function running the parallel operations, this is because the backtracking function requires knowledge of the current position so as to calculate where to backtrack to.

8.3.4 Projection

Projection involves calculating the gradient field and subtracting it from the current velocity field to create the new mass conserving velocity field.

Calculating the gradient field involves first generating a very approximate partial derivative of the velocity field by taking the average of the vertical and horizontal values around each grid cell in the original velocity field and dividing by the length of the simulation. This field is then passed to the iterative linear solver. Similarly to the diffusion stage we iterate the solver 20 times for this approximation for the same reason as mentioned before, this number of iterations gives a good balance between accuracy and time constraints. The gradient field

```
-- backtrack densities to point based on velocity field
--  and make sure they are in field
x = checkLocation $ ((fromIntegral i) - dt0 * u)
y = checkLocation $ ((fromIntegral j) - dt0 * v)

-- calculate discrete locations surrounding point
i0 = truncate x
i1 = i0 + 1

j0 = truncate y
j1 = j0 + 1

-- calculate ratio point is between the discrete locations
s1 = x - (fromIntegral i0)
s0 = 1 - s1

t1 = y - (fromIntegral j0)
t0 = 1 - t1
```

Figure 8.6: Code for calculating backtracked position and the values for linear interpolation

and the original velocity field are then traversed together, with the gradient field values being averaged and subtracted from their corresponding velocity field values.

Our implementation uses Repa's `fromFunction` to create the first approximation of the gradient field. As mentioned, this is passed through the linear solver which we discuss next, and then we use the `traverse` function in subtracting the result of the linear solver with the original field. The `fromFunction` and `traverse` functions are both parallel operations so the main bottleneck with this stage is the coordination that needs to occur between each parallel operation, as the whole array is required to be computed before the next operation occurs.

The reason for using `traverse`, as opposed to a more specialised array operation, is the same as for advection: we needed access to the position of the cell we were operating on. Other Repa functions are too specialised to allow the flexibility.

8.3.5 Linear solver

Linear solver and set boundary, which we explain in the next section, are the two operations which account for the majority of processing time in the simulator. Before performance analysis was performed on the functions, calls to the linear solver function accounted for approximately

66% of the operational time. The reason for this is because the function is called a large number of times per time step and each call requires two traversals of the array.

Our original implementation of the linear solver used one traversal of the array applying an operation similar to that described in (8.3), however it is generalised to allow use with both diffusion and projection, to become:

$$\mathrm{x}_{i,j} = \frac{\mathrm{x}^0_{i,j} + \mathrm{a}(\mathrm{x}_{i+1,j} + \mathrm{x}_{i-1,j} + \mathrm{x}_{i,j-1} + \mathrm{x}_{i,j+1})}{c} \tag{8.4}$$

This implementation traversed the array and performed the following steps:

1. Grab the neighbouring cells
2. Add them together
3. Multiply them by a
4. Add the original value
5. Divide by c

When the neighbour cells were grabbed we needed to check to ensure that the index we were attempting to access was inside the array. While this is only an issue with grid cells on the edge of the array there was no way of checking this and so the checks needed to be made on every cell.

While developing the simulator some developments to Repa were made, the most significant to our implementation was the extension to include stenciling operations which we described in Section 4.2. As the stenciling operation could only be applied to one array at a time and the $\mathrm{x}^0_{i,j}$ component of (8.4) requires the addition of the original value, while the other values are updated as the linear solver iterates, we needed to add a second pass over the array each iteration. To do this we added a `zip` operation over the two arrays which added the neighbouring cells, multiplied by a, and the original value. The result was then divided by c. Even with the additional traversal after using the stencil operations we were able to more than double the performance of the simulator, as using the stenciling removed the bounds checking we were performing before.

The operation of the final linear solver worked as follows:

1. Initialise the `Stencil`
2. Apply the stencil to the array

3. Zip the result with the original array

4. Repeat operation until performed appropriate number of times

8.3.6 Set boundary

To prevent the velocity from moving either the density, or itself, off the side of the simulation after the various stages in the velocity pipeline, we apply the `setBoundary` function to the velocity field to keep the velocities on the edges pointing inwards, simulating a physical boundary.

The set boundary function has three cases for the grid cell positions: edges, corners and interiors. For the interior case the function does not affect the velocity in the cell; in edge cases the velocity is reflected, pointing inwards; and in corner cases the average of the neighbouring edge cells is taken.

We had two ideas for implementing this stage of the velocity field pipeline. The first was a naive traversal of the array, testing for when we encounter corner cases or edge cases and leaving the cell alone otherwise. Upon reaching one of the edge or corner cell cases the function would alter the value accordingly. This implementation is fairly slow due to the number of checks that need to be done to detect these cases, however it is easier to understand the operation workings.

The second idea for implementing set boundary involved using Repa's slices of arrays functionality. Performing set boundary in this manner involved the following steps:

1. Slice off the edges of the array

2. Perform the appropriate action depending whether it is an edge or a corner

3. Reconstruct the array with these new values for the edges and corners

Slicing off the edges of the array is performed by the `backpermute` function which creates a new array whose elements are just the edges and corners. The `backpermute` function is defined in Figure 8.8. It takes: the extent of the result array, in this case the extent is `(Z :. 4 :. width)` where `width` is the length of a side of the simulator; a function to map the indices of the resulting array to what they correspond to in the original array, and the original array. A diagram of this process is shown in Figure 8.7. It may be noticed that the corner cells are repeated in the extracted array. This is for ease of definition and in practice the corner cells are only processed once.

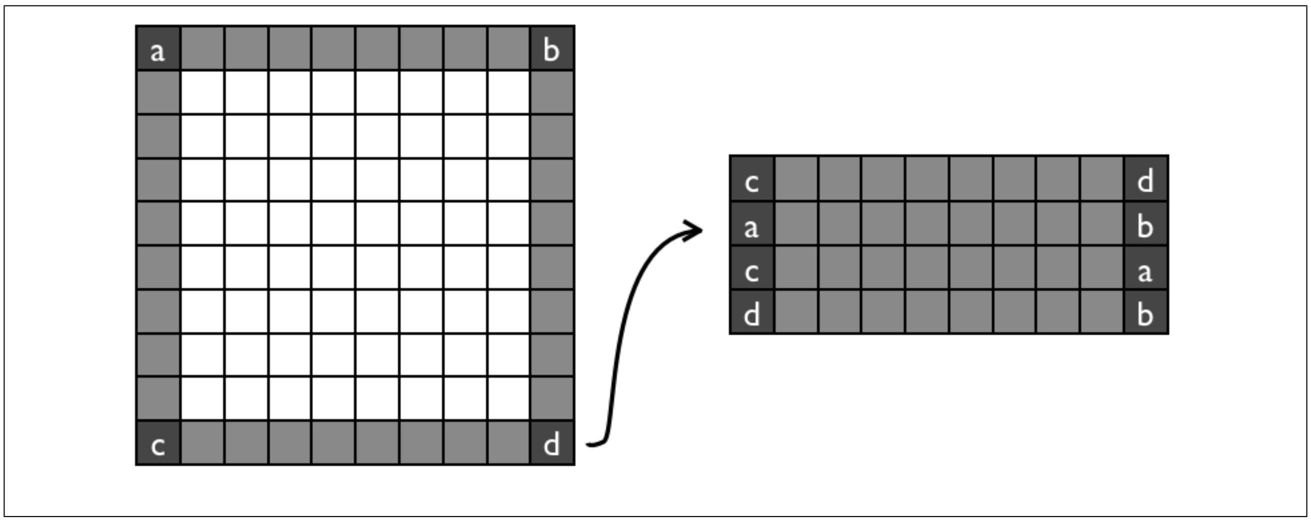


Figure 8.7: Diagram of `backpermute` extracting edges

Once we have generated the array of edge and corner velocities we perform the appropriate operations on them to make sure they are pointing inwards.

Finally, the new edge and corner values are combined with the original interior values using the `backpermuteDft`. As can be seen from the type definition in Figure 8.8 `backpermuteDft` is defined very similarly to `backpermute` except that it takes an additional array, for default values, and the index mapping function outputs a `Maybe sh` rather than just `sh`. The `backpermuteDft` function operates similarly to `backpermute` except that when the mapping function returns `Nothing` the value from the additional array is used. For our purposes we give the extent of the original velocity field and the original velocity field for the default values. The mapping function outputs the appropriate index in the edges array if the index given is on an edge, otherwise it returns `Nothing` and the original velocity field value is written to the new array.

This second method proved to give much better performance even though it still traverses over the whole array when reconstructing the array. There are still a number of checks being performed at every grid location, however the number is considerably lower than our first design.

8.4 Putting the pipeline together

When constructing the pipelines for the velocity field and the density field, as well as some of the intermediate arrays within the stages, care needs to be taken when using Repa's operations due to Haskell's laziness.

As was described in Chapter 4 the `force` function is provided to allow the developer control over when to force the parallel operations to take place. However, due to Haskell's laziness even this operation can be wrapped in a thunk for later processing. This can have disastrous

```
backpermute
   :: forall sh sh' a
   .  (Shape sh, Shape sh', Elt a)
   => sh'           -- Extent of result array.
   -> (sh' -> sh)  -- Function mapping each index in the result array
                    --    to an index of the source array.
   -> Array sh a   -- Source array.
   -> Array sh' a

backpermuteDft
   :: forall sh sh' a
   .  (Shape sh, Shape sh', Elt a)
   => Array sh' a        -- ^ Default values
   -> (sh' -> Maybe sh) -- ^ Function mapping each index in the result
                          -- array to an index in the source array.
   -> Array sh  a        -- ^ Source array.
   -> Array sh' a
```

Figure 8.8: Type definitions for `backpermute` and `backpermuteDft`

performance consequences in the simulator as it means in each time step most of the array operations are stored up until the computer needs them to display the updated simulation state on the screen. The operations are then performed at the point in time when they are already required to be displayed. This proves to hamper the performance of the simulator considerably.

Detection of this problem came from another development in Repa that occurred while developing the simulator. Detection of the `force` applications building up was added to Repa and a warning outputted when it was found.

Solving this requires the use of another Repa function: `deepSeqArray`. This function forces the evaluation of the elements in the array when used in conjunction with the `force` function by creating a dependance on not just the array as a whole but on the array elements as well. Doing this does detract from the declarativeness of the code, as can be seen from the `densitySteps` function in Figure 8.9. To ensure that the `force` operations are not building up, a `deepSeqArray` is performed on any arrays that a function depends on before calling the function.

Upon solving this, apart from giving a boost in performance, it also made the simulator more scalable when increasing the number of cores used in processing. However, as mentioned,

```
densitySteps :: DensityField  -> Maybe (Source Float)
             -> VelocityField -> DensityField
densitySteps df ds vf = df'
   where
      df' = df2 `deepSeqArray` vf `deepSeqArray` (advection vf df2)
      df2 = df1 `deepSeqArray` (diffusion df1 diff)
      df1 = df  `deepSeqArray` addSources ds df
```

Figure 8.9: Definition of `densitySteps` with `deepSeqArray` obscuring clarity

it obscures the clarity of the code. This issue is inherent in Haskell's laziness, rather than a fault of the Repa library, and a means of solving this while keeping the code declarative is not currently known.

8.5 Increasing performance

Once the simulator was implemented it ran considerably slower than the C implementation and a number of additions to the code needed to be made to increase the performance. These additions did not change the algorithms being used, but tended to obscure the code's readability. However, they were necessary for the simulator to get competitive performance to the C implementation. The first addition we have mentioned in Section 8.4 and will summarise here, the others are required for the Haskell compiler, GHC, to generate more efficient code.

8.5.1 `force` and `deepSeqArray`

As mentioned in Section 8.4, to ensure that the parallelised operations were executed in a timely fashion we needed to add `force` and `deepSeqArray` calls after a call to a Repa operation. This did increase the scalability and performance of the simulator considerably, however this did not fully eliminate our performance problem relating to the execution of the parallel operations. We will discuss the additional problem that occurred in Section 8.5.4.

8.5.2 Inlining

Pragmas are messages to the compiler to alter its default behaviour in specific sections of code. The `INLINE` pragma when attached to a function increases the chances that the function will be placed in the body of any function that calls it. By doing so it removes the overhead of the function call while still allowing us to modularise our code during development. Using the

```
{-# SPECIALIZE linearSolver :: Field Float          -> Field Float
                            -> Float -> Float -> Int
                            -> Field Float #-}
{-# SPECIALIZE linearSolver :: Field (Float, Float) -> Field (Float, Float)
                            -> Float -> Float -> Int
                            -> Field (Float, Float) #-}
linearSolver :: (FieldElt a)
             => Field a
             -> Field a
             -> Float
             -> Float
             -> Int
             -> Field a
```

Figure 8.10: Example of `SPECIALISING` pragma

`INLINE` pragma has the biggest effect when being performed on small functions that are called often, like the functions defined in the type class `FieldElt`. Adding these calls gives a boost to performance by encouraging GHC to remove these function calls to trivial function.

Inlining pragmas were also applied on all the helper functions for the stages, except for the linear solver's top function. The reason inlining cannot be applied to this function is because it uses self-recursive calls to perform the iterative solving that boosts the accuracy of the solver. When using self-recursive calls, GHC cannot determine if the cycle will end or if it will recursively call the function indefinitely. Instead of attempting to inline linear solver, we use the `SPECIALISING` pragma, which we discuss next.

8.5.3 Specialising function

The generalising of the simulation stages that we discussed in Section 8.1 created additional function calls to look-up the type information to determine which type was being used in the function call. Once again we used a pragma to solve this issue. This time the `SPECIALISING` pragma was applied to every top-level stages' function and the linear solver.

The specialising pragma makes GHC generate copies of the function which have a specific type, removing the polymorphism. Figure 8.10 shows the use of the pragma with the linear solver function. As can be seen, the polymorphic `Field a` is replaced in the pragma with either `Field Float` or `Field (Float, Float)`. The code generator will now create two copies, along with the original polymorphic function, and will cut down significantly the overhead with calling these functions. This pragma was also applied to the top-level function for each stage in the simulator.

8.5.4 Pattern matching

The final addition to the simulator's code that we made gave the most significant boost to performance. However, it also created the most obscurity in the code. It was noticed in the intermediate code generated by GHC that after the self-recursive call to `linearSolver` a number of case matches on the internal structure of the array returned were being made. Regardless which structure was matched, the same result was always returned. This created a lot of unnecessary overhead when performing the `linearSolver` function.

The reason for this is GHC cannot be certain of how the Repa array type is being constructed and must cover all cases. Solving this requires the addition of explicit, strict matches for intermediate arrays that they only accept manifest arrays, that is Repa arrays that have had their elements evaluated. To be sure that these pattern matches don't fail, they should only be used after a `force` is applied to an `Array`. Adding these pattern matches tells GHC that there is a restriction on what constructors will be given, removing many of the checks GHC felt required to do and therefore giving a boost in performance. As a side note, it was after this addition to the code that the simulator began achieving better than C results.

This addition to performance came at a cost to declarativeness in the code. In Figure 8.11 we see how the pattern matching obscures the function's operations, especially compared with the already slightly obscured version in Figure 8.9.

As is common in applications, obscurity is added proportionally to the addition of performance. This obscurity is necessary for the simulator to meet our aim of competitiveness with the C implementation's performance and is introduced by a shortcoming in GHC. Because GHC cannot determine that the `Array` output from a call to `force` will always be of one structure, it cannot remove the checks to the structure, resulting in our need to add the pattern matching.

8.5.5 Conclusion

If not for the additional obscurity required to instruct GHC on how the program should be compiled to run efficiently, Repa would provide a clean and powerful means to implement the back-end of the simulator. As it currently stands, Repa can gain good performance but this comes at a cost of code clarity. With extensions to GHC's analysis of functions during optimisation, the use of Repa for parallel numerical calculations on arrays would be ideal.

```
densitySteps :: DensityField -> Maybe (Source Float) -> VelocityField
             -> DensityField
densitySteps df ds vf = df'
   where
      df'@(Array _ [Region RangeAll GenManifest{}])
         = df2 `deepSeqArray` vf `deepSeqArray` (force $ advection vf df2)
      df2@(Array _ [Region RangeAll GenManifest{}])
         = df1 `deepSeqArray` (force $ diffusion df1 diff)
      df1@(Array _ [Region RangeAll GenManifest{}])
         = df `deepSeqArray` (force $ addSources ds df)
```
Figure 8.11: Definition of `densitySteps` with `deepSeqArray` obscuring clarity

CHAPTER 9

Results and Performance

We have seen how our fluid simulator was developed, both the front-end and back-end, and have mentioned that we have met the aim of getting performance that is competitive to Stam's C implementation. We will now discuss the specifics of our benchmarking process and the results that were seen.

9.1 Benchmarking

Our benchmarks were performed using two 64-bit Intel Quad-core Xeon E5405's running at 2.0GHz, running Ubuntu 2.6.32. Stam's original C implementation was compiled using GCC 4.4.3 while our simulator was compiled with GHC 7.1. We compiled our simulator with both the default GHC back-end and with the LLVM back-end. When running the simulator, parallel garbage collection was turned off.

The benchmark run was a slightly modified version of each simulator, removing the graphical front-end of the simulator and adding in a counter for how many steps have passed. Once the counter reached a certain threshold, 100 in the case of our benchmarks, the program would terminate. Our measure of performance relative to the C implementation was the speed which each implementation completed the 100 steps.

With 8 processing cores available from the two Xeon's, our benchmarking was run using 1 to 8 threads of execution.

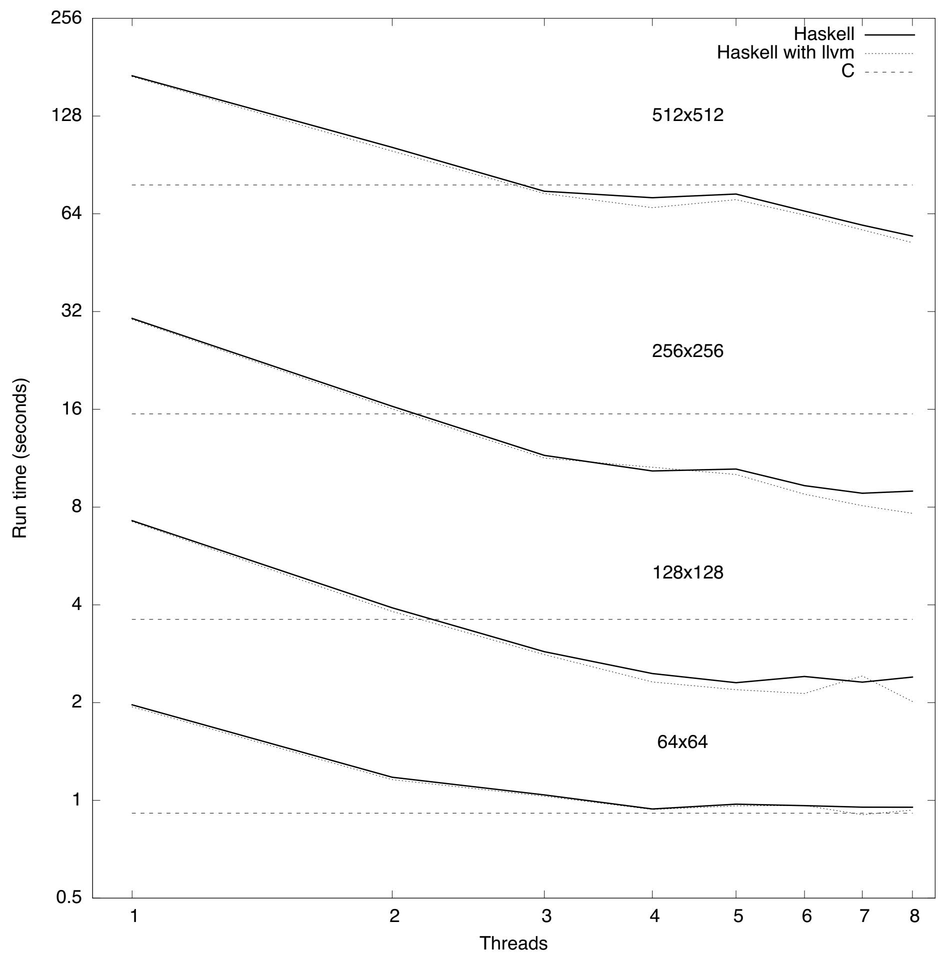


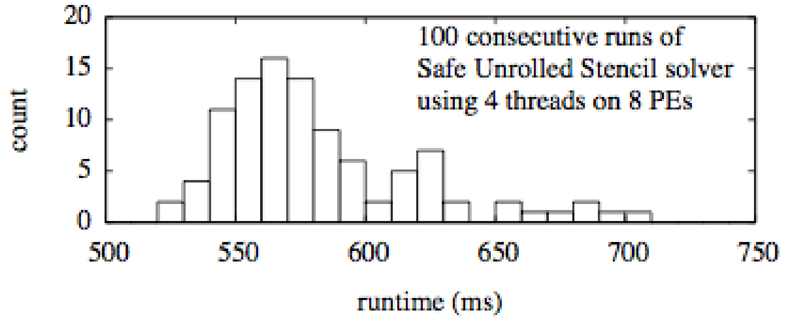
Figure 9.1: Performance of simulators with 1-8 threads

9.2 Results

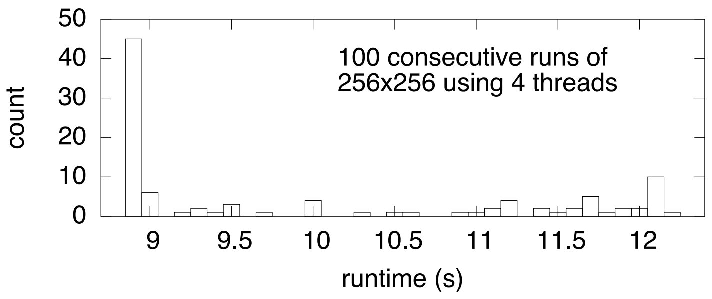
We now discuss the results of our implementation, both in benchmarking the performance of the implementation and compare the relative clarity of the code of our Haskell and the C implementations.

9.2.1 Performance

One of our major aims set out in Chapter 5 was to develop a simulator using Haskell that implemented the same algorithm as Stam's C simulator [Sta03] and ran competitively. As can be seen in Figure 9.1 when run on three or more threads our performance can beat the performance of the C implementation on the larger simulation sizes tested. For the smaller, 64x64, simulation there is not enough data to parallelise in order to beat the C version and coordination of the parallel threads becomes more work than is saved by parallelising. However, our implementation arguably runs competitively to the C implementation when run with 4 threads or more.



(a) [LKPJ11] variation



(b) Simulator's variation

Figure 9.2: Depiction of disparity in run times

There is still obvious room for improvement in the simulator's performance. When the number of parallel threads increase, the amount of time spent processing the data decreases. As this occurs coordination and synchonisation of threads becomes a higher proportion of processing time. Therefore, coordination can limit the scalability of an implementation if threads do not have enough work. Our simulator exhibits this behaviour as the run time does not increase linearly as the number of threads increase. This is behaviour that plagues many parallel implementations and increasing the scalability relies on giving more work to the threads while decreasing the amount of coordination that needs to occur. To gain greater scalability we would need to diverge from Stam's method of implementation to allow for more parallel operations between synchronisations of the threads. We will discuss an idea on how to achieve this in Section 10.1.

While performing the benchmarking, large fluctuations in run times were noticed when using 4 or more threads. These fluctuations reveal a large disparity between the optimal time and the worst time the simulator could run in. This behaviour was also noticed by Lippmeier et. al. in [LKPJ11] and was attributed to scheduling issues caused by shorter lengths of time of parallel computation as the number of threads increases. As our implementation uses stencilling in our most called function, `linearSolver`, it appears that our implementation is also suffering from this scheduling issue. However, the disparity in times does not exhibit the bell curve shape shown in [LKPJ11], repeated in Figure 9.2(a) for reference. To illustrate this we ran our simulator 100 times consecutively with a 256x256 simulation on 4 threads with a distribution of the run times shown in Figure 9.2(b). As can be seen from the graph, our simulator runs at an optimal time for the implementation close to the majority of the time however scheduling issues appear to create a large range in the run times. With the benchmark shown in Figure 9.2(b) the range was almost 3.5 seconds. Note that each bar refers to a range of 0.1 seconds. We attribute this less ordered variation of times to our simulator running many more parallel operations than in the paper. Creating more opportunities for scheduling to occur in the more common, and optimal, order.

An additional scheduling issue can be seen in Figure 9.1. When the simulator is run on 5 cores, there is typically an increase in the running time. This increase appears to imply that the threads are being moved between the two CPUs of the benchmarking machine, limiting the opportunity to exploit caching and increasing the amount of low-level communication required.

Even with the coordination issues, Repa can still out perform the C implementation if given enough data to parallelise between synchronisations of the arrays. Increasing the amount of parallel work per parallel operation is possible by altering the underlying data structure of the simulator. Doing so, however, would require diverging from Stam's method of implementation, limiting how much direct comparison between the two implementations. Future work to increase the absolute performance of the simulator is a definite possibility.

9.2.2 Code clarity

Our other aim, to develop a simulator that is declarative and retains its links to the equations which define it was not as successful. Gaining the performance required to beat the C implementation compromised the code clarity. To show an example of this we compare our implementation of a linear solver, our means to approximate a solution for diffusion and projection stages, with the linear solver from Stam's implementation.

Figure 9.3 shows our linear solver and Figure 9.5 shows the linear solver from Stam's solution. As can be seen, in our solution the meaning of the code is obscured through the pragmas and pattern matching necessary to coerce GHC to perform how we want it to. Removing these additions greatly increases the clarity, as seen in Figure 9.4.

The C version, while somewhat ambiguous in its variable naming, is clearly performing a loop over an array. Processing each cell using Stam's linear solver method. As C is a lower

```
{-# SPECIALIZE linearSolver :: Field Float -> Field Float
                              -> Float -> Float -> Int
                              -> Field Float #-}
{-# SPECIALIZE linearSolver :: Field (Float, Float) -> Field (Float, Float)
                              -> Float -> Float -> Int
                              -> Field (Float, Float) #-}

linearSolver :: (FieldElt a)
             => Field a
             -> Field a
             -> Float
             -> Float
             -> Int
             -> Field a
-- If linearSolver would not actually change anything by running, skip
linearSolver origF@(Array _ [Region RangeAll GenManifest{}]) _  !0  !_  !_
   = origF
-- If linearSolver has finished loop
linearSolver _     f@(Array _ [Region RangeAll GenManifest{}])  !_  !_  !0
   = f
-- Calculate intermediate array
linearSolver origF@(Array _ [Region RangeAll GenManifest{}])
                 f@(Array _ [Region RangeAll GenManifest{}]) !a !c !i
   = f `deepSeqArray` origF `deepSeqArray` f' `deepSeqArray` f'' `deepSeqArray`
   linearSolver origF f'' a c (i-1)
   where
      f''@(Array _ [Region RangeAll GenManifest{}])
         = c' `seq` force $ A.zipWith (zipFunc c') origF f'
      f'@(Array _ [Region RangeAll GenManifest{}])
         = force $ mapStencil2 (BoundConst E.zero) (linearSolverStencil a c) (force f)
      c' = 1/c
```

Figure 9.3: Our implementation of the linear solver

level language than Haskell, the operations being performed are a direct mapping to what is being done after compilation. This affords some succinctness for the implementation, in this particular application, in comparison to using GHC which requires adding some obscurity to specialise the code.

These additions to the code are only added so that GHC compiles the simulator to an efficient program, and so these shortcomings are attributed to shortcomings in the compiler. To be able to use the version in Figure 9.4 would require changes to GHC, namely better analysis of functions to identify these specialisations and being able to recognise that subsets of a type are outputted, specifically with the `force` function.

In summary, while using Repa allows for a relatively succinct and clear means to implement parallelism, obscurity needs to be added to help GHC recognise the specialised use of the

```haskell
linearSolver :: (FieldElt a)
             => Field a
             -> Field a
             -> Float
             -> Float
             -> Int
             -> Field a
-- If linearSolver would not actually change anything by running, skip
linearSolver origF _  !0  !_  !_
   = origF
-- If linearSolver has finished loop
linearSolver _     f  !_  !_  !0
   = f
-- Calculate intermediate array
linearSolver origF f !a !c !i
   = f `deepSeqArray` origF `deepSeqArray` f' `deepSeqArray` f'' `deepSeqArray`
   linearSolver origF f'' a c (i-1)
   where
      f'' = force $ A.zipWith (zipFunc c') origF f'
      f'  = force $ mapStencil2 (BoundConst E.zero) (linearSolverStencil a c) (force f)
      c' = 1/c
```

Figure 9.4: Cleaner version of the linear solver

```c
void lin_solve ( int N, int b, float * x, float * x0, float a, float c )
{
   int i, j, k;

   for ( k=0 ; k<20 ; k++ ) {
      FOR_EACH_CELL
         x[IX(i,j)] = (x0[IX(i,j)] + a*(x[IX(i-1,j)]+x[IX(i+1,j)]+
                                        x[IX(i,j-1)]+x[IX(i,j+1)]))/c;
      END_FOR
      set_bnd ( N, b, x );
   }
}
```

Figure 9.5: Stam's implementation of the linear solver [Sta03]

functions and types. This is not a catastrophic shortcoming, but users of Repa would benefit from extensions to GHC to add this analysis of functions.

CHAPTER 10

Conclusion

Our fluid simulator presented in this thesis shows that with parallelism a simulator written in Haskell can perform competitively to one written in C. With the current version of GHC some obscurity in the code must be introduced to gain this performance and so we still must sacrifice clarity for performance. However, some means to alleviate this compromise have been identified.

With some additions to GHC's analysis of functions during optimisation, we could achieve the aim of combining clarity with performance using the Repa library. Experience gained from implementing this fluid simulator has shown the ease with which parallelism can be incorporated into an algorithm using Repa, as long as some care is taken with how functions are structured, as we discussed in Chapter 8. The library does seem well suited for performance on large bodies of numerical data, as used by systems such as fluid simulators. With divergence from Stam's implementation, and in doing so increasing the underlying structure to be parallelised, using Repa should allow for excellent parallelised performance. With the increase in multi-processor computers, using Repa should make developing parallel systems more accessible to users. Using Repa also gives additional type safety that is not afforded by C, or similar languages traditionally used for high-performance systems.

Use of the Gloss library allowed for a simple implementation of the front-end of the simulator. However, if the simulator were to be extended to perform computation on 3-dimensional simulations then a new graphical library would need to be used to display the simulation.

10.1 Future work

Although we achieved one of our main aims, there are some shortcomings that could be addressed in future work. Additionally, the simulator has room for additional features, and development on it could be forked to apply it to specific areas of fluid simulation.

10.1.1 Extension of GHC

As was described, a major future work would be to enhance GHC's analysis of functions. Extending GHC to identify polymorphic functions that should be specialised, and to identify functions that only output a subset of constructors, could greatly decrease the amount of code required by developers to instruct GHC to behave in the optimal way by decreasing run-time checks on typing.

10.1.2 Extension of simulator

A number of extensions were originally planned for the simulator, which we described in Section 5.2.3, that would extend the functionality of the simulator. Due to additional time being used to increase performance, these extensions were not completed in the scope of this thesis but would be quite fitting for future work on the simulator.

10.1.3 Added parallelism

Our current implementation is developed so as to keep it as similar to Stam's implementation as possible while adding parallelism. By diverging from Stam's method to open up the possibility of additional parallelism, even better performance and scalability than demonstrated could be achieved. One method of achieving this would be to combine the fields into one and performing the stages on the density field and the velocity field at the same time, increasing the amount of parallelism while decreasing the amount of coordination required between stages.

10.1.4 Off-line rendering

Using Repa to parallelise operation of the fluid simulator became relatively more beneficial as the size of the simulation was increased. These large simulations are unsuitable for real-time interactions due to the absolute processing time being greater than is useful for interaction. However, performing these higher resolution renderings are useful for off-line rendering. Off-line rendering is when a simulation is run to completion before displaying, and is commonly used for animation. Therefore, future work could be performed to use Repa to develop a high-definition simulator for animation.

APPENDIX A

User Guide

A.1 Installing

The simulator can be obtained from the darcs repository by running:

```
$ darcs get http://patch-tag.com/r/blambo/fluid
```

With `$` indicating the command prompt.

Configuring, building and installing can be done using cabal. Alternatively the provided Setup.hs file can be used as such:

```
$ runghc Setup.hs configure --user
$ runghc Setup.hs build
$ runghc Setup.hs install
```

Ensuring that the installation directory is on your working path.

A.2 Running

Provided the simulator was installed as above, beginning the simulator should just require:

```
$ fluid
```

A number of configuration arguments can be passed to the simulator:

1. `width`: specifies the length of the dimensions of the fluid simulator.

2. `dt`: sets the time step, larger time steps progress the simulator faster but lower accuracy.

3. `diff`: sets the diffusion rate of the density field.
4. `visc`: sets the viscosity rate of the velocity field.
5. `windowWidth`: changes the default size of the display window.
6. `dens`: specify the amount of density added when clicking.
7. `vel`: specifies the scale of the velocity added when clicking.
8. `rate`: changes the number of steps per second the simulator aims to run at. Note that specifying the rate too high is detrimental to performance.
9. `maxSteps`: limits the number of steps the simulator will run in, useful for benchmarking
10. `batch-mode`: runs the simulator in 'batch-mode' which turns of the graphical front-end. Used for benchmarking.

A Haskell run-time argument that should be used to add parallel threads is:

```
+RTS -Nn -RTS
```

Replacing `n` with the desired number of threads to run. It is not recommended to use more threads then there are processing cores on the machine running the simulator.

An example of invoking the simulator, with diffusion and viscosity rate, and with 2 threads is:

```
$ fluid --diff=0.001 --visc=0.001 +RTS -N2 -RTS
```

A.3 Interacting

Once the simulator is running, interaction is done using the mouse. A left-click will add some density at the position pointed to by the mouse. Right-clicking will add some velocity, the direction of which is controlled by holding down the button and moving the cursor in the direction of flow.

To close the simulator, press the 'q' key.

APPENDIX B

Mathematical Background

The following explanations are referenced from [BFMF06].

Divergence

The divergence operator, $\nabla \cdot \mathbf{u}$, measures the convergence, or divergence, of the vectors in a velocity field. The operation is shorthand for (in 2-dimensions):

$$\begin{aligned}\nabla \cdot \mathbf{u} &= \nabla \cdot (u, v) \\ &= \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\end{aligned}$$

This operator is used in the calculation of advection in Euler and Navier-Stokes equations.

Curl

The curl operation measures how much rotation is occuring in a vector field around a point. It is a similar operation to the divergence operator, being shorthand for:

$$\begin{aligned}\nabla \times \mathbf{u} &= \nabla \times (u, v) \\ &= \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}\end{aligned}$$

This operator is used in calculating vorticity in the vorticity confinement technique described in Chapter 6.

Gradient

The gradient operator takes the partial derivatives of a function in each dimension, outputting a vector:

$$\nabla f(x, y) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right)$$

In the Navier-Stokes equations the gradient operator is used with the pressure component.

Laplacian coefficient

The Laplacian coefficient is a measure of the divergence of the gradient of a velocity field. It is the second order partial derivatives of each dimension added together:

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

Poisson equations

Laplace's equation is when the Laplacian, $\nabla^2 f$, equals 0. If it equals something nonzero it is referred to as a Poisson equation:

$$\nabla^2 f = q$$

An example of this is encountered when attempting to solve diffusion. Diffusion is actually referred to as a Poisson problem as it is multiplied by a scalar field, the diffusion rate.

Bibliography

[BFMF06] R. Bridson, R. Fedkiw, and M. Müller-Fischer. *Fluid Simulation.* SIGGRAPH 2006 Course Notes, April 2006.

[CLPJ$^+$07] M. M. T. Chakravarty, R. Leschinksiy, S. Peyton-Jones, G. Keller, and S. Marlow. Data parallel haskell: a status report. In *DAMP 2007: Workshop on Declarative Aspects of Multicore Programming.* ACM, ACM Press, 2007.

[CM92] Alexandre J Chorin and Jerrold E. Marsden. *A Mathematical Introduction to Fluid Mechanics.* Springer-Verlag, 3rd edition, 1992.

[EMF02] Douglas Enright, Stephen Marschner, and Ronald Fedkiw. Animation and rendering of complex water surfaces. In *Proceedings of SIGGRAPH 02.* ACM, 2002.

[FSJ01] Ronald Fedkiw, Jos Stam, and Henrik Wann Jensen. Visual simulation of smoke. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 15–22, August 2001.

[KCL$^+$10] G. Keller, M. M. T. Chakravarty, R. Leschinksiy, S. Peyton-Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in haskell. In *Proceedings of ICFP 2010: The 15th ACM SIGPLAN International Conference on Functional Programming.* ACM Press, 2010.

[LKPJ11] Ben Lippmeier, Gabriele Keller, and Simon Peyton-Jones. Efficient parallel stencil convolution in haskell. Submitted to ICFP 2011, 2011.

[MCG03] Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In D. Breen and M. Lin, editors, *SIGGRAPH Symposium on Computer Animation*, pages 154–159, 372. The Eurographics Association, 2003.

[Sta99] Jos Stam. Stable fluids. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 121–128, August 1999.

[Sta03] Jos Stam. Real-time fluid dynamics for games. Game Developer's Conference, 2003.

[WLL04] Enhua Wu, Youquan Liu, and Xuehui Liu. An improved study of real-time fluid simulation on gpu. *Computer Animation and Virtual Worlds*, 15:139–146, July 2004.